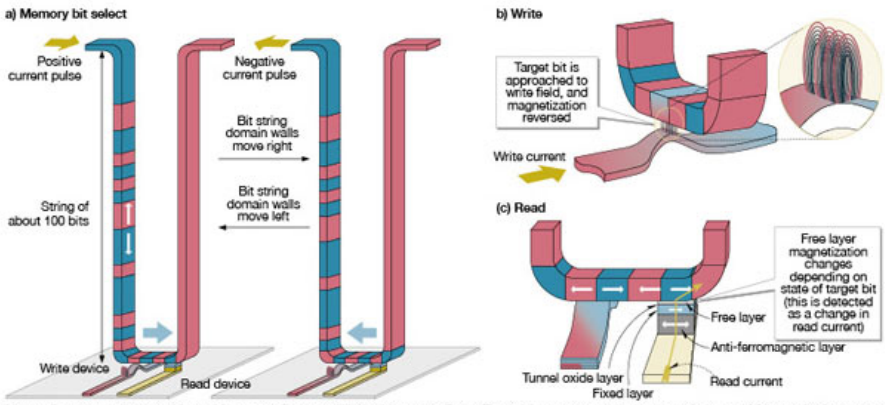
COL719 Project Report

Domain Wall Memory

Introduction

Domain-wall memory, also known as racetrack memory, is an experimental non-volatile memory which is being developed by IBM's Almaden Research Center. A 3-bit version of the memory was demonstrated in early 2008. Racetrack memory has the potential to offer a storage density higher than solid-state memory devices like flash memory.



Using Current Pulses to Move Magnetic Domain Walls Magnetic Race-Track Memory records a string of about 100 bits of information perpendicularly to the Si substrate for each read/write device. This means information density is about 100 times higher than MRAM. Operating principles are shown for access memory bit select (a), write (b) and read (c). When selecting the memory bit for access a current pulse is applied to the magnetic material, causing the magnetic domain wall to move. A positive current pulse will move the wall to the right in the diagram, and a negative one to the left. The quantity of pulses can be controlled for random access, with data read executed after the target bit has been accessed. (Diagram by *Nikkei Electronics* based on material courtesy IBM)

Implications of the Design

As mentioned in the previous section, the design offers a very high storage density. However, this comes at the cost of the memory access times and the access time is not as *random* as in the case of a DRAM or a similar memory design. Instead, the time taken is proportional to the distance between the memory addresses. This happens because the data is shifted to the "reading point" for every memory access.

Thus, compiler optimisation techniques that assume DRAM-like memory will not lead to the best performance if the memory is replace by a racetrack memory and hence we need alternative optimisation techniques to fully exploit the benefits of this memory design.

Proposed Plan of Action

Although the memory designs of DRAM and Domain Wall Memory require different kinds of memory access patterns to reduce the delays, they are still similar in the sense that they both require *spatial locality*. Although the meaning and requirement of this locality differs for both cases, simple modifications and extensions of compiler optimisation techniques can be made to work with Domain Wall Memory (this will avoid the need to reinvent the wheel from scratch).

Apart from the spatial locality optimisations, additional optimisations can be made in the direction of modifying the data storage techniques such as,

1. Having a dedicated cache in conjunction with the memory for storing large arrays which have random accesses - since not all arrays are used at the same time, migrating the array data to a cache (which offers equal latency on average) after reading it sequentially from the racetrack memory might help reduce the delays
2. Modifying the prefetching techniques - since modern processors tend to pre-fetch the data before it is required, this can lead to out-of-order accesses to the memory which might cause back-and-forth data movement in the racetrack memory
3. Explore the interaction between processor cache and memory better - since the memory access patterns matter a lot more than in the case of a DRAM, the interaction between the cache and the racetrack memory needs to be monitored more *strictly* and disallow (or reduce) memory accesses that are too far apart unless necessary

Problem Statement

There are multiple optimisation parameters that can be taken into consideration while designing a compiler for a racetrack memory. Some of them involve the memory access patterns, some involve the data storage techniques and some involve the interaction between the processor cache and the memory. The optimisation can also be done for a fixed code and pre-determined memory sizes. The exact problem that is taken into consideration for the current project is:

Given a CDFG using p different arrays with sizes $\{s_1, s_2, \dots, s_p\}$, find an optimal location to *place* the data for each array location such that the total cost incurreced by the Domain Wall Memory during the execution of the CDFG is minimised. The mapping function from array index to the memory location should require $O(1)$ memory for each array and should return the index in $O(1)$ time.

This formulation is considered for the project (after multiple iterations and discussions with Prof Panda) due to the following reasons:

1. The number of arrays and their sizes are fixed and known at compile-time - this is done to somewhat simplify the problem and to obtain the best possible results for all possible formulations given a certain CDFG. The results would be sub-optimal if the number of arrays and their sizes are left unknown.
2. The mapping function is required to be $O(1)$ in time - this is done to ensure that the memory access time is not affected by the mapping function. If the mapping function is not $O(1)$, and is instead linear, then the time taken to compute the address would be very large and would significantly reduce (or negate) the performance gains of using a DMW.
3. The mapping function is required to be $O(1)$ in space - if the mapping function requires linear number of bits to be stored, then storing this mapping function would itself require its own memory. This would lead to a cyclic dependance since the mapping function would require its own mapping function to optimally fetch (and compute) the function while minimising the total access time of the DMW on which the mapping function is stored.

Alternative Formulations

The above formulation is not the only possible formulation for the problem. Some of the other formulations that were considered are:

1. Given an assembly level code, modify (and re-order) the optimisation passes such that it reduces the memory access time for a DMW rather than a DRAM. This formulation was not considered since it would require a lot of changes to the existing optimisation passes and would require a lot of time to implement and test.
2. Consider modified DMW designs which have multiple read (and write) ports, thus increasing the scope for optimisations while making the problem harder. This formulation was not considered since the modifications, although they have been proposed in literature, the technology is still in its infancy and hence it would be better to first optimise the placement and mapping for a single port DMW.

Proposed Solution

For the chosen formulation, the solution that has been proposed is inspired from the intuitive idea of "array interleaving". A few simplifying assumptions were made that have been mentioned in the following section. The representation of the problem statement has been motivated from the ideas presented in "Efficient Data Placement for Improving Data Access Performance on Domain-Wall Memory" by X. Chen, et al.

Assumptions

The following assumptions have been made during the development of the solution:

1. All n-D arrays have been unpacked into multiple 1D arrays and each 1D array is treated independently, i.e., the total number of 1D arrays after the unpacking is a constant and hence the resultant mappings can take $O(p)$ space and $O(1)$ time.
2. All arrays have sizes that are of similar sizes and only differ by a constant or sub-linear factor. This ensures that the interleaving is simpler.
3. The CDFG was assumed to have already been optimised for spatial locality and hence the memory access patterns are already optimal. This assumption was made to simplify the problem and to focus on the mapping function rather than the memory access patterns.
4. The initial indices of the outermost loops are either known (constants) or can be computed in $O(1)$ time. This assumption makes arguing the correctness of the solution easier and it is also a reasonable assumption for the indices of the outermost loops.

Mathematical Formulation

A graph is built using the indices of the arrays (that can be computed) and edges are made between consecutive accesses. The algorithm is given as input a number k, which corresponds to the number of bins (or the maximum number of interleavings) that are allowed. Each bin then corresponds to a particular array and the mapping function corresponding to the bin b is of the form $f_b(i) = a * i + b$ where a, b are constants. The algorithm then tries to find the optimal interleaving of the bins such that the total cost incurred by the DMW is minimised.

For any given CDFG, the number of bins required will never exceed the maximum number of different variables in a DFG that involve accesses to arrays. Hence, the number of bins will always be a constant and hence the space requirement of the mapping function will be $O(1)$. Similarly, each f_b will take $O(1)$ time to determine the location of the corresponding array index.

Algorithm

The algorithm is given as input a number k and a CDFG. The algorithm then computes the number of bins required for the CDFG and then tries to find the optimal interleaving of the bins such that the total cost incurred by the DMW is minimised. The algorithm is given below:

```
BuildGraph(CDFG):
    G <- Graph()
    for DFG in CDFG: # DFGs on the outermost level of the CDFG
        indices <- GetInitialIndices(DFG) # gets the initial indices
```

```
                                        # of the outermost loops
        for access1, access2 in DFG:
            # generate the node considering only the stride
            # wrt previous index in the same array
            node1 <- (access1.array, access1.stride, access1.initial_index)
            node2 <- (access2.array, access2.stride, access2.initial_index)
            if (node1, node2) is not in G:
                G.add_edge(node1, node2)
            else:
                G[node1, node2].weight += 1 # increase the weight of the node
        updateInitialIndices(DFG) # propagate the initial indices
                                  # into the inner loops
        G <- G + BuildGraph(DFG, k) # recursively call the function
                                    # for the inner loops
    return G

MinimiseDMWAccessTimes(CDFG, k):
    G <- BuildGraph(CDFG)
    bins <- k
    functions <- {}
    while bins > 0:
        # find the path in graph G with the maximum weight
        # having atmost bins edges
        path <- FindPath(G, bins)
        # generate the mapping function for the path
        for node in path:
            functions.add(node.array, node.stride, node.initial_index)
        bins <- bins - path.length
    return functions
Copy
```

Evaluation

Infrastructure

Python files: `DMW.py`, `Mapping.py`, `loops.py` were made which make it easier to implement any mapping function and evaluate it on various loops that are available.

Results

10 out of 24 standard `Livermore Loops` were used to evaluate the proposed algorithm. Arrays of size 500 were used and the same CDFG was executed 10 times to also simulate the delays caused due to multiple executions with different data values.

The base cost is considered to be a linear mapping as it would be for a DRAM.

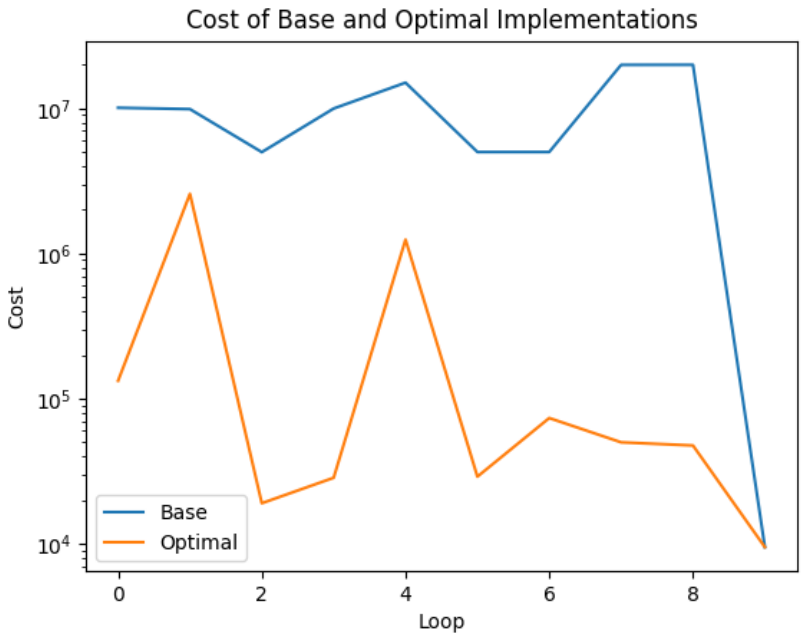
A huge improvement (over 99%) was observed for the simple loops that involved accessing multiple arrays. Some of the relatively complicated loops also showed a significant improvement (over 90%). The minimum improvement was 73.9% for a loop that involved slightly non-linear pattern of accesses.

Table and Plot

The results are shown in the following table:

Loop	Base Cost	Optimal Cost	Improvement
loop1	10110499	132627	98.688225%
loop2	9898767	2578833	73.947937%
loop3	5000499	18981	99.620418%
loop5	9989481	28445	99.715250%
loop7	15060499	1246761	91.721649%
loop11	5009481	28962	99.421856%
loop12	5010499	73401	98.535056%
loop19	19980001	49935	99.750075%
loop22	20000499	47481	99.762601%
loop24	9463	9463	0.000000%

The plot for comparing the base cost and the optimal cost is shown below:



Future Work

Improving Infrastructure

The infrastructure can be improved to allow more complicated and arbitrary mapping functions rather than simple mapping functions. Another improvement can be to allow for a different mapping policy which does not involve separating the memory into bins.

Relaxing Assumptions

The assumptions made initially can be gradually relaxed to obtain a more general solution. The following are some of the possible relaxations that can be made:

1. Instead of assuming that all arrays are of same sizes, the algorithm can be modified to partially include the path weights corresponding to the smallest array in that path. The remaining path weights can then be allocated to a different bin, however, this would then require non-constant number of bins but the number would still be small.
2. If the initial indices of each of the loops are not known, or they are not computable, then the algorithm can be made a probabilistic algorithm. It can try to simulate the CDFG execution for some values of the initial indices and then populate the graph. All initial values will not have equal probability and hence the simulation will ensure that those initial values are more likely which are practically more probable.
3. Multi-dimensional arrays can be incorporated by increasing the function computation time to $O(\log n)$ or $O(\sqrt{n})$ where n is the size of the array. This would slightly slow the mapping function computation but if it offers considerable improvement over the linear mapping, it would still be a suitable approach.