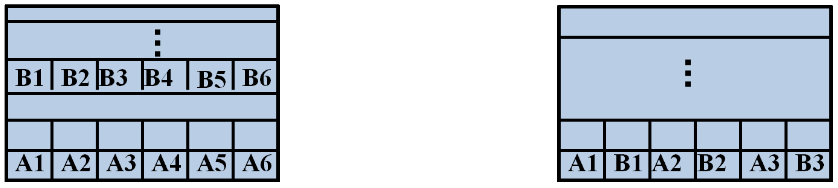
CSL728 (Compiler Design) Assignment 3: PCM Write Aware Data Layout Transformation

I. Introduction

Array Interleaving, a data layout transformation, has been explored with the objective of

- Reducing cache conflicts [1]
- Grouping selective memory accesses in vector architectures [2].

Figure 1 shows the data layout transformation due to interleaving.



(a) Conventional Storage (b) Interleaved Storage

Fig. 1: Effect of Array Interleaving Transformation on Data Layout

We wish to solve the array interleaving problem in the context of Phase Change Memory (PCM). Because of its unique characteristics, the interleaving solution for PCM may be different here than in the DRAM, cache, and vector register contexts.

PCM differs from DRAM in the following ways:

- Lower READ energy than DRAM
- Expensive WRITEs both in terms of latency and energy for PCM
- Limited lifetime because of the destructive write operations. Endurance, the number of writes that can be reliably programmed, ranges from 10^4 to 10^9.
- Row Buffer operation for PCM is different from that of DRAM.

A. DRAM/PCM Interface with Row Buffer

The PCM interface and operation can be assumed to be identical to that of DRAM (Figure 2). To access a data word that is not present in the row buffer, from the memory, an ACTIVATE command is issued to an appropriate memory bank. It is then read or written by issuing a READ or WRITE command respectively. Before the next row is fetched into the buffer for the DRAM bank, the row is pre-charged by a PRECHARGE command and written back to the memory array. In PCM, a PRECHARGE command is issued only if the row had a write operation.

Since the PRECHARGE commands cause actual writes to memory array and are costly, especially for PCM, these need to be minimized. Table I summarizes the different memory controller commands.

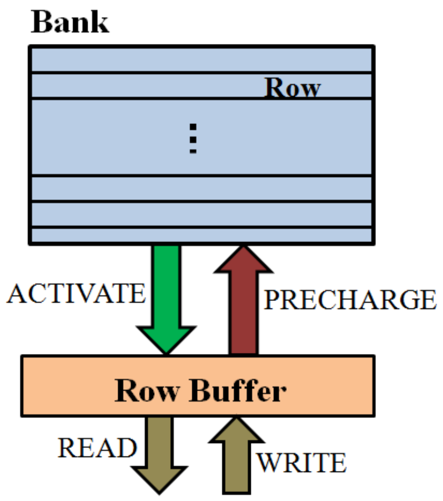


Fig. 2: DRAM/PCM Bank interface with the Row Buffer

Command	**Memory Operation**
ACTIVATE	**Memory Array Read**
READ	Row Buffer Read
WRITE	Row Buffer Write
PRECHARGE	**Memory Array Write**

TABLE I: Memory controller commands and their operations

Energies/Latencies for row buffer accesses are almost the same for both DRAM and PCM. The difference with these memories is in the way of row buffer operation. For DRAM, all the pages (rows) brought into the buffer are to be written back to main memory whenever a row is replaced. In PCM, a page in the buffer is written to main memory only if there was any WRITE operation in that row. Otherwise it can be replaced without any memory writes.

For example, consider the memory command trace in Figure 3.

The **ACTIVATE** command fetches the row (or page) from the memory to the row buffer while **PRECHARGE** command writes back the row to memory. For the same set of READs and WRITEs associated with pages, the sequences for DRAM and PCM are different because of the different row buffer operations. For the example above, we have
For DRAM : #MEM_RDs = 4, #MEM_WRs = 4
For PCM : #MEM_RDs = 4, #MEM_WRs = 1

B. Relative Latency/Power/Energy Numbers

We use the following normalized integer values for the READ/WRITE access latencies, powers, and energies for different memories [3].

Row 1 **ACTIVATE**
Row 1 READ
Row 1 READ
Row 1 **PRECHARGE**
Row 2 **ACTIVATE**
Row 2 READ
Row 2 **PRECHARGE**
Row 1 **ACTIVATE**
Row 1 WRITE
Row 1 **PRECHARGE**
Row 3 **ACTIVATE**
Row 3 READ
Row 3 READ
Row 3 **PRECHARGE**

(a) Sequence for DRAM

Row 1 **ACTIVATE**
Row 1 READ
Row 1 READ
Row 2 **ACTIVATE**
Row 2 READ
Row 1 **ACTIVATE**
Row 1 WRITE
Row 1 **PRECHARGE**
Row 3 **ACTIVATE**
Row 3 READ
Row 3 READ

(b) Corresponding sequence for PCM

Fig. 3: Memory command traces

$T_{PR} = 2, T_{PW} = 6,\ T_{DR} = T_{DW} = 1$
$P_{PR} = 1, P_{PW} = 3,\ P_{DR} = P_{DW} = 5$
$E_{PR} = 2, E_{PW} = 18,\ E_{DR} = E_{DW} = 5$
where,
T_{PR}, T_{PW} : READ and WRITE latencies for PCM
T_{DR}, T_{DW} : READ and WRITE latencies for DRAM
P_{PR}, P_{PW} : READ and WRITE powers for PCM
P_{DR}, P_{DW} : READ and WRITE powers for DRAM
E_{PR}, E_{PW} : READ and WRITE energies for PCM
E_{DR}, E_{DW} : READ and WRITE energies for DRAM

II. Determining Possible Candidates for Interleaving

In DRAM, for arrays A_1 and A_2 to be interleaved, they should have:

- Same Array Size
- Same access pattern i.e. $A_1[i]$, $A_2[i]$ or $A_1[2i+1]$, $A_2[2i+1]$
- Same lifetime

For PCM, in addition to above points, the arrays to be interleaved should have *same access type*. e.g. READ $A_1[i]$, READ $A_2[i]$. If we have READ $A_1[i]$, WRITE $A_2[i]$, then they should not be interleaved to reduce redundant writes to PCM. Selective bit writing may help in blocking redundant writes to some cells but requires extra hardware and will incur performance overhead, thus it may not be energy efficient.

III. Deliverables

Considering the differences in Row Buffer operation, our objective is to reduce the row buffer writes to main memory by an energy efficient data layout transformation. The interleaving decision should consider both the savings in terms of writes to PCM, which in turn refers to overall energy, and interleaving overheads, to account for interleaving decisions at various loop levels.

- Give a formal definition of this problem.
- Give a strategy for solving the array interleaving problem in such a way that either Latency or Energy is minimised.
- Explain the asymptotic complexity of your algorithm.
- Submit a design document detailing your strategy and test plan.
- Implement your algorithm using the LLVM infrastructure. You may use PIN for obtaining profile-based data when information cannot be easily obtained from code at compile-time.
- Write your own small test cases and make sure the implementation is inline with your expectation.
- Pick up C programs from anywhere, and use some of them as test cases. We may give you some test cases.
- Write a report on your findings.
- Give a demonstration of your implementation.

References

[1] Chidamber Kulkarni, Francky Catthoor, Hugo De Man. Advanced Data Layout Optimization for Multimedia Applications. *IPDPS* , 2000.

[2] Namita Sharma, Preeti Ranjan Panda, Francky Catthoor, Praveen Raghavan, Tom Vander Aa. Array Interleaving - An Energy Efficient Data Layout Transformation.

[3] Dhiman, G.,Ayoub, R. ; Rosing, T. PDRAM: A hybrid PRAM and DRAM main memory system. *DAC* , 664 - 669, 2009.

CSL728 Project Report

Nishant Kumar Himanshu Gangwar

November 15, 2014

1 Problem statement formulation

We assume that the program consists of m loops - $L_1, L_2, L_3.., L_m$ and n arrays - $A_1, A_2, A_3...A_n$. Let S be the set of all the arrays . During each loop L_i , there are a set of live arrays - denoted by S_i that represents the set of arrays that are considered for interleaving in loop L_i.

For a particular loop, L_i , there are various possibilities for interleaving the live arrays. Let P_i represent the set of all the interleaving possibilities and let P_{i_j} represent the j^{th} interleaving possibility.

P_{i_j} will partition the set of live arrays of L_i into $P_{i_j}^1, P_{i_j}^2$ ... with $P_{i_j}^k$ of the form $\{A_{i,j,k}^1, A_{i,j,k}^2 ...\}$, representing that these arrays are interleaved in this loop.

To formulate the problem , we define a graph in the following way. Suppose we represent each loop L_i by a level. So, there are m levels, and each level consists of nodes representing all the interleaving possibilities of L_i. A node has an associated cost with it , that represents the cost of memory accesses from the PCM [accounting for cache misses , cache hits , row buffer miss , row buffer hits] if the interleaving of the live arrays in the loop was of the form given by that node. A node in level i ,say P_{i_j} is connected to some nodes in the next level (say to $P_{(i+1)_k}$) , with an edge denoting that the interleaving between the loops , L_i and L_{i+1} is changed from P_{i_j} to $P_{(i+1)_k}$ at runtime.

Since changing interleaving costs need to be taken into account, so lets say that each edge from some P_{i_j} to $P_{(i+1)_k}$ has some associated cost that represents the cost of changing the interleaving of arrays from that given by P_{i_j} to that given by $P_{(i+1)_k}$.

So, now we have a graph - lets call it Array Interleaving Graph(AIG).

The problem then reduces to finding a path from some node in 1^{st} level to some node in m^{th} level, with the cost of the path being the sum of all the nodes and edges on the path.

So, the problem is effectively to find the best set of partitions - $P_{1_{K_1}}, P_{2_{K_2}}..P_{m_{K_m}}$ such that the following cost is minimized :

$$\begin{aligned} totalcost = cost(P_{1_{K_1}}) + transition(P_{1_{K_1}} \rightarrow P_{2_{K_2}}) + cost(P_{2_{K_2}}) \\ + transition(P_{2_{K_2}} \rightarrow P_{3_{K_3}}) + ... + cost(P_{m_{K_m}}) \end{aligned}$$

2 Assumptions

- We assume that there is only one level of cache in memory. This assumption helps in simplifying the calculation of the number of activates and precharges during the execution of the loop.

- We assume that no capacity or conflict misses occurs in the cache. Only compulsory misses are possible. This means that the cache misses only occur when the data is not in the cache. Also, since there are no conflict or capacity misses, the data will remain in cache till the program ends. This simplifies the calculation of the number of cache misses possible.
- In calculating the cost of an edge from one interleaving to another, we assume that we first de-interleave all the arrays and then interleave them. Finding the optimal way of changing the interleaving is quite difficult.
- While changing the interleaving we copy one array into a new one and so we assume that we are not limited in the amount of memory available.
- Whenever we need to change the interleaving of some array, we assume to insert a 'for' loop that basically copies the array into the new interleaving. But there can be cases where this can be avoided. So, say A, B are 2 arrays that are being written in the loop L_1 and we need to change the interleaving from say $\{\{A\}, \{B\}\}$ to $\{\{A, B\}\}$. This can be done by writing the arrays in the interleaved form in the loop L_1 itself. But we donot consider this and instead assume that a new loop is required that copies A, B into a new array in interleaved form.
- In calculating the cost for a given loop and interleaving, we approximate the number of activates and precharges required. So, for example consider the loop

  ```
  for(i=0;i<I;i++){
     A[i] = B[i]+C[i];
  }
  ```

  and the interleaving be $\{\{B, C\}, \{A\}\}$. Then if the cache line size is L, then after $L/2$ iterations of the loop, there will be a miss for B, C. This will bring the row corresponding to $\{B, C\}$ into the cache and the row buffer. After another $L/2$ iterations, again a cache miss will occur for reading the value of B, C and writing A. But since $\{B, C\}$ will be read before writing A, and the row buffer contains $\{B, C\}$, so, there won't be an activate in this case. But we assume that here an activate will occur. So, we are basically overcounting the number of activates and precharges. But clearly these types of cases will be very few if the number of partitions in the interleaving is high since then the probability of the array being in the row buffer which had a cache miss will be quite low.
- We also assume that the input program does not have any multi-dimensional array.

3 The Algorithm

- We first define a function that takes as parameters a given loop and an interleaving of the arrays and returns the cost of executing the loop with this interleaving, in terms of the number of activates and precharges.

3.1 Function for cost of a loop given the interleaving

INPUT: A loop and an interleaving of the live arrays in the loop
OUTPUT: The cost of running the loop with the given interleaving in terms of number of activates and precharges.

NOTE: In the following the cost is calculated in terms of cost of 1 activate and precharge - $ACTV and PRE$. This can be replaced by the following:

$$ACTV = \alpha * T_{PR} + \beta * P_{PR} + \gamma * E_{PR}$$

$$PRE = \alpha * T_{PW} + \beta * P_{PW} + \gamma * E_{PW}$$

where T_{PR}, P_{PR}, E_{PR} = Read latency,power and energy for PCM
T_{PW}, P_{PW}, E_{PW} = Write latency,power and energy for PCM
and α, β and γ are input parameters.

Lets first consider the **base case**, where there are **no nested loops** and the coefficient of the induction variable in all array accesses is 1 (meaning there is no access like $A[ki], k \neq 1$

- Let the set of live arrays in the loop be given by $LIVE = \{A_1, A_2, A_3.., A_n\}$
  Also, let

  $$I = \text{Number of iterations of the loop}$$

  $$L = \text{Line size of the cache}$$

  $$R = \text{Row buffer size of the PCM interface}$$

- If $(\sum |A_i|\ \forall A_i \in LIVE) \leq R$ then interleave all the arrays and return. This is because if all the arrays can be fitted in the row buffer, then its best to interleave all of them and during the whole iteration this interleaved array will be present in the row buffer and there will be no misses from the row buffer.

- Let the interleaving provided to the function be given by $INTL = \{P_1, P_2..P_k\}$, where each P_i is a partition of the form $P_i = \{B_{i_1}, B_{i_2}..B_{i_m}\}$, where B_{i_p} is any array not necessarily live in the loop.

- Now we define the following 2 sets:

  $$P_R = \{P_i : P_i \in INTL \text{ and } \exists B_{i_j} \in P_i \text{ which is only being read in the loop and } \nexists B_{i_k} \in P_i \text{ which is being written in the loop.}\} \tag{1}$$

  $$P_W = \{P_i : P_i \in INTL \text{ and } \exists B_{i_j} \in P_i \text{ which is being written in the loop}\} \tag{2}$$

- Now define the costs due to these 2 sets:

  $$C_{P_R} = ACTV \times \sum \frac{x_i \times I}{L} \text{ where } x_i \text{ is the size of } P_i \in P_R \tag{3}$$

$$C_{P_W} = (2 \times ACTV + PRE) \times \sum \frac{x_i \times I}{L} \quad \text{where } x_i \text{ is the size of } P_i \in P_W \tag{4}$$

Since C_{P_R} is including the cost of all the arrays which are being only read and C_{P_W} is including the cost of all the arrays which are being written, so

$$\text{Total Cost of the loop} = C_{P_R} + C_{P_W}$$

- According to the assumptions made, there are no conflict misses in the cache. So, after the first iteration of the loop all the $P_i \in P_R$ will be present in the cache. The block of the cache corresponding to P_i will support $\frac{L}{x_i}$ iterations of the loop, meaning that till $\frac{L}{x_i}$ iterations of the loop, the data will be picked up from the cache. After every $\frac{L}{x_i}$ iterations of the loop, there will be a cache miss and so an activate. So, in I iterations of the loop, there will be $\frac{I}{L/x_i} = \frac{x_i \times I}{L}$ activates in total.
- Similarly, for $P_i \in P_W$ after every $\frac{L}{x_i}$ iterations of the loop, there will be an activate. So, there will be $\frac{x_i \times I}{L}$ activates during the execution of the loop. These dirty blocks of cache will be written back to memory whenever some other block conflicts with this. We make the assumption that one activate and one precharge is required to write one dirty block back to memory. Since one dirty block of cache will be put in after every $\frac{L}{x_i}$ iterations of the loop, number of dirty blocks= $n = \frac{x_i \times I}{L}$. Hence there will be $\frac{x_i \times I}{L}$ activates and precharges in addition to the ones already counted.

Now if there are **nested loops** in the given input loop:

- If the outermost loop runs for I_1 iterations and the immediately inner loop runs for I_2 iterations, then

$$\begin{aligned}\text{Total cost of the loop} = I_2 \times \text{Cost due to the just inner loop}\\ +\text{Cost due to the accesses to arrays only in the outermost loop}\end{aligned} \tag{5}$$

- Cost due to the just inner loop can be obtained from a recursive call to the same function.

In the general case where the coefficient of induction variable in the array accesses can be more than 1, we have the following:

- Consider the case there is an access of the form $A[2i]$ and the interleaving containing A has a size of 1, then the effective size of this interleaving becomes 2. This is because in the cache the elements accessed are at a spacing of 1, just like if A had been interleaved with some other array.
- Using the above idea, we can say that the effective size of the interleaving becomes coefficient times the actual size.
- Suppose the array access is $A[ki]$ and the interleaving containing A is $\{B_1, B_2..B_p\}$ then the effective size of the interleaving will become $p * k$.

3.2 Finding the cost of a node in the AIG

- A node represents a loop and an interleaving. So, to find the cost we just need to call the function defined above.

3.3 Finding the cost of an edge

- Let the edge be from node N_1 to N_2.
- Let $N_1 = \{P_{11}, P_{12}..P_{1r}\}$ and $N_2 = \{P_{21}, P_{22}..P_{2s}\}$ be the partitions corresponding to the 2 nodes.
- If N_1 and N_2 can be considered as sets then let D_1 and D_2 be as follows:

$$D_1 = N_1 \backslash \left(N_1 \bigcap N_2 \right)$$

$$D_2 = N_2 \backslash \left(N_1 \bigcap N_2 \right)$$

- So, now we basically need to find the cost of changing the interleaving from D_1 to D_2.
- Finding the optimal way of doing this is itself a difficult problem. So, we use a simple approximation and first de-interleave all the arrays and then interleave them. For example suppose $\{A, B\} \in D_1$ and $\{A, C\} \in D_2$. Then we first de-interleave the 2 arrays - A and B and then interleave A and C.
- The cost of de-interleaving the arrays is just a 'for' loop, which copies the 2 arrays into separate arrays. To find this cost, we use the above function described and find the cost of this 'for' loop.
- Similarly, the cost of interleaving is just another 'for' loop that copies the required arrays into another array. We use the function defined above to compute the cost of this 'for' loop.
- The total cost is then the cost of de-interleaving and interleaving.
- Note that since D_1 and D_2 donot include $N_1 \bigcap N_2$, we basically leave the interleavings that are the same in both the nodes. Example, if $\{A, B\} \in N_1$ and also $\in N_2$, then we basically leave this as it is.

3.4 Handling If-else statements

This subsection deals with the case where loops are present inside if-else statements.

- As described in the main algorithm later, we create a dummy node of cost 0 at the zeroth level. The edges coming out of this node has a constant cost of $C = constant$ each.
- Lets consider the case where there are no nested if-else and each of the if and else branches have one loop inside them. Then our algorithm works as follows:

- Find the probability of if and else branches being taken. This can be found using dynamic analysis and profiling. Basically, we use pintool and instrument the code so that a counter is incremented every time a if branch is taken. Similarly for else branch. This instrumentation across various runs of the input program should give the average probability of if/else branches being taken.
- Suppose L_1 and L_2 are the loops present in the if and else branches respectively. Also, let P be the probability that the if branch is taken and $1-P$ be the probability that the else branch is taken. Create all possible partitions corresponding to L_1 and L_2 and keep them both at the same level of our AIG. Define the node costs normally.
- Note that there will always exist a level above this if-else level since we also create a dummy node at the zeroth level.
- Now join the nodes in the above level to nodes corresponding to L_1. Find the edge cost normally - let this be e. Then the actual edge cost will be $\frac{e}{P}$.
- Similarly the edges joining the else nodes from the above level will have a weight of $\frac{e}{1-P}$.
- If there are other loops after the if-else branch, then their node and edge costs are found normally.

- Now generalizing this idea to the case where the branches are of the form

```
if{
    L1;L2 ..
}
else if {
    M1;M2; ..
}
else {
    N1;N2 ..
}
```

where $L1, L2..M1, M2..N1, N2..$ are loops.

  - Let the last level in the AIG before this if-else branch be n. Note $n \geq 0$.
  - Now in the $n+1$ level of the AIG, put nodes corresponding to $L1$, $M1$ and $N1$. Calculate the cost of these nodes normally.
  - The edge cost from some node in level n to some node of $L1$ will be $\frac{\text{Actual cost of edge}}{\text{Probability of execution reaching L1}}$
  - Similarly, for the nodes corresponding to $M1$ and $N1$.
  - Nodes corresponding to Li ,Mi ,$Ni, i \neq 1$ are put in the level below that corresponding to $L(i-1)$,$M(i-1)$ and $N(i-1)$ respectively.

- Now completely generalizing the idea to nested branches, we have the following:

– Suppose the input program has the following general form. This representation of the program covers the case where there is just a single if branch or if there is a sequence of if-else branches.

```
if{
    ...
}
else if {
    ....
}
else if {
    ....
}
.
.
.
else if {
    ....
}
```

– Now each of the branches may be further nested. Find the probability of entering into each of the branches. This can be done as already explained, by dynamic analysis - using pintool to increment a counter whenever a branch is taken.

– Now recursively solve and create an AIG for each of the branches.

– Note the 0th level of AIG has a dummy node. Remove it from each of the "sub" AIG's constructed for the branches.

– Now keep all these sub-AIG's side-by-side on the next level of the main parent AIG which was being constructed.

– Now the only task left is of connecting the nodes in the last level of the parent AIG to the first level nodes of each of the sub-AIGs. Find each of the edge costs normally and divide it by the probability of the branch being taken. This is the actual edge cost.

In this way, using a recursive call to the construction of AIG, we have handled the case where a general if-else nested branch occurs.

3.5 Search Space Pruning

Since the number of possible interleavings grows exponentially in the number of arrays under consideration, we need some strategies to prune out some high cost interleavings. For this we use the following strategies:

- The arrays being interleaved must have the same size and lifetime.
- The interleaved arrays must have the same reference pattern. For example, if we have $A[i]$ and $B[2*i]$ being read in the loop, then we can leave/prune out all possible interleavings that interleave both A and B. This is because if in such a case A and B were being interleaved as $\{A, B\}$, then A would not lead to a miss till L iterations, whereas B would lead to misses after $L/2$ iterations. This would lead to increase in the number of activates. So, its better not to interleave them at all.

- If one of 2 arrays in a loop is being read and other is being written, then we can leave all possible interleavings of the 2 arrays. Interleaving such arrays would lead to increase in the number of precharges.

3.6 Constructing AIG

- We analyse the functions one after another. For each function this algorithm finds the optimal interleaving by constructing the array interleaving graph (AIG).
- Consider the simpler case where the program just consists of a sequence of loops - $L_1, L_2 ... L_n$.
  - First we create a dummy node, the cost of which is zero and place it at the 0^{th} level of the AIG.
  - Any loop L_i consists of a set of live arrays - $A_{i1}, A_{i2}..A_{ik}$. We consider all possible interleavings of these live arrays and create a node for each such interleaving. These nodes will be placed in the i^{th} level of the AIG.
  - The cost of any node can be calculated as already described above.
  - Edges are added from each node at i^{th} level to each node at $(i+1)^{th}$ level. The edge cost represents the cost of changing the interleaving from one node to another and is computed as described in the previous subsection.
- Now consider the general case where the program consists of loops that can be nested or there can be nested if-else statements. Nested loops can be handled with no particular exception as the function described takes care of nested loops. For if-else statements, we can make a slight modification to the AIG as described in the previous subsection.

3.7 Finding the optimal path through AIG

- Once AIG has been constructed as described above, the problem reduces to finding the minimal cost path through a sequence of nodes and edges at each level of the graph.
- We use Dynamic Programming to find this minimal cost path as follows: Let $min\_cost(P_{i_j})$ represent the cost of the path of minimum weight from the dummy node in the 0^{th} level to P_{i_j}.

$$min\_cost(P_{i_j}) = \min_{all predecessors P_{(i-1)_k} of P_{i_j}} (min\_cost(P_{(i-1)_k}) + transition(P_{(i-1)_k} \rightarrow P_{i_j}))$$

  Base case :

$$min\_cost(P_{0_1}) = 0$$

  Using this DP recursive formulation, *min_cost* of all nodes can be computed levelwise.

  Then the final optimal answer = $\min(P_{m_j})$, where m is the total number of levels and j ranges over all the partitions at the m^{th} level.

3.8 Implementation

- We first take the input program, convert it to LLVM IR[.bc or .ll file format] and run the LLVM **indvars** pass on it.
  - This transformation transforms all the loops to have a single induction variable that starts at 0 and steps by one.
  - Also, all the pointer arithmetic recurrences are raised to use the array subscript. This will help in dealing some trivial cases of pointer arithmetic.
- Couple of other passes like the **mem2reg** , **loop-simplify** and **scalar-evolution** are run on the IR to further simplify the loop.
- Now a LLVM loop pass is written that works on the simplified IR, one loop at a time and constructs the AIG.
  - To do this, as it processes a loop, it finds the set of live arrays in the loop. Corresponding to these live arrays, it performs pruning on the set of possible partitions and finds the required set of partitions.
  - For each possible partition, it calls the function described earlier to find the cost of running the loop given the interleaving. It then adds a node with this cost to the AIG at the apt level.
  - After all the loops have been processed in this way, it adds edges between nodes computing the cost of the edges as described earlier.
  - Then the dynamic programming algorithm is run, that finds the minimum cost path through the AIG.
  - The nodes along this path then represent the optimal interleaving for the corresponding loops.
- To handle if-else conditions, we need to find the probability that a particular branch is taken. This can be found using pintool.
  - We write a pintool that instruments the input program to increment a counter whenever a particular branch is taken.
  - Finding the values of this counter across different runs of the input program enables us to find the probability of a particular branch being taken.

4 Analysis

- Once the AIG has been constructed, we are just finding the optimal path using the Dynamic Programming approach. So, the complexity of the algorithm is $O(E)$ where E is the total number of edges in the AIG.
- Although we are using pruning strategies to limit the number of interleavings, but in the worst case, the total number of edges in the AIG can be exponential in the number of arrays.
- The calculation of the number of activates and precharges is approximate due to the assumptions that there are no conflict and capacity misses.

- Inspite of the above demerits, since the AIG exhaustively searches for the best possible interleaving, we are sure to get an optimal answer.

5 Simple Example without any assumptions

Consider the loop :

```
for(int i=0 ; i<N ; i++){
    A[i] = 0;
    B[i] = 0;
}
```

Also suppose that there is one level of cache - L1 and PCM main memory. The cache has a size of 16 elements and each line/block of cache has 4 elements and the cache is direct mapped cache. Suppose the row page buffer has a size of 16 elements. 1 element is considered to be an integer.

We know if some element has an address of $addr$, then its mapping in the cache is given by:

$$(\frac{addr}{4})mod4$$

.

Consider the case when A and B are not interleaved. The address of $A[0]$ is 0 and $B[0]$ is 1000. So, the mapping of $A[0]$ is in line 0 in the cache and that of $B[0]$ is in 2.

- Initially there is a compulsory miss when $A[0]$ is accessed. This results in the page buffer having elements of A only. Accessing B will now result in a page miss. So, for the first iteration, there have been 2 ACTIVATE commands.
- The cache line 0 now has elements of $A[0] - A[3]$ and cache line 3 has elements $B[0] - B[3]$.
- For $i = 0$ to $i = 3$ there will be cache hits.
- At $i = 4$, $A[4]$ and $B[4]$ are required, which are not present in the cache. So, a page miss happens and there are 2 ACTIVATE commands issued to PCM.
- $A[4] - A[7]$ is mapped to cache line 1 and $B[4] - B[7]$ is mapped to cache line 3.
- From $i = 4$ to $i = 7$ there will be cache hits and so upto $i = 7$ a total of 4 ACTIVATE commands have been issued.
- From now on, for every 4 iterations there will be one PRECHARGE and one ACTIVATE command for each A and B.
- So, the total cost of this loop is

$$4A + \frac{N-8}{4} * 2 * (P + A)$$

Now suppose that A and B are interleaved and address of $A[0]$ is 0.

- Initially, there will be a compulsory cache miss and $A[0]$, $B[0]$ ... $A[7]$, $B[7]$ will be fetched into the row buffer using one ACTIVATE command.
- For $i = 0$ to $i = 7$ the cache misses (if any) will not lead to any page miss since $A[0]$, $B[0]$ ... $A[7]$, $B[7]$ are present in the row buffer.
- From $i = 8$ onwards, for every 8 iterations there will be one PRECHARGE and one ACTIVATE command issued to PCM.
- So, the total cost is
$$1A + \frac{N-8}{8} * (P + A)$$
- This is definitely lower than the case where there are no interleavings.

So, we analysed an example where interleaving lead to reduction in the total cost.

6 Experimental Validation

1. Consider the following simple loop:

   ```
   for(int i=0;i<1000;i++){
       A[i] = B[2i]+C[i];
   }
   ```

   - According to the proposed algorithm, after pruning the only possible interleaving is $\{\{A\},\{B\},\{C\}\}$.
   - Verifying using the gem5 simulator provided, we have:
     - $\{\{A\},\{B\},\{C\}\}$: Number of precharges = 13
     - $\{\{A\},\{B,C\}\}$: Number of precharges = 41
     - $\{\{A,B,C\}\}$: Number of precharges = 68
   - Thus, as found using the simulator interleaving $\{\{A\},\{B\},\{C\}\}$ is best among all possible interleavings.

2. Consider this program:

   ```
   for(int i=0;i<1000;i++){
       A[2i]=B[3i]+C[i];
   }
   for(int i=0;i<1000;i++){
       B[i] = C[3i]*A[2i];
   }
   ```

   - According to the proposed algorithm, after pruning the best possible interleaving is $\{\{A\},\{B\},\{C\}\}$. This is because the reference pattern of $\{\{A\},\{B\},\{C\}\}$ is different.
   - Verifying this on the gem5 simulator, we have
     - $\{\{A\},\{B\},\{C\}\}$ throughout: Number of precharges = 77
     - $\{\{A,B,C\}$ throughout: Number of precharges = 299

    - $\{\{A\},\{B\},\{C\}\}$ for first loop and $\{\{A,C\},\{B\}\}$ for the second loop: Number of precharges = 249

- We calculated the precharges for other possible interleavings on the simulator and found that the interleaving $\{\{A\},\{B\},\{C\}\}$ throughout the 2 loops has the least number of precharges, which agrees with the result of our algorithm.
- Manually we calculated the cost of all possible interleavings for both the loops and the transition between the interleavings and we found that it matched with the results given by pruning and the simulator.

3. Consider an example of a nested loop:

```
for(int i=0;i<1000;i++){
    A[i] = B[i];
    for(int j=0;j<1000;j++){
        C[i] = A[i] + D[i];
    }
}
```

    - According to our algorithm the **scaled** number of activates and precharges for each possible partition are:
        - $\boldsymbol{\{\{A\},\{B\},\{C\},\{D\}\} : 4A + 1P}$
        - $\{\{A,B\},\{C,D\}\} : 6A + 2P$
        - $\{\{A,C\},\{B,D\}\} : 6A + 2P$
        - $\{\{A,D\},\{B,C\}\} : 6A + 2P$
        - $\{\{A,B\},\{C\},\{D\}\} : 5A + 1P$
        - $\{\{A,C\},\{B\},\{D\}\} : 5A + 2P$
        - $\{\{A,D\},\{B\},\{C\}\} : 4A + 1P$
        - $\{\{B,C\},\{A\},\{D\}\} : 6A + 2P$
        - $\{\{B,D\},\{A\},\{C\}\} : 5A + 1P$
        - $\{\{C,D\},\{A\},\{B\}\} : 5A + 2P$
        - $\{\{A,B,C\},\{D\}\} : 7A + 3P$
        - $\{\{A,C,D\},\{B\}\} : 6A + 3P$
        - $\{\{A,B,D\},\{C\}\} : 5A + 1P$
        - $\{\{B,C,D\},\{A\}\} : 7A + 3P$
        - $\{\{A,B,C,D\}\} : 8A + 4P$
    - Verifying this on the simulator, we found the actual number of activates and precharges as:
        - $\{\{A,D\},\{B\},\{C\}\} : 10PRE + 493ACT$
        - $\{\{A,B,C,D\}\} : 22PRE + 352ACT$
        - $\boldsymbol{\{\{A\},\{B\},\{C\},\{D\}\} : 9PRE + 442ACT}$
        - $\{\{A,B,C\},\{D\}\} : 23PRE + 485ACT$
    - The highlighted interleavings are the ones having minimum cost in either case, and they happen to be the same. Also, for other interleavings the relative cost calculated using the algorithm and the simulator matches.