Computer architecture - HOMEWORK 2 - ECGR-5181

Observed and Documented by: Sumukh Raghuram Bhat – 801131997

Problem 1:

The DineroIV which was used in the previous homework is used to exploit the prefetching capabilities. But this time it is used for different prefetching policies such as -tfetch, -pfdist and -pfabort.

For this problem, Cache size is chosen to be 32 KB, cache type to be unified, block size to be 32 and direct associativity on the trace file from Homework 1 which is the trace.din file and on an I5 processor.

Prefetching Policy	Hit rate	Miss rate
-pfdist	99.63	0.32
-pfabort	99.63	0.32

-pfdist: It will set the prefetch distance in sub-blocks. If prefetch distance is 2, it means the next prefetch sub-block is 2 blocks from the most recently fetched block from the memory.

-pfabort: This gives the percentage of prefetches that have been aborted. It is used to examine the effects of data references blocking prefetch references from reaching a cache that is combined.

-tfetch: tfetch is the switch that provides various options for prefetching policies. The various policies are: d, a, m, t, l and s.

1. d -demand: fetches the block from the main memory on demand, as and when needed.
2. a -always: if the switch is enables, after every demand reference, it gets fetched from the main memory
3. m -missed: fetch takes place after every demand miss
4. t -tagged: fetch takes place after first demand miss to a sub block
5. l -load forward: works like always within a block. It will not fetch from other blocks. It works only for sub-block placement
6. s -sub-block: it is similar to always within a block except when references near the end of the block

Observation: -pfdist and -pfabort do not work with d as this is the demand fetch and for a, prefetching happens after every demand reference.

The command used - ./dineroIV -l1-usize 32768 -l1-ubsize 32 -l1-uassoc 1 -l1-ufetch d -informat d<trace.din>hw2_d1.out. This is for the unified demand fetching which provides miss rate of approximately 32%.

Similarly, many simulations were done for different values and are tabulated as below:

The commands used is as following - ./dineroIV -l1-usize 32768 -l1-ubsize 32 -l1-uassoc 1 -l1-ufetch a -l1-upfdist 1 -lq-upfabort 50 -informat d<trace.din>x1.out.

With the use of a:

Prefetching Policy		Total fetches	Demand fetches	Prefetch fetches	Total misses	Demand misses	Prefetch misses	Miss rate
-pfdist	-pfabort							
1	0	1560331	832477	727964	2412	1528	884	0.15
1	50	1196728	832477	364305	2338	1528	808	0.20
3	0	1560638	832477	727964	2444	1587	885	0.17
3	50	1196728	832477	364305	2538	1661	887	0.21
10	0	1560638	832477	727964	2453	1577	869	0.15
10	50	119728	832477	364305	2559	1747	812	0.21

Observation: The demand or compulsory misses are increasing with the increase in the two switches (-pfdist and -pfabort).

The miss rate increases with increase in -pfdist and -pfabort.

With the use of m:

Prefetching Policy		Total fetches	Demand fetches	Prefetch fetches	Total misses	Demand misses	Prefetch misses	Miss rate
-pfdist	-pfabort							
1	0	833373	832477	896	2119	1468	651	0.25
1	50	832935	832477	254	2040	1704	336	0.24
3	50	832925	832477	448	2041	1714	327	0.25
10	50	832924	832477	447	2072	1771	301	0.25

Observation: The prefetching takes place only after a demand miss and thus we can see a difference in the cache miss rate. The demand misses increase with increase when the switches are used. The miss rate increases with increase in -pfdist and the miss rate decreases as -pfabort increases.

With the use of t:

Prefetching Policy		Total fetches	Demand fetches	Prefetch fetches	Total misses	Demand misses	Prefetch misses	Miss rate
-pfdist	-pfabort							
1	0	833073	832477	619	2119	1768	361	0.25
1	50	832735	832477	400	2106	1794	236	0.24
3	50	832933	832477	387	2041	1714	227	0.24
10	50	832929	832477	397	2072	1871	227	0.25

Observation: The demand misses increase with the usage of the switches. The miss rate increases with increase in -pfdist and it decreases with increase in -pfabort.

With the use of l:

Prefetching Policy		Total fetches	Demand fetches	Prefetch fetches	Total misses	Demand misses	Prefetch misses	Miss rate
-pfdist	-pfabort							
1	0	845327	845327	0	1768	1768	0	0.24
1	50	845327	845327	0	1768	1768	0	0.24
1	100	845327	845327	0	1768	1768	0	0.24

Observation: This prefetching policy has absolutely no effect on the demand misses and the miss rate as well.

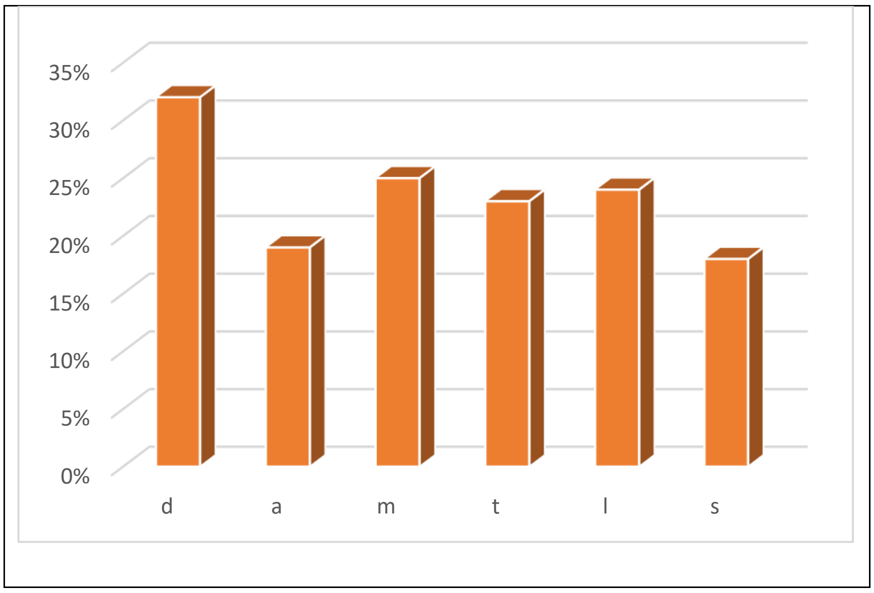
With the use of s:

Prefetching Policy		Total fetches	Demand fetches	Prefetch fetches	Total misses	Demand misses	Prefetch misses	Miss rate
-pfdist	-pfabort							
1	0	1661641	845327	718925	1768	1768	0	0.14
1	50	1195762	845327	391126	1768	1768	0	0.16
1	100	845327	845327	0	1768	1768	0	0.24

Observation: By using this prefetching policy, we see that there is no effect on demand misses but there is a lot of change in the miss rate.

The comparison:

The average cache miss rate (in %) was plotted with its respective prefetching policy.



Prefetching policy	Average Cache miss rate (in %)
d	32%
a	19%
m	25%
t	23%
l	24%
s	18%

Observation: The number of misses will increase if the prefetch distance increases. It is very hard to determine the best prefetching policy. There are a lot of parameters that must be considered in order to decide the best policy but we limit our parameters to just -pfdist and -pfabort. Using a prefetcher is advisable as long as there is an improvement with the hit rate.

Problem 2:

Row and Column major is used. In the row-major memory layout, the values are stored in successive row elements and in the column-major layout, the values are stored in successive column elements.

1. **By gprof:** This can be used for profiling and the time taken to execute each function can be determined. Both the row-major and column-major functions are combined and the performance is evaluated.

Function	Number of calls	%time
Row_major	1	24.37
Column_major	1	81.83

Observation: The run time for the function row_major is significantly low when compared to that of column_major function. This is because of the memory layout of the system on which this program was executed and tested which is row major.

2. **By pin tools:** The cache performance of the row-major and column-major functions can be analyzed. We can analyze the cache hit and miss rates for the functions. Different levels of cache are analyzed.

Cache Level	Hit Rate (%)	
	Row-major	Column-major
L1(data cache)	92.47	92.44
L2	75.81	75.76
L3	1.26	1.01

Observation: The hit rate is high for row major program than the column major one. This indicated that the system’s RAM uses row-major for its memory layout. Row major is kept in a contiguous way and hence spatial locality is preserved.

Problem 3:
Pin was used to create the memory trace for Dhrystone and Linpack Benchmarks.

The cache simulator of previous homework is modelled for data cache size 16KB, cache block size of 32B and Associativity of 4-way.

A stride prefetcher design **is tried** in python next to the cache simulator with the Prefetcher size 500B, 1KB, 2KB, 4KB and prefetching confidence bits of 2 and 3 respectively. Prefetch requests were queued in a 32-entry request queue.

The performance comparison based on accuracy and aggressiveness and other performance parameters:

Accuracy	**Lateness**	**Bandwidth Usage**	**Action**
High	Late	Low	Increase aggressiveness
High	Not Late	Low	Increase aggressiveness
Low	Late	Any	Increase aggressiveness

Formula:

Prefetcher Accuracy = Number of Used Prefetches / Number of Sent Prefetches

Coverage = Prefetched misses / All Misses

Observation: Performance of the Cache with prefetcher should be improved and it can be seen that there is an amount of overhead due to the implementation of the prefetcher.

Prefetcher accuracy would be nearly **30-40%** on average making a fair amount of reduction in the L2 miss rates. Hence the percentage of compulsory misses covered by the prefetcher can be seen. The prefetcher monitors the address and the outcome of each memory references from the CPU and issue prefetch requests via a finite-capacity request queue. Each memory block when first referenced causes a compulsory miss. This implies that the number of compulsory misses is the number of distinct memory blocks ever referenced. They are called **cold misses** too. Hence such Cold misses can be avoided when block is prefetched.

Problem 4:

"Cooperative Prefetching: Compiler and Hardware Support for Effective Instruction Prefetching in Modern Processors"

The research paper presents its objective of having a Cooperative Prefetching method which uses both hardware and software prefetching together to provide better prefetching in order to reduce the miss latency in modern processors. Since instruction cache miss latency is a major performance bottleneck for many applications, the main aim of the paper is to have prefetching early enough to improve the speedups and thus provide reduction in miss rates. This idea is implemented by having a fully automatic prefetching by the use of an aggressive hardware sequential prefetcher (use of 2-bit counter to dynamically detect next 8 line) along with a novel prefetcher (compiler inserted prefetcher with prefetcher filtering) which inserts instruction prefetches explicitly into executable to prefetch well ahead in advance to get better coverage and hiding miss latency without polluting the cache. The paper further elaborates by showing how the existing instruction prefetching techniques (target line prefetching, Markov prefetching, wrong path prefetching, etc.) is no longer effective for modern processors since they fail to prefetch the non-sequential instructions. The concept is also made to run experimentally and prefetch distance, bandwidth and the cache latencies are varied as a part of the experiment to show that the average speedups is increased by twice. The experimental analysis of the paper also helps us to conclude that the proposed idea is robust and can be employed for effective instruction prefetching in modern processors.

Benefits: This type of prefetching provides good results for the non-sequential instructions. The software prefetching (compiler inserted prefetcher) reduces the load on the programmer. The novel approach speedups average 13.5% on out-of-order superscalar processors by hiding the instruction stall time which is double the amount in existing schemes (6.5%). Based on various comparisons, it can be observed that this type of prefetching helps in removing more than 50% of latency i.e., by reducing the late prefetcher misses. Even when the instruction cache size is increased, this paper provides a solution which is more effective. It is also cost effective. The prefetch filtering and compiler-inserted prefetching components of the design are essential and complementary and the scheme is robust with respect to variations in miss latency and bandwidth.

Negatives: The research was conducted on a simulator and not using a real processor. The prefetcher distance is one of the tradeoffs with the performance as prefetcher distance should be chosen large enough to hide the expected miss latency. The sequential and non-sequential accesses of instructions are considered to be done separately which does not happen practically. Non-sequential accesses are prefetched using software-based approach which decreases the instruction fetch bandwidth. A lot of assumptions are made in this research, such as, the associativity, block size, cache size, etc.