Simulation Homework 1

Brandon Saldanha

***Abstract*—This document is the submission for CSE240C Homework 1. The document discusses the winners of the 1st Instruction Prefetching Championship. Out of these, DJOLT and ¡insert another one¿ are discussed in detail and a space exploration study is performed on these.**

I. D-JOLT: Distant Jolt Prefetcher

A. Summary

DJOLT is a prefetcher that focuses on the history of function calls and misses. This is due to the correlation between prefetcher characteristics and association of history and miss addresses, the length of the history and the distance to the prefetch target. The architecture for DJOLT consists of a long-range prefetcher, a short-range prefetcher and a fallback prefetcher. For the distributed traces, according to the rules of IPC1, DJOLT shows a 28.9% improvement compared to a processor without instruction prefetching.

The study focuses on the return-address-stack directed instruction prefetcher (RDIP). A return address stack (RAS) is used to implement an RDIP, which generates signature form addresses recorded in the RAS and associates cache misses with this address. The RDIP assumes that the processor will access the same cache locations for signature values.

The main characteristic parameters for the DJOLT prefetcher are: The signature generation method implemented in the paper is FifoRetcent which generates signatures using a first-in, first-out FIFO that records a called address in every function call and a counter that counts the number of successive returns.

DJOLT takes a hybrid configuration for the prefetchers. Each prefetcher covers the same target, and one prefetcher covers the target when the other fails to predict.

- Long-range prefetcher: The prefetcher has long histlen and distance. As it has to predict prefetches that have a long distance, it is equipped with a longer histlen to compensate for lower accuracy.
- Short-range prefetcher: The prefetcher has shorter histlen and distance. When the long-range prefetcher fails to predict the address, the short-range prefetcher comes into play.
- Fallback prefetcher: A modest stream prefetcher predicts the address when the previous two prefetchers fail to do so.

DJOLT achieves a 28.9% IPC speedup compared to no instruction prefetching.

B. Design

DJOLT resource usage:

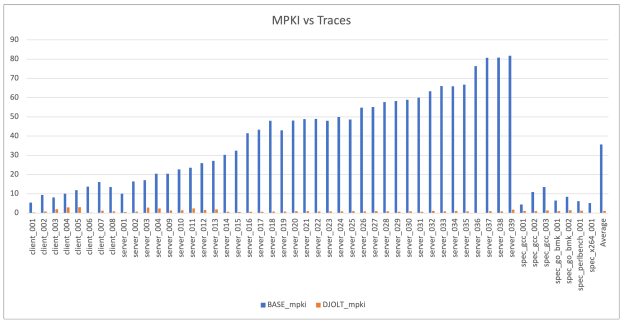
Long-range Prefetcher	Signature Generator	FIFO (7*32 + 3 bits), return counter (32 bits)		259
	Signature Queue	Signature (23 bits)	15 entries + misc.	349
	Miss Table	Tag (12 bits), LRU (2 bits), miss vector (2*31 bits)	2048 sets, 4-way (8192 entries)	622592
Short-range Prefetcher	Signature Generator	FIFO (4*32 + 3 bits), return counter (32 bits)		162
	Signature Queue	Signature (23 bits)	4 entries + misc	94
	Miss Table	Tag (13 bits), LRU (2 bits), miss vector (2*31 bits)	1024 sets, 4-way (4096 entries)	315392
Extra-miss Table		Tag (15 bits), LRU (2 bits), miss vector (2*31 bits)	256 sets, 4-way (1024 entries)	80896
Fallback Prefetcher	Stream Train Table	Valid (1 bit), LRU (4 bits), line address (58 bits), counter (2 bits)	16 entries	1040
		Valid (1 bit), LRU (4 bits), line address (58 bits)	16 entries	1008
Upper Bit Table		Valid (1 bit), upper bit (40 bits)	31 entries	1271
				1023062

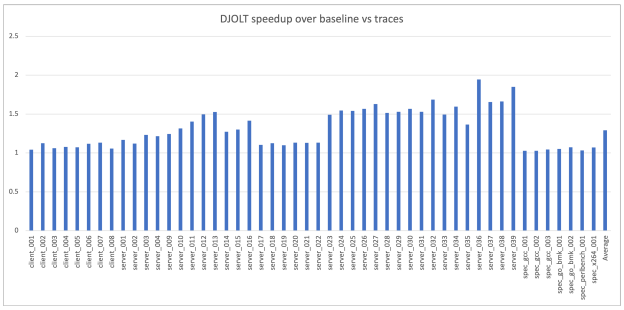
The behaviour of the DJOLT prefetcher is as follows:

- Issuing prefetch: When a call/return instruction is fetched, D-JOLT calculates new signatures based on FifoRetcnt for each prefetcher and these are then pushed into the signature queue. Then D-JOLT accesses the miss table, upper bit table, and extra-miss table with the updated signature. D-JOLT reconstructs raw prefetch line addresses from the obtained data, and issues prefetches.
- Learning miss address: When a cache miss occurs, D-JOLT records missed line addresses into the tables using a signature obtained from the signature queue. This signature is one that was pushed to the queue distance times ago. A missed line address is divided into the upper and lower bits, and the bits are recorded into each table. If a signature hits on the miss table and the missed line address is not within the range of the bit vector in the entry, the missed line address is recorded in the extra-miss table.
- Signature generation algorithm: When a call/return instruction is fetched, FIFORETCNT calculates a new signature. When a call instruction is fetched, the address is pushed into the queue, and the return counter is reset to zero. When a return instruction is fetched, the queue is not updated, and the return counter is incremented. After fetching a call/return instruction, it calculates a new signature from the histlenaddresses obtained from the queue head and the return counter.

C. Results

Clearly, D-JOLT performs much better than the baseline which has no instruction prefetching in the l1 level. DJOLT achieves a 28.9% IPC speedup compared to the baseline which has no L1i prefetching.

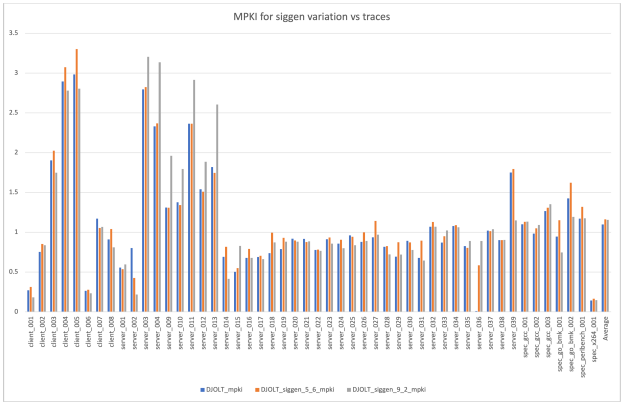




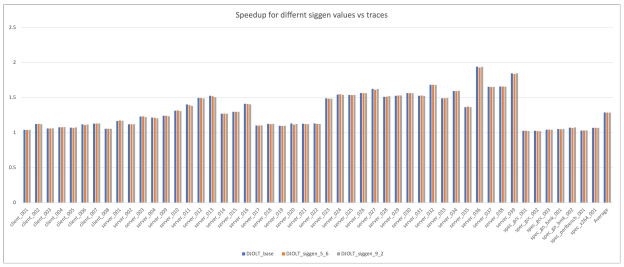
D. Space Exploration

The first parameter that is explored is the 'siggen' variable that decides the number of entries used to create the index to the tables. In order to keep the hardware budget the same, while increasing the siggen bits for the long-range prefetcher, the bits for the short-range prefetcher are reduced and vice versa.

	Baseline DJOLT	Case 1	Case 2
Long-range Prefetcher	7	5	9
Short-range Prefetcher	4	6	2



After changing the siggen parameter we see a 5.8% and 5.0% increase in MPKI for case 1 and case 2 respectively.

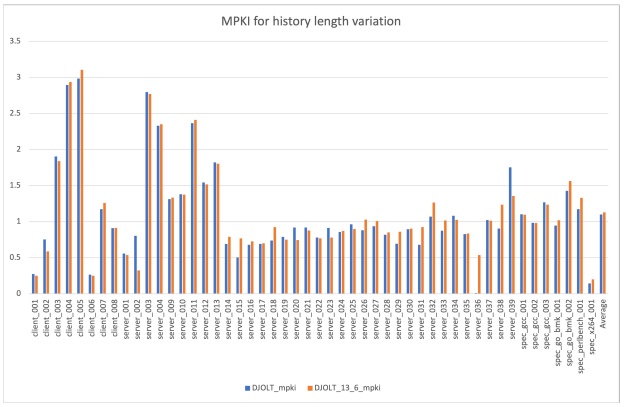


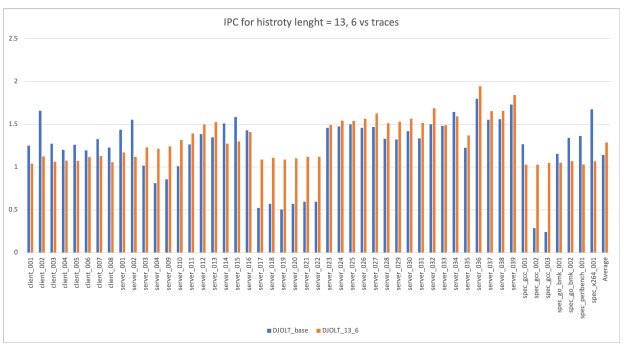
A 12% decrease over the baseline DJOLT in IPC occurs for both these cases.

The next parameter that is varied is the history length. The history length corresponds to how far ahead in the instruction queue the prefetcher fetch address. Again, in order to maintain the same hardware budget, while the history for the long-range prefetcher was increased, the one for the short-range prefetcher was decreased and vice versa.

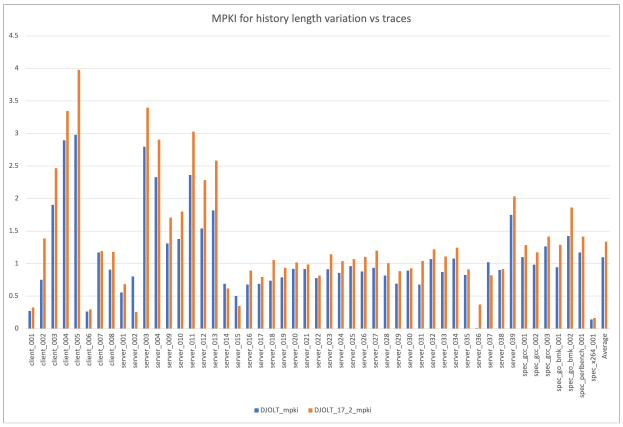
It is observed that when history lengths for the long-range prefetcher and short-range-prefetcher are 13 and 6 respectively, the prefetcher performs just a little bit worse (2.61% increase in MPKI) than the base version of D-JOLT that has history lengths, 15 and 4 for the long-range prefetcher and short-range-prefetcher respectively. This depreciation in performance is probaly due to the short range prefetcher's increase in inaccuracy as it now has to prefetch values for instructions further down the pipeline with modest hardware resources.

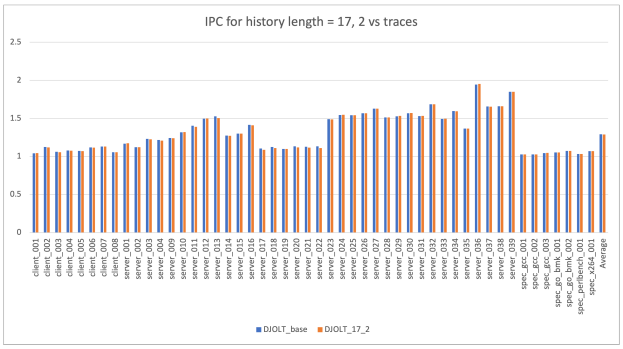
The IPC also increases by 12.7% for this change in history length.





When history lengths for the long-range prefetcher and short-range-prefetcher are 17 and 2 respectively, the prefetcher performs much worse (24% increase in MPKI) than the base version of D-JOLT that has history lengths, 15 and 4 for the long-range prefetcher and short-range-prefetcher respectively. This is due to the fact that having a large history length for the long-prefetcher increases inaccuracy in prediction and having too short a history length for the short-prefetcher does not leave enough time for the prefetcher to get data. This is accompanied with a 12.7% increase in IPC.





E. Varying hardware budget

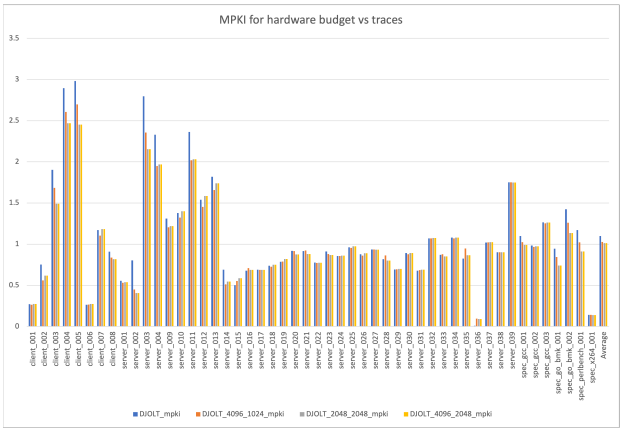
Base DJOLT:

Long-range prefetcher: 2048 sets, 4 way

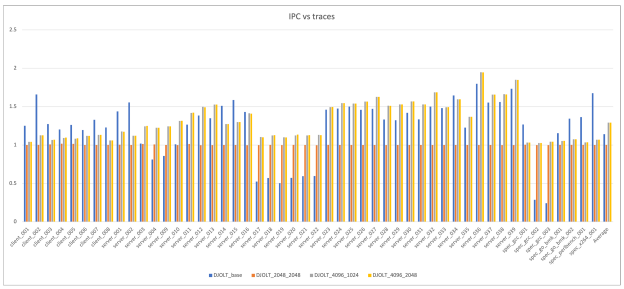
Short-range prefetcher: 1024 sets, 4 way

Increasing the hardware budget:

Prefetcher	Case 1	Case 2	Case 3
Long-range	4096 sets, 4 way	2048 sets, 4 way	4096 sets, 4 way
Short-range	1024 sets, 4 way	2048 sets, 4 way	2048 sets, 4 way

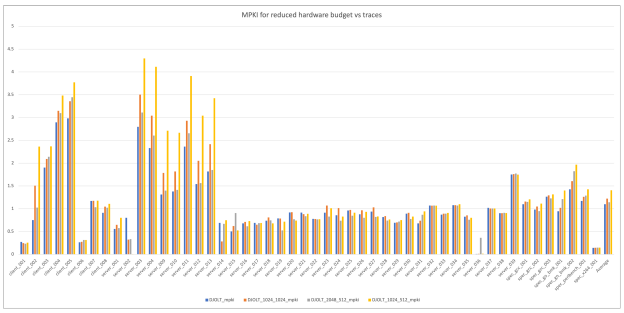


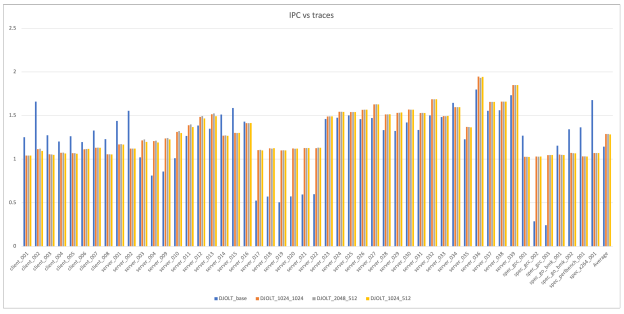
Increasing the hardware budget, we observe an increase in average MPKI of 6.4%, 7.5% and 8.0% for case 1, case 2 and case 3 respectively.



Decreasing the hardware budget:

Prefetcher	Case 1	Case 2	Case 3
Long-range	1024 sets, 4 way	2048 sets, 4 way	1024 sets, 4 way
Short-range	1024 sets, 4 way	512 sets, 4 way	512 sets, 4 way





Reducing the hardware budget results in worse performance in all cases with an increase in average MPKI of 11.3%, 3.9% and 27.8% for case 1, case 2 and case 3 respectively.

F. Computations for prefetching

For DJOLT, the hash function creates a signature based on the return-address-stack this signature is used to index into the miss table. We get the address for the prefetch from this table. A prefetch request is initiated whenever there is a change in the signature value.

II. The FNL+MMA Instruction Cache Prefetcher

A. Summary

It is essential to decide which events trigger a prefetch to perform well. The FNl+MMA prefetcher proposes to initiate prefetch requests on I-Shadow cache misses. The I-Shadow cache is s small cache that monitors misses from demands to the cache. Spatial locality tells us that the following line is likely to be used shortly, but fetching every next line leads to over fetching and high bandwidth consumption. It is imperative to determine the likelihood that the following line will be needed soon. The FNL overcomes this difficulty by selecting if the following line will be required and achieves a 16.5% speedup while doing so. On the other hand, the sequence of I-cache misses is partially predictable if no prefetching is used. It can be said confidently that when a particular block B misses, the nth block required after that block is often the same block Bn. This can be said confidently for n as large as 30. The MMA predictor takes advantage of this property, and with a 96KB FNL+MMA block, the machine achieves a 28.7% speed up and decreases the I-cache miss rate by 91.8%. The one downside to this implementation being the 38.3% increase in L2 access.

On an I-Shadow cache miss, FNL prefetches contiguous blocks, but it cannot fetch non-contiguous blocks. The MMA prefetcher targets this limitation of the FNL prefetcher.

Without prefetching, it is possible to predict with high accuracy the next block that should be prefetched when a block miss occurs. This property holds even for the I-Shadow cache that monitors demand accesses on the I-cache. Using this property, when blocks B and B' are missing consecutively in the I-Shadow cache and B' is missing in the I-Cache B' is associated with Block B in a 'next prediction' table. When the same association occurs twice, the entry is considered highly correlated, and on the next occurrence of Block B, B' is prefetched into I-cache.

The only downside to this method is that prefetches are triggered too late. To avoid these late prefetches, the predictor predicts the block that should be prefetched 'n' blocks after a miss on block B.

B. Design

FNLMMA resource usage:

I-Shadow Cache	192 * 17-bit entries	408 bytes	
Touched table	64k * 1-bit entries	8k bytes	
WorthPF table	64k * 2-bit entries	16k bytes	
Miss ahead prediction table	8k * 71-bits entries	71k bytes	
MMA filter	16 * 58-bit entries	116 bytes	
FNL filter	128 * 17-bit entries	136 bytes	
		97940 bytes ≈ 96k bytes	

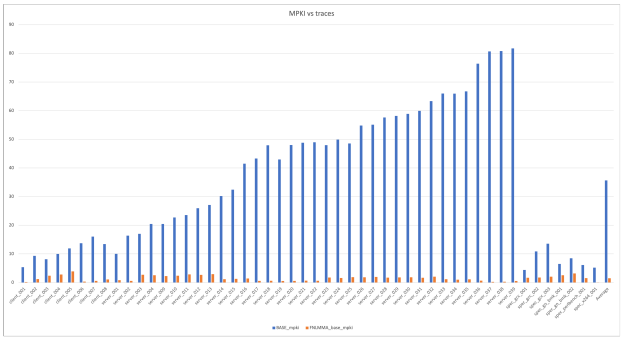
The I-Shadow cache is a 3-way associative cache with LRU replacement policy. It uses 15-bit partial tags, with each entry being 17-bit wide (15 bits + 2 replacement bits). The FNL predictor helps prefetch contiguous blocks in memory up to the following 'k' lines. Block B+1 is considered for prefetching after block B if this access is done during the interval of 'L; I-Shadow misses after the I-Shadow miss on block B. The FNL predictor monitors the future accesses using two direct-mapped tables. The 'Touched' table is a 1-bit entry table that monitors a cache block recently requested. The WorthPF is a 2-bit entry table that predicts that the following line should be prefetched. Both tables have 64k entries. On an I-Shadow miss on block B, Touched[B] is set to 1, and if Touched[B-1] is also set, the WorthPF[B-1] is set to 3. On each interval of 8192 misses on the I-Shadow cache, if Touched[B] = 1, then WorthPF[B] is decremented and Touched[B] is reset. This ensures that WorthPF[B] is non-null if B was touched during one of the 3*8192 misses on the I-Shadow cache. On an I-Shadow miss on block B, if WorthPF[B] is non-null, the next block (B+1) is prefetched. This is extended to the subsequent five blocks (if WorthPF[B+1] is non-null, prefetch block B+1 too).

As the above technique may lead to redundant block prefetching, a simple FNL filter is implemented. When prefetch for block B is triggered, block B-1 is searched in the FNL filter. If block B-1 exists, blocks B, B+1, B+2, B+4 have been prefetched earlier and need not be prefetched again. Only block B+5 needs to be prefetched. The FNL filter is implemented as a 128-entry, 4-way associative filter with FIFO replacement.

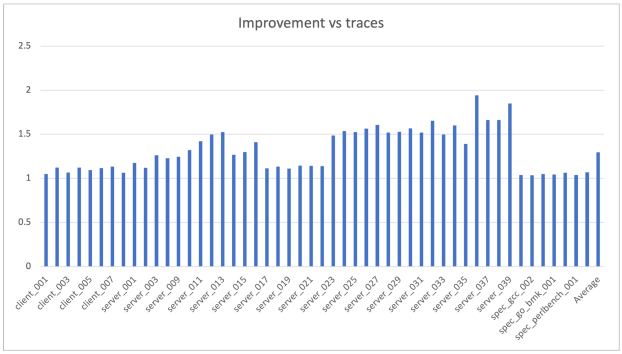
The MMA prefetcher is implemented using a 8K-entry skewed-associative next miss prediction table. An average 12.2 % speed-up is obtained over no-prefetching.. Only 23.2% of the misses are eliminated, but the extra demand on the L2 cache is limited to a mere 1.0%. In practice the performance benefit is mainly brought by decreasing of the average miss penalty from 21.9 cycles to only 10.5 cycles.

C. Results

For the updated code, FNL+MMA Instruction Cache Prefetcher achieves 1.52 MPKI for the given traces - a 95.7% decrease in MPKI as compared to the baseline with no prefetcher; 3.9% more than the number reported with the old code in the IPC1 competition.



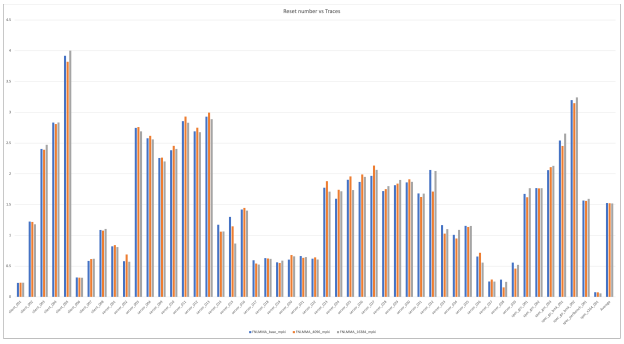
FNLMMA shows a 29% improvement in IPC over the baseline with no prefetching implemented.



D. Space Exploration

The first parameter that is varied is the reset value for the FNL prefetcher. This value corresponds to the miss interval for which the I-Shadow accesses for a block are used to model the correlation between blocks. Every entry of the Touched table is partially reset after a period L of I-Shadow misses.

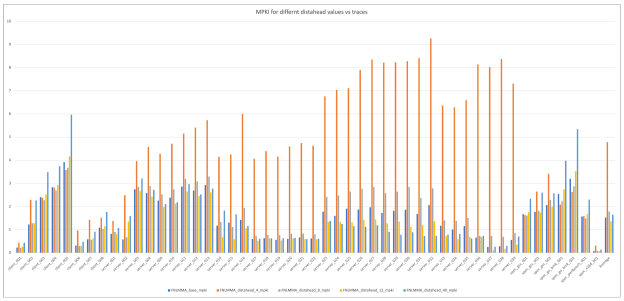
	Baseline NFL	Case 1	Case 2
Reset value	8192	4096	16384



The change in MPKI for case 1 and case 2 is negligible as compared to a baseline NFL prefetcher.

The next parameter that is varied is the 'distahead' for the MMA prefetcher. This variable corresponds to the number of blocks following the current block that the prefetcher will try to predict and prefetch.

	Baseline	Case 1	Case 2	Case 3	Case 4
Distahead value	10	4	8	12	40

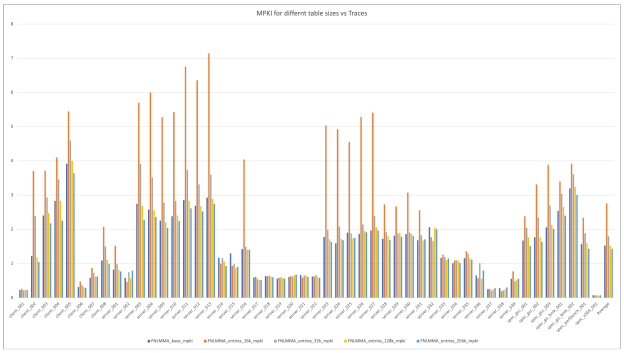


On average, case 3 gives us the best decrease (10.7%) in MPKI as compared to the baseline FNLMMA prefetcher. Case 2 has an increased MPKI (17.2%) that can be attributed to not having enough time for a prefetch. This is further proven by the really bad MPKI achieved by Case 1 (213.5% increase as compared to base NFLMMA). Unsurprisingly, Case 4 also shows an increase in MPKI (7.8%), and this can be attributed to the large distance ahead that it has to fetch ahead which may not be required as we reach the particular block due to branches.

E. Varying Hardware budget

To test different hardware budgets, the size of tabels for FNL and MMA prefetchers are varied over the following cases:

	Baseline	Case 1	Case 2	Case 3	Case 4
Entries	64k	16k	32k	128k	256k



Increasing the table sizes does not yield much performance increase but lowering the budget shows an immediate impact to the MPKI values. The hardware increase cases 3 and 4 show a decrease of 0.3% and 6.1% respectively, whereas the decrease in hardware budget for cases 1 and 2 give a increase of 80.8% and 17.9% in MPKI values.

F. Computations for prefetching

In the FNL prefetcher, when an I-Shadow cache miss occurs, the WorthPF table is accessed first at the index corresponding to the block miss. If this bit is set, the prefetch

of the next contiguous block is triggered. This is repeated for five blocks. Once prefetch is activated, the FNL filter table is accessed, and the entire table is checked for recently prefetched blocks. If the blocks that have prefetch triggered in the previous step are present in the Filter, these prefetches are ignored to reduce polluting the L2 bus with irrelevant prefetches.

For the MMA prefetcher, when a miss occurs for the same two blocks a specific 'distance' (specific number of fetches) away, they are associated together. When this association occurs twice, this entry is considered highly correlated. On the next occurrence of the first block address missing from the I-Shadow cache, a prefetch is triggered for the other block. This prefetch request is checked throughout the MMA filter, and if it has not been requested recently, it is prefetched.

III. Comparing prefetchers

It is clear that FNLMMA performs better than DJOLT for a tighter budget constraint. This is due to:

- DJOLT requires more time to generate a prefetch than FNLMMA. As the signature generation in DJOLT is much more involved than a simple table lookup in FLNMMA. This causes prefetches to take longer and may not be available in time.
- DJOLT uses 3 prefetchers to ensure code coverage, thus having redundant tables and requiring access to multiple tables and issuing new prefetch requests all the time. FLNMMA on the other hand optimizes this to two prefetchers and uses lesser prefetchers that have orthogonal prefetch strategies.
- DJOLT issues a new fetch every time the signature value for it changes, thus polluting the L2 line and consuming increased bandwidth. FNLMMA on the other hand, implements a Filter table that ensure lower bandwidth utilization by skipping redundant prefetch requests.

Hence, NFLMMA is more timely than DJOLT prefetcher.

IV. The Entangling Instruction Prefetcher

The EIP prefetcher uses the correlation between instructions to determine which instruction to prefetch to ensure that instructions are prefetched in time. A history table of cache misses and accesses is maintained, and whenever there is a miss, the timestamp of that miss is recorded. EIP then calculates the fetch latency for that instruction and finds the instruction that should have ideally requested for the missed instruction to have no fetch latency.

This way, instructions are 'entangled' with instructions that should trigger the prefetch to ensure the timely arrival of the requested instruction. This method ensures that EIP is agnostic to the application characteristics and can perform well for any workload.

To reduce storage overhead caused by entangling multiple pairs of instructions, EIP only entangles the head of basic blocks where a basic block represents the set of consecutive cache lines. EIP achieves a 29.5% speedup over the baseline processor with no prefetcher and has an increased cache hit ratio of 0.996 (22.5% over the baseline). Additionally, EIP has a 95.6% code coverage rate, ensuring cache misses are very low.

V. Running the code

github link: https://github.com/Brandon451/HW1_CSE240C
The script to run is script.sh

After selecting the prefetcher to run in the script by modifying line number 21 run the script, results will be generated in the folder. Use results generated to compare.