Optimal cache-aware layouting of random forests with formal LRU cache modeling

Undergraduate Thesis

Submitted in partial fulfillment of the requirements of
BITS F421T Thesis

By

Saurav Shukla
ID No. 2018B5AA0653G

Under the supervision of:

Dr. Jian-Jia Chen
&
Dr. K.R. Anupama

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, GOA CAMPUS
December 2022

Declaration of Authorship

I, Saurav Shukla, declare that this Undergraduate Thesis titled, 'Optimal cache-aware layouting of random forests with formal LRU cache modeling' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: Saurav Shukla

Date: 22/12/2022

"The cave you fear to enter holds the treasure you seek"

Joseph Campbell

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, GOA CAMPUS

Abstract

Masters of Science, Bachelor of Engineering (Hons.)

Optimal cache-aware layouting of random forests with formal LRU cache modeling

by Saurav Shukla

Optimisation of computing resources is an active field of study in today's times. The following study discusses an Integer Linear Programming approach to produce an optimal cache-aware layout for random forests with LRU cache modeling. The results produced are global optimums unlike the results produced by machine learning approaches. The work builds on a pre-existing decision tree generator and analyses the cache performance for varying decision tree depths. The ILP approach produces a high hit ratio for varying decision tree depths and analysis is done to discuss the effect of the input parameters on the resulting cache-hit count. Finally, suggestions have been made to possibly further increase the hit ratio and build upon the presented work.

Acknowledgements

I would like to express my sincere gratitude to Prof. Jian-Jia Chen for hosting me within the Design Automation For Embedded Systems Group (DAES), TU Dortmund University as a Bachelors Thesis student. I really appreciate his patience, insight, and advice.

My sincere thanks also goes to my co-supervisor, Prof. K.R. Anupama, whose support and flexibility made this undertaking possible.

My sincere thanks also goes to Christian Hakert and Nils Hölscher for helping me throughout the thesis.

Contents

List of Figures

I dedicate this to my family who have always believed in me.

Chapter 1

Introduction

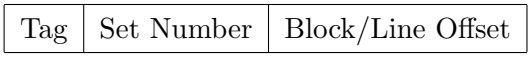
Over the last few decades, computer systems have seen significant development, partly due to improvements in different levels of cache memory. Cache memory is a smaller but faster form of memory that stores only the most frequently accessed instructions and data.

Whenever the processor needs to access data, it first checks if it is available in the cache memory. If the data is available in the cache memory, then main memory access is not required. This is known as a "cache-hit". If the data is unavailable in the cache memory, then the main memory will have to be accessed. This is known as a "cache-miss". Since cache memory access is several times quicker than main memory access, a high cache-hits to cache-miss ratio is desired for better performance.

Two major factors affecting cache performance are cache mapping policy and the cache replacement strategy. A cache mapping policy is mainly of three kinds: directly mapped cache, fully associative mapping and n-way set associative mapping. In this study, the cache memory under consideration is an n-way set associative cache.

As the name implies, a block is a contiguous "block" of memory of a particular size. In a n-way set associative cache memory, each block is assigned to a particular "cache-set" but within a particular cache-set the memory block is free to occupy any given cache-line. Hence, the cache set that a particular memory block is assigned to plays a massive role in cache performance because if blocks are assigned to cache sets in a manner in which they remain in the cache memory for long, the processor performance will be better.

The physical memory division for a set associative cache is done in the following manner:



Hence depending on the physical memory of the information, it is assigned to a particular set and a give block. If the set number and the block/line offset is given, then a physical memory location can be assigned to the data for an optimal number of cache hits.

In this thesis, the aim is that given a decision tree, the tree nodes need to be assigned to the memory in a manner that is optimized to give a high hits:miss ratio.

To carry out the above optimization, an Integer Linear Programming (ILP) formulation is utilized, wherein given the decision tree and details of the computer architecture, the ILP outputs the position of each memory object within the cache memory.

Finally, using this information from the ILP, a C++ program is written where the memory objects are placed in positions such that they acquire memory addresses according to which they will be placed in locations dictated by the ILP output.

Chapter 2

Problem Statement

The problem statement for the presented study is as follows: Given a decision tree, implement a cache-aware layout for optimal cache-hits for the case of LRU replacement strategy.

The given quantities for the problem are:

(i) Decision Tree information

(ii) Information about the system architecture

Information about the decision tree includes the probability for taking any decision and the left and right child of each node.

Information about the system architecture includes specifications for the system cache, associativity of the cache, cache-line size.

The following chapters will present a methodology to create an ILP formulation and further implement it on an actual LRU cache system.

Chapter 3

Decision Tree Representation

The study builds on the concept of decision trees and the work discussed in [1]. The code generator developed in [1] is used to construct decision trees of varying depths.

Consider the following decision tree:

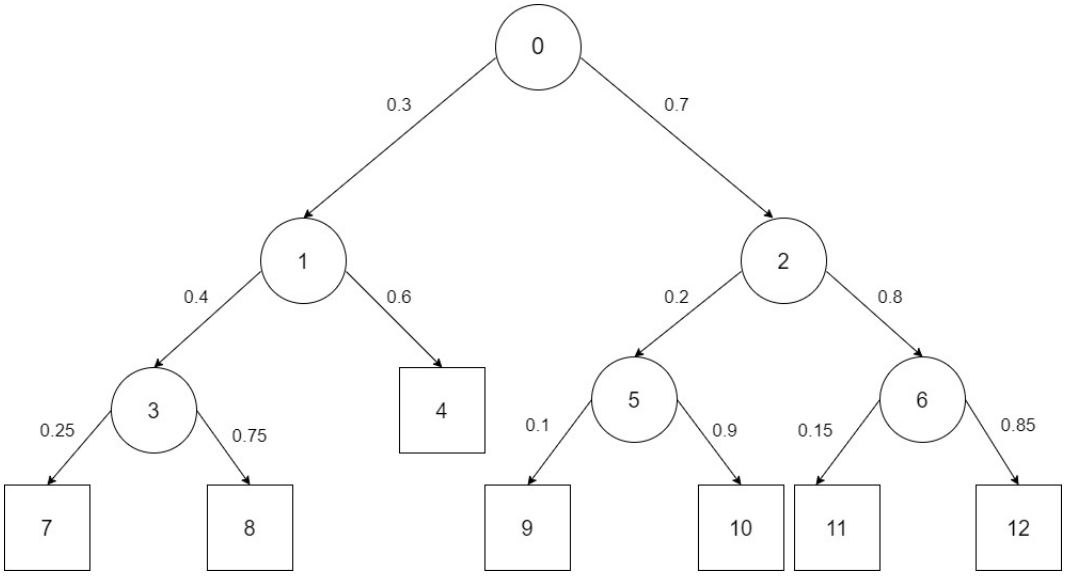


FIGURE 3.1: Decision Tree of Depth 4

The above figure showcases an example of a decision tree of depth 4. Inner nodes are depicted with circles, leaf nodes are displayed as rectangles. Each node has a unique id and every path is associated with a probability. For example, the probability to go from the root node 0 to its right child 2 is $p(0 \rightarrow 2) = 0.7$.

The first step of the study is the conversion of the decision tree into a format that is suitable for an ILP formulation. For this purpose, each node of the decision tree under consideration is represented by an integer. Hence this becomes a tree where each node is represented by

an integer and each edge represents the probability of making a particular decision. For the representation of this tree, random walks along the tree of length "l" are taken. Here a random variable "threshold" is chosen from a uniform distribution and the decision to visit a particular node in a tree is taken depending on whether the probability of visiting that node is greater than or less than the value of the threshold random variable. In a random walk, if a leaf is encountered then the path again reverts back to the root and again a random path is taken. This process is continued until the length of the access sequence is equal to the specified access sequence length "l". The visited nodes will form a sequence of integers or an "access sequence". The longer the access sequence the better it represents the decision tree. The generated access sequence is then fed to the ILP which uses this and the architecture details of the system to output an optimal cache mapping for the memory objects.

Chapter 4

Cache Behaviour Prediction

4.1 Cache Specifications

A cache memory can be specified using the following architecture details:

(i) **Capacity** - The number of bytes a cache may contain

(ii) **Line Size** - The number of contiguous bytes that are transferred from memory on a cache miss. The cache can hold at most (capacity)/(line size) blocks.

(iii) **Set Count** - The number of sets in a given cache memory

If a block can reside in any location within a cache memory then it is called a fully associative cache and if it can reside in k locations within a set then it is a k-way associative cache.

4.2 Replacement Policy

In general, in a cache memory a cache line may be replaced when the cache memory is full and the processor requests further data. Various replacement algorithms such Least Recently Used (LRU), Least Frequenty Used (LFU), First In First Out (FIFO) may be used to select the cache line that is to be replaced. In this study, all the analysis is done assuming a Least Recently Used (LRU) cache replacement policy.

In an LRU replacement policy, as the name suggests, the least recently used block is replaced in the cache memory. For the implementation of this strategy, the algorithms needs to keep a track of what was used when.

4.3 Formalisation of LRU replacement policy

4.3.1 Cache Behaviour in Fully Associative Cache Memeory

As discussed in the previous section, the LRU algorithm implementation needs to keep a track of what block was used when. The formalisation for this strategy is done based on discussion carried out in [3].

For keeping a track of how recently a block was accessed, we define the quantity 'age' for each block. The reason for using 'age' is that initially each block should be 'old' or unused. As these memory blocks keep getting accessed they become 'younger'. When a block is not being accessed, then it should keep getting 'older'. The 'oldest' block or the block with the maximum age present in the cache memory should be replaced. The "age" values are modified using the following operations:

- **"refreshing"**: The age of an accessed block is "refreshed". Arithmetically, this involves setting the age of the accessed block to 0.
- **"aging"**: Every other block apart from the accessed block undergoes "aging". This simply means that for all other blocks, the age is incremented by 1 from their previous ages.

Consider a fully-associative cache with cache size 32 bytes, let the line size be 8 bytes. This implies that the cache has 4 cache-lines. By definition, a memory block takes up the space of 1 cache line. Hence, the cache memory can hold 4 memory blocks at once. Let the blocks under consideration be labelled 'a', 'b', 'c', 'd', 'e', 'f', 'g'. Initially, we assume that none of blocks are present in the cache memory. Let the access sequence be {b,c,b,e,f,c}. Initially, all blocks should be 'old' and the blocks which keep getting used or accessed should be younger. So initially the ages of all blocks should be initialised to a very high age. In this case, we assign the age to be 8. This is because at every access the age of a block is incremented by 1 or it becomes zero. Hence, the age of any block cannot exceed 8 since there 6 accesses in the access sequence.

The rule for categorizing a memory access as a hit or a miss is that the accessed block should be among the 'n' youngest blocks where 'n' is the number of blocks that the cache can hold at once. In this case, n = 4.

As stated, the access sequence is: {b,c,b,e,f,c}.

Access 1: For the first access the age are as follows:

Memory Block	a	b	c	d	e	f	g
Age	8	8	8	8	8	8	8

Accessed Block: b

Since b is not among the 4 youngest blocks according to the above ages, hence this access is a **cache-miss**.

Access 2: For the second access the age are as follows:

Memory Block	a	b	c	d	e	f	g
Age	9	0	9	9	9	9	9

As visible above, the age of block 'b' has been "refreshed" or set to 0. All the other blocks have undergone "aging" or their ages have been incremented by 1.

Accessed Block: c

Since c is not among the 4 youngest blocks according to the above ages, hence this access is a **cache-miss**.

Access 3: For the third access the age are as follows:

Memory Block	a	b	c	d	e	f	g
Age	10	1	0	10	10	10	10

The age of block 'c' has been set to 0 and the ages of all blocks have been incremented by 1.

Accessed Block: b

Since b is among the 4 youngest blocks, this access is a **cache-hit**.

Access 4: For the third access the age are as follows:

Memory Block	a	b	c	d	e	f	g
Age	11	0	1	11	11	11	11

The block 'b' has undergone "refreshing" while all other blocks have undergone "aging".

Accessed Block: e

Since e is not among the 4 youngest blocks, this access is a **cache-miss**.

Access 5: For the third access the age are as follows:

Memory Block	a	b	c	d	e	f	g
Age	12	1	2	12	0	12	12

The block 'e' has undergone "refreshing" while all other blocks have undergone "aging".

Accessed Block: f

Since f is not among the 4 youngest blocks, this access is a **cache-miss**.

Access 6: For the third access the age are as follows:

Memory Block	a	b	c	d	e	f	g
Age	13	2	3	13	1	0	13

The block 'f' has undergone "refreshing" while all other blocks have undergone "aging".

Accessed Block: f

Since c is the 4th youngest block, this access is a **cache-hit**.

Finally, after all the accesses the ages become:

Memory Block	a	b	c	d	e	f	g
Age	14	3	0	14	2	1	14

From the above discussion, for fully-associative cache memory, cache-hits and cache-misses can be categorized using the concept of 'ages' and the 'refreshing' and 'aging' of blocks

4.3.2 Cache Behaviour in Set Associative Cache Memeory

The discussion done in the last section for a fully associative cache memory can be extended to a k-way set associative cache memory.

A set associative cache with 's' sets can be thought of as 's' individual fully associative cache memories for implementation purposes. This means that the ages of blocks needs to be maintained for each cache-set separately and for every block will have some age in every set. For classifying a memory access as a cache-hit or a cache-miss, the accessed block should be among the 'n' youngest in the cache-set that it is mapped to. Here 'n' refers to the maximum number of memory blocks that a cache-set can hold at any given time.

For the updation of ages, the "refreshing" and "aging" also needs to be carried out in sets individually. For example, if block 'c' is mapped to cache-set 2 and if it is accessed then the age of block 'c' will be refreshed in set 2 but for all other sets the block 'c' should undergo aging since it is not present in those cache-sets. For all other memory blocks, aging should take place in every cache-set.

Chapter 5

Principles of Integer Linear Programming

5.1 Introduction

Integer Linear Programming is a technique for optimising an objective function subject to a set of linear equalities and inequalities. An ILP formulation has three major parts:

(i) Declaration of variables and allowed values

(ii) Formulating constraints for the declared variables

(iii) Stating the objective function

5.2 Variable declaration

When taking the ILP approach for solving problems, the declared variables can take mixed integer, integer or binary values. In an integer case, all variables need to necessarily be integers. In a mixed integer approach, only some variables need to be integers while other variables can assume any numerical values. In the case of binary values all variables need to assume binary values.

5.3 Constraint Equations

Constraint equations are where details about the problem are fed into the ILP formulation. The declared variables are subject to certain constraints which may vary for different problems. Apart

from the specifications of the problem, constraints are also used to specify logical constraints. For example, if a problem deals with absolute quantities such as lengths or weights then a constraint specifying that the variables representing these quantities should always be positive can be added.

5.3.1 Constraint Equation Types

Apart from linear constraints modern ILP solvers allow variables to function under quadratic, polynomial, logarithmic, exponential and trigonometic constraints. For example, y = tan(x) is a valid constraint for ILP solvers like Gurobi.

To make the modelling of real life problems easier, constraints could also be piecewise in nature where the behaviour of a variable depends on the range of input values.

A constraint equation can depend on other variables too. For example, the value of a binary variable may depend on whether another variable is positive or negative. This can be used for implementing standard mathematical functions like the sgn(x) function.

5.3.2 Helper Variables

Apart from being used for modelling problem specifications, constraint equations can also be used in defining "helper variables". Since in an ILP, every variable is defined using a set of constraint equations, hence, random access to the values of these variables is not possible in many cases. In such instances, helper variables can be used whose constraints equations involve a set of equations that finally retrieve the value of a required variable.

For example, consider the binary array with only one non-zero value: A = {0,0,0,0,0,1,0,0,0}. In a sequential program, conditional statements can be used to retrieve index where the value in the array is non-zero. In an ILP implementation, a helper variable with the following constraint may be used to achieve a similar result:

$$h = \sum_{i=0}^{i=8} A_i \tag{5.1}$$

In this manner, the helper variable 'h' will assume value of the index containing a non-zero value.

5.4 Objective Function

Finally, after all the constraints have been specified, an objective function needs to be stated which can be optimised by the ILP to retrieve a solution. This optimisation maybe in order to maximise or minimise the objective function.

5.5 A sample ILP Formulation

Consider, the popular knapsack problem which has a simple dynamic programming solution. An optimal solution can also be achieved using an ILP formulation.

5.5.1 Problem Statement

Given a set of objects, each object has a value $c_i > 0$ and a weight $w_i > 0$. A knapsack with weight capacity b has to be filled such that the total value of the objects inside the knapsack is maximized. Assume weight and cost values to be integers.

5.5.2 Variable declaration

Apart from c_i and w_i a the ILP formulation needs a set of binary variables x_i , where $x_i = 1$ represents that the item i must be included in the knapsack.

5.5.3 Constraint Equations

The constraint equation for the above problem is:

$$\sum w_i x_i <= b \ \forall_{i=\{1,2....n\}} \ x_i \in \{0,1\} \tag{5.2}$$

5.5.4 Objective Function

The objective function for maximising cost of items in the knapsack can be stated as:

$$\max \sum_{i=0}^{i=n} c_i x_i \ \forall_{i=\{1,2....n\}} \ x_i \in \{0,1\} \tag{5.3}$$

Hence, in this manner an ILP formulation can be used to get optimised solution to various kinds of problems.

Chapter 6

ILP Formulation for LRU Cache

6.1 Given Quantities

For an ILP formulation, the following quantities are given:

Quantity	**Description**
Access Sequence (z)	A sequence of integers where each integer represents a memory access
Access Sequence Length (l)	Length of access sequence
Set Count (s)	Number of cache-sets in the cache memory of the given system
Lines per cache set (c)	Number of cache lines per cache-set
Line Size (n)	Number of memory objects that can fit within a given cache-line
Memory Object Count (m)	Total number of memory objects in the given decision tree

6.2 Constraint Equations

6.2.1 Memory Mapping Constraints

The first task of the ILP is to return a mapping for each memory object. This mapping should include the set and the cache line to which a particular memory object should belong to. For this, we create a 3D binary search space 'X' where the first dimension specifies the cache-set, and the second dimension specifies which memory object is under consideration. The third dimension specifies the cache-line to which the memory object belongs.

For example, if X[2,4,6] = 1, then this implies that the memory object labeled '4' belongs to cache-set labeled '2' and cache-line labeled '6'. The dimensions of 'X' should be (s x m x m). Here, s represents the set-count of the cache memory, and m represents the total number of

unique memory objects in the decision tree. The first dimension has size 's' because it represents the cache-set to which the memory object belongs. In total, there are 's' cache-sets. The second dimension has the size 'm' because it represents the memory object being considered. In total, there are 'm' unique memory objects. The third dimension represents the cache line where the memory object is present. This dimension is of size 'm' because in the extreme case where each memory object is not grouped with any other memory object in a cache line, there will be a total of 'm' cache lines required.

To represent the above situations in an ILP, constraint equations need to be formulated. The constraint equations for 'X' are as follows:

1. Each memory object should be present in exactly one cache-set and even in that cache-set only in one cache line.

$$\sum_{i=0}^{\text{s-1}} \sum_{k=0}^{\text{m-1}} X_{i,j,k} = 1 \ \forall \ j \in [0, \text{m-1}] \tag{6.1}$$

2. A cache-line cannot have more than 'n' memory objects.

$$\sum_{j=0}^{m-1} X_{i,j,k} <= n \ \forall \ i \in [0, s-1], k \in [0, m-1] \tag{6.2}$$

6.2.2 Age Calculation

After receiving a valid set of values for the 'X' binary variable space, each memory access needs to be categorized as a cache-hit or a cache-miss. Finally, the ILP can be optimized for maximizing the total number of cache-hits.

Since the replacement algorithm is the Least Recently Used algorithm, hence, for categorizing each memory access as a cache-hit or a cache-miss we define the "age" of a memory object.

We define a 3-D variable space 'A' for maintaining the ages. This will keep track of how recently a particular memory object was accessed. Instead of storing the age for each memory object, we store the age for each cache group i.e, each group of objects that are present in a given cache-line. This is because these objects are added and removed from the cache-memory simultaneously. Hence, all of them will have a common age.

The 3D variable space 'A' dimensions are (s x m x (l+1)). The first dimension represents that we maintain the age of cache line groups separately for each set. The second dimension is the maximum possible number of cache-line groups. The third dimension represents that we initialize the age for each cache-line group and then update it for each of the 'l' memory accesses.

The age of a cache-line group is calculated as follows:

1. Initially, the age of each memory object should be infinite. Instead of infinite, we initialize it to (l+2), which is an unachievable age for any memory object.

$$A_{i,j,k} = (l+2)\ \forall\ i \in [0, s-1], \forall j \in [0, m-1], \forall k \in \{0\} \tag{6.3}$$

2. The age at each subsequent memory access should be updated as follows:

(i): If cache-line group 'j' has not been accessed at access point 'k':

$$A_{i,j,k} = (A_{i,j,k-1} + 1)\,,\ \forall\ i \in [0, s-1],\ \forall\ j \in [0, m-1]\,,\ \forall\ k \in [1, l] \tag{6.4}$$

(ii): If cache-line group 'j' has been accessed at access point 'k':

$$A_{i,j,k} = 0\ ,\forall\ i \in [0, s-1]\ ,\forall\ j \in [0, m-1]\,,\ ,\forall\ k \in [1, l] \tag{6.5}$$

To represent the above two conditions using a single equation we take the help of 'helper variables'. We define helper variables 'H' of dimensions same as 'A'. $H_{i,j,k}$ should be 1 whenever cache-group 'j' is accessed at access point 'k' in set 'i'. Otherwise it should be 0 in all other cases. This can be created as follows:

$$H_{i,j,k} = X_{i,z_k,j} \forall i \in [0, s-1], \forall j \in [0, m-1], \forall k \in [0, l-1] \tag{6.6}$$

Now to obtain the age at any access point, following operation can be implemented:

$$A_{i,j,k} = (1 - H_{\ i,j,k-1}) * (A_{i,j,k-1} + 1)\ \forall i \in [0, s-1] \forall j \in [0, m-1], \forall k \in [1, l] \tag{6.7}$$

6.2.3 Total cache-hit count calculation

The rule for classifying memory access as a cache hit is that the "age of the accessed cache-line group should be among the 'c' youngest in the cache-set to which it belongs." To apply this rule we first need the age of the accessed object at each memory access. For this, we utilize another helper variable, 'O'. 'O' will be a sequence of integers of length 'l' where each integer will represent the age of the accessed cache-line group at the previous access point. This can formally be represented as:

$$O_k = \sum_{j=0}^{m-1} \sum_{i=0}^{s-1} H_{i,j,k} * A_{i,j,k}\ , \forall k \in [0, l-1] \tag{6.8}$$

To find whether the age of the accessed object is among the 'c' youngest in the cache-set or not, we take the help of another set of helper variables. These helper variables will assume binary values depending on whether the age of the cache-line group at an access point is greater than or less than the age of the accessed cache-line group.

Hence, we define helper variables 'D' in the following manner:

$$D_{i,j,k} = (A_{i,j,k} <= O_k - 1) \ \ ,\forall\ i \in [0, s-1],\ \forall\ j \in [0, m-1]\ ,\forall\ k \in [0, l-1] \tag{6.9}$$

The next step in this process involves counting the number of objects younger than the accessed memory object. This can be done by taking a sum across the second dimension of the 'D' helper variables, which in 3D variable space of dimensions (s x m x l). These values are then stored in another set of helper variables, 'Q'. The constraint equations for these variables are:

$$Q_{i,k} = \sum_{j=0}^{m-1} D_{i,j,k},\ \forall\ i \in [0, s-1],\ \forall\ k \in [0, l-1] \tag{6.10}$$

Finally these helper variable values are compared to 'c' and depending on whether they are greater than or less than 'c', the access is categorized as a cache-hit or a cache-miss for that particular cache-set.

Hence, to calculate total cache-hits for each cache set, we define a 2D variable space 'B', which is of dimensions (s x l). The first dimension is of size 's' because it needs to store the cache-hit, cache-miss data about each cache-set. The second dimension is 'l' because, for each set, every access needs to be categorized as a cache-hit or a cache-miss. The values for 'B' are defined as follows:

$$B_{i,j} = (Q_{i,j} <= c - 1)\ \forall\ i \in [0, s-1],\ \forall\ j \in [0, l-1] \tag{6.11}$$

Finally, from this 2D variable space, we need to consider the cache-miss or cache-hits for only the sets that have the accessed block. For all other sets, that memory access should be considered a cache-miss. The reason for doing this is that using this method, '1' will be used to signify a cache hit, while for other cache-sets which do not contain the accessed block, the value will be '0'. We can sum all values from this to get the total number of cache hits. Hence, now we define a new variable space, 'F', which will store the final cache-hit and cache-miss data for processing. In 'B', when a block that was accessed was not in the set being considered, it was not made sure that this was a cache-miss. To ensure this, we will multiply all such values with 0 to convert them to a cache miss. The resulting values will be stored in the variable space 'F'. This can formally be expressed as:

$$F_{i,j} = (\sum_{k=0}^{m-1} H_{i,k,j}) * (B_{i,j}) \ \forall \ i \in [0, s-1], \ \forall \ j \in [0, l-1] \tag{6.12}$$

6.3 Objective Function

Finally, we need to optimize the cache-layout to maximize the total number of cache-hits. 'F' contains the cache-hit and cache-miss data for each cache set. The total number of cache hits will be given by simply summing over all values of this 2D variable space. Hence, the objective function for the above ILP is given as follows:

$$Maximize : \sum_{i=0}^{s-1} \sum_{j=0}^{l-1} F_{i,j} \tag{6.13}$$

Chapter 7

Implementation of ILP Output

After receiving the an optimal cache-layout from the ILP, a C++ program can be written which can be run on an actual set-associative LRU cache based system. From the ILP we get the block to which each memory object will belong to. From this we need to map each node of the decision tree to a physical memory location such that the set and block of the node matches with the output returned by the ILP.

For a C++ implementation of the decision tree, we represent each tree node as a structure which contains information about the left child and right child of the node as well as the prediction probabilities. Thus, the entire tree will simply be an array of these structures. The position of each node will be decided by the ILP.

Consider the following representation of the cache memory:

Cache-Set 1	**Cache-Set 2**	**Cache-Set 3**	**Cache-Set 4**
Location #1	Location #2	Location #3	Location #4
Location #5	Location #6	Location #7	Location #8
Location #9	Location #10	Location #11	Location #12
Location #13	Location #14	Location #15	Location #16

If the starting memory location of the array is mapped to say Location #1, and the size of each structure is half the size of Cache-Set 1 then the next array item will also be mapped to Cache-Set 1 and the next 2 array items will be mapped to Cache-Set 2 and so on. Using this, we can store each array item to a position such that it is mapped to the required Cache-Set.

Chapter 8

Results

8.1 Variation of ILP results with input parameters

In the first stage, the decision tree nodes are used to create a sequence of memory objects by taking random walks in the tree. We study the effect of different parameters on the ILP output by defining a quantity called the hit ratio. Hit ratio = (Cache-Hits)/(Cache-Hits + Cache-Misses).

8.1.1 Effect of decision tree depth on hit ratio

When the depth of the decision tree is varied while keeping the length of the generated access sequence constant, the following hit ratio pattern is observed:

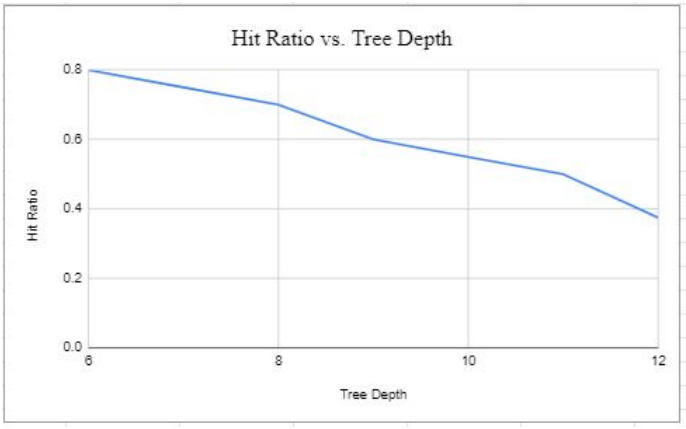


FIGURE 8.1: Hit Ratio vs Decision Tree Depth Plot

In the above graph, it can be observed that the hit ratio decreases with an increase in tree depth. This is because the number of memory objects increases exponentially with an increase in the

tree depth. In the case of small tree depths, certain high-probability nodes start appearing together, and the ILP groups them together, thus producing a high hit ratio. When this depth is increased, no such repetition occurs; hence, the hit ratio declines.

8.1.2 Effect of access sequence length on hit ratio

In the graph above, one factor that needs to be considered is the length of the access sequence. If the access sequence is too low, the memory access will always be a cache-hit irrespective of the decision tree depth. For example, in the previous case for the same architecture, if the access sequence length is four, the hit ratio will be 1 for all the mentioned tree depths. The access sequence should be a few times larger than the number of blocks the cache memory can accommodate at a given time. Otherwise, all blocks will always remain in the cache memory, and the mapping will be sub-optimal.

Consider the following graph displaying the variation of hit ratio with a change in access sequence size:

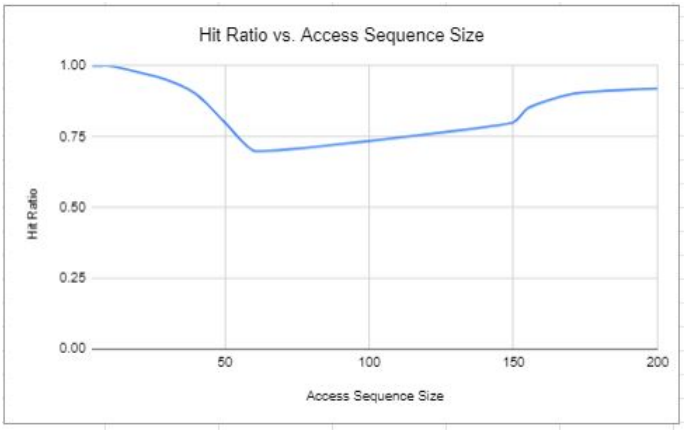


FIGURE 8.2: Hit Ratio vs. Access Sequence Size Plot

In the figure above, the hit ratio is initially 1 because when the access sequence is too low, all the accessed memory objects always stay in the cache memory. Upon increasing this length, the hit ratio decreases because now the number of unique memory objects being accessed starts increasing, due to which the cache memory is unable to hold all memory objects, thus increasing the number of cache misses. After a point, again the cache-hits start increasing because certain high-probability paths in the decision tree start repeating themselves in the access sequence, and the objects, when grouped together in a block, stay for longer in the cache-memory. This leads to an increase in the hit ratio.

The access sequence length cannot be too long either because the calculation of age for each memory block which is the most computationally heavy part of the ILP, is of complexity (s x

m x l). Here l is the length of the access sequence. If the access sequence length is not chosen optimally, then the model becomes impossible to test in realistic environments.

Following is a plot for the variation of the total number of variables in the ILP formulation with the change in the access sequence length:

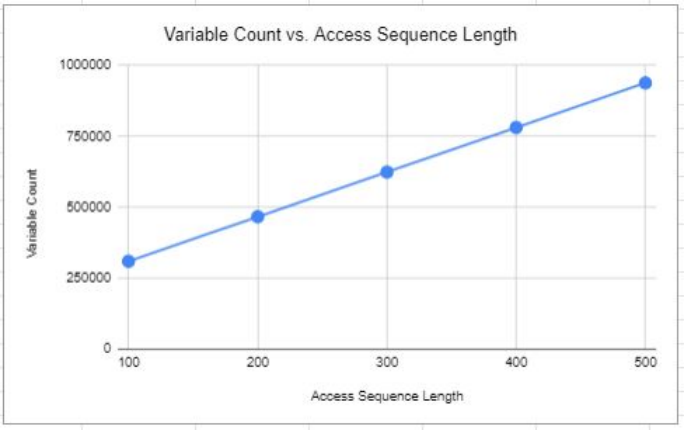


FIGURE 8.3: Variable Count vs. Access Sequence Length Plot

As expected from the previous analysis the total number of variables scales exactly linearly with the increase in access sequence length.

8.2 Issues with implementation on a real LRU cache system

Consider the specifications of a real LRU cache system:

Specification	**Size**
Level-1 Instruction Cache Size (Bytes)	32768
Level-1 Instruction Cache Associativity	8
Level-1 Instruction Cache Linesize (Bytes)	64
Level-1 Data Cache Size (Bytes)	32768
Level-1 Data Cache Associativity	8
Level-1 Data Cache Linesize (Bytes)	64
Level-2 Cache Size	524288
Level-2 Cache Associativity	8
Level-2 Cache Linesize (Bytes)	64
Level-3 Cache Size	268435456
Level-3 Cache Associativity	0
Level-3 Cache Linesize (Bytes)	64

From the above table, we can see that for any level of cache, the cache size is in kilobytes, and the associativity is 8. To target such a cache, the decision tree to be used needs to be deep. This is because the total number of memory objects should at least be a few times greater than the cache capacity. For a cache size of 32768 bytes with an associativity of 8, we need a decision tree having a depth of at least 15 or 16. Then, to represent this tree in an ILP properly, we need to further increase the size of the memory access sequence. Let's say that for the L1 D-Cache from the above table, we have a tree depth of 15 and an access sequence of around 500, which is already very low for a tree this deep. Even then, the number of variables in the ILP totals to around 25 million. This number is way too high to be computed with realistic processing power within reasonable time.

Chapter 9

Future Work

Integer Linear Programming is a promising approach to cache optimization because, when compared to machine learning approaches, this provides a globally optimal solution to the problem. This helps in providing much tighter run-time bounds which leads to a more accurate Worst case execution time analysis. With minor improvements in the work discussed in the previous sections, ILP formulation can be used for producing a realistic cache-aware layout that can be executed on real LRU systems.

9.1 Changes in access sequence formation

In the formulation above, an access sequence is formed by taking random walks within the tree to represent the decision tree. Then, the resulting access sequence is fed to the ILP formulation. This approach has a few issues since the number of variables scales linearly with the increase in access sequence length. Instead of doing this, a case could be made for feeding the probabilities of each tree node directly to the ILP formulation and using these probabilities to get a cache mapping. This would possibly eliminate the inaccuracies developed by the access length formation and could even lower the total variable count.

9.2 Pre-fetching of memory blocks

Along with the ILP formulation described in the study, certain blocks which appear one of the other with high probability can be prefetched to increase the hit ratio. Cache-graphs discussed in [2] can be employed for formalizing this and implementing the idea of prefetching in an ILP formulation.

9.3 Reduction of helper variable count

Helper variables in the above ILP formulation are used for carrying out operations such as categorizing each access as a hit and accessing each block's age. The count of such variables is very high because all these operations are carried out in several steps, each involving a 2D and sometimes even a 3D variable space. Optimizations in the usage of such variables can lead to a significant reduction in the variable count.

Appendix A

Code

The python notebook implementation of the LRU ILP formulation discussed in the study above can be found **here**

Bibliography

[1] Sebastian Buschjager et al. “Realization of Random Forest for Real-Time Evaluation through Tree Framing”. In: *2018 IEEE International Conference on Data Mining (ICDM)*. 2018, pp. 19–28. DOI: 10.1109/ICDM.2018.00017.

[2] Y.-T. S. Li, S. Malik, and A. Wolfe. “Cache modeling for real-time software: beyond direct mapped instruction caches”. In: *17th IEEE Real-Time Systems Symposium*. 1996, pp. 254–263. DOI: 10.1109/REAL.1996.563722.

[3] Reinhard Wilhelm. “Why AI + ILP Is Good for WCET, but MC Is Not, Nor ILP Alone”. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Bernhard Steffen and Giorgio Levi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 309–322. ISBN: 978-3-540-24622-0.