Program 1: tr-simpleloop.ref

FIFO	memsize = 50	memsize = 100	memsize = 150	memsize = 200
Hit rate	70.8399	72.9592	73.3516	73.4301
Hit count	7220	7436	7476	7484
Clean Eviction Count	209	45	16	12
Dirty Eviction Count	2713	2611	2550	2496

CLOCK	memsize = 50	memsize = 100	memsize = 150	memsize = 200
Hit rate	72.6158	73.6264	73.6754	73.6754
Hit count	7401	7504	7509	7509
Clean Eviction Count	96	4	0	0
Dirty Eviction Count	2645	2584	2533	2483

EXACT LRU	memsize = 50	memsize = 100	memsize = 150	memsize = 200
Hit rate	72.7041	73.6656	73.6852	73.6852
Hit count	7410	7508	7510	7510
Clean Eviction Count	90	2	0	0
Dirty Eviction Count	2642	2582	2532	2482

OPT	memsize = 50	memsize = 100	memsize = 150	memsize = 200
Hit rate	73.9453	74.1992	74.1992	74.1992
Hit count	7572	7598	7589	7589
Clean Eviction Count	16	0	0	0
Dirty Eviction Count	2602	2542	2492	2442

Program 2: tr-blocked.ref

FIFO	memsize = 50	memsize = 100	memsize = 150	memsize = 200
Hit rate	99.7318	99.8207	99.8253	99.8687
Hit count	2411570	2413720	2413831	2414881
Clean Eviction Count	4181	2758	2652	1877
Dirty Eviction Count	2255	1478	1423	1098

CLOCK	memsize = 50	memsize = 100	memsize = 150	memsize = 200
Hit rate	99.7828	99.8339	99.8370	99.8681
Hit count	2412803	2414039	2414115	2414867
Clean Eviction Count	2866	2616	2574	1927
Dirty Eviction Count	2337	1301	1217	1062

EXACT LRU	memsize = 50	memsize = 100	memsize = 150	memsize = 200
Hit rate	99.7842	99.8435	99.8442	99.8472
Hit count	2412837	2414271	2414288	2414361
Clean Eviction Count	2818	2605	2558	2435
Dirty Eviction Count	2351	1080	1060	1060

OPT	memsize = 50	memsize = 100	memsize = 150	memsize = 200
Hit rate	99.8466	99.8755	99.8955	99.8955
Hit count	2414427	2415126	2415608	2415608
Clean Eviction Count	2572	1825	1305	1305
Dirty Eviction Count	1087	1085	1073	1073

Program 3: tr-matmul.ref

FIFO	memsize = 50	memsize = 100	memsize = 150	memsize = 200
Hit rate	60.9649	62.4788	98.8085	98.8266
Hit count	1760565	1804282	2853424	2853945
Clean Eviction Count	1083230	1061224	32943	32433
Dirty Eviction Count	42987	22226	1315	1254

CLOCK	memsize = 50	memsize = 100	memsize = 150	memsize = 200
Hit rate	63.9437	63.9532	98.8501	98.8606
Hit count	1846588	1846861	2854625	2854929
Clean Eviction Count	1040083	1039789	31974	31623
Dirty Eviction Count	1111	1080	1083	1080

EXACT LRU	memsize = 50	memsize = 100	memsize = 150	memsize = 200
Hit rate	63.9443	65.1485	98.8613	98.8616
Hit count	1846605	1881378	2854947	2854958
Clean Eviction Count	1040070	1005275	31656	31595
Dirty Eviction Count	1107	1079	1079	1079

OPT	memsize = 50	memsize = 100	memsize = 150	memsize = 200
Hit rate	79.6581	96.7867	99.0784	99.3329
Hit count	2300456	2795115	2861298	2868648
Clean Eviction Count	586319	91612	25379	17979
Dirty Eviction Count	1087	1085	1085	1085

Program 4: tr-overflow.ref

FIFO	memsize = 50	memsize = 100	memsize = 150	memsize = 200
Hit rate	97.2321	98.5185	98.7580	98.7580
Hit count	10960	11105	11132	11132
Clean Eviction Count	107	0	0	0
Dirty Eviction Count	155	67	0	0

CLOCK	memsize = 50	memsize = 100	memsize = 150	memsize = 200
Hit rate	97.7555	98.6781	98.7580	98.7580
Hit count	11019	11123	11132	11132
Clean Eviction Count	74	0	0	0
Dirty Eviction Count	129	49	0	0

EXACT LRU	memsize = 50	memsize = 100	memsize = 150	memsize = 200
Hit rate	97.8886	98.7402	98.7580	98.7580
Hit count	11034	11130	11132	11132
Clean Eviction Count	65	0	0	0
Dirty Eviction Count	123	42	0	0

OPT	memsize = 50	memsize = 100	memsize = 150	memsize = 200
Hit rate	90.0461	98.7580	98.7580	98.7580
Hit count	10150	11132	11132	11132
Clean Eviction Count	880	0	0	0
Dirty Eviction Count	192	0	0	0

Reason for Choosing overflow.c:
overflow.c is a program that constantly reaches the memory outside of declared array, which is interesting to me because those memory addresses even though calculable but their values are unpredictable. However, it appears in the result of all algorithm that if the memory size is large enough, there is no eviction count for any algorithm. Even though the output this program is random and uncertain, the addresses accessed by this program can be traced and mapped entirely with large enough memory size.

Comparison paragraph:
The graphs above demonstrates that as memory size increases, the hit rate of all the algorithms increases, but as more and more memory is given, such decrease in hit rate becomes less and less significant. Eventually, OPT algorithm reaches a maximum hit rate that it does not vary as memory size increases. Other algorithms approaches such limit as well. Even though theoretically FIFO suffers from Belady's Anomaly, but it is not very obvious from the graphs above. In comparison, LRU has the hit rate that is the closest to OPT algorithm. CLOCK algorithm has a hit rate just a little below LRU and FIFO has the worst outcome. The running time for LRU, however, is expensive since every reference takes O(memory size) time. Since CLOCK algorithm takes O(memory size) upon eviction which is rare comparing to reference, CLOCK is the most practical algorithm of them all.

LRU Description:

It can be observed that as memory size increases, the hit rate using LRU algorithm increases. When memory size is small(50), increasing memory size would cause hit rate to increase more significantly comparing to increasing the same amount of memory when memory size is large. Also, the total eviction count decreases as more memory is used.

OPT ALGORITHM

Datastructures:

hashmap_ptr → points to a hashmap of size memsize on heap

hashmap: for each occurance of vaddr, hash it into its index using hash-function

reference_string → points to a reference-string of size line-number on heap.

coremap_head → points to the struct vaddr in frame that will be evicted on next eviction. (A sorted linked list by the next occurance of each virtual-address. the higher the next occurance number is, the longest the virtual-address will not be referenced, the more likely it will be evicted.)

Example how it works:

Consider next 7 virtual addresses that will be referenced in order.

I 4000000

S fff000000

I 4001000

S 4101100

L 4000000

S 41300ef

I 4001000

reference_string upon initialization:

index 0: 4000000 → 0 → 4
1: fff000000 → 1
2: 4001000 → 2 → 6
3: 4101100 → 3
4: 4000000 → 0 → 4
5: 41300ef → 5
6: 4001000 → 2 → 6

Occurance sequence of each virtual address Implemented using struct allocated on heap

hashmap upon initialization

index 0: fff000000 → 41300ef → NULL
index 1: 4000000 → 4001000 → NULL
index 2: 4101100 → NULL

Note: each virtual address represents a struct that contains a pointer to the next struct in hashmap

Start Referencing: 4000000

reference_string: index 0: 4000000 → 4 → NULL
1: fff000000 → 1 → NULL
2: 4001000 → 2 → 6 → NULL
3: 4101100 → 3 → NULL
4: 4000000 → 4 → NULL
5: 41300ef → 5 → NULL
6: 4001000 → 2 → 6 → NULL

The head of occurance string gets removed

coremap-head (self defined linked list)

↓

Coremap : index 0 4000000 → 4

index 1 NULL

index 2 NULL

Reference fff000000

reference-string: index 0: 4000000 → 4 → NULL

1: fff000000 → NULL

2: 4001000 → 2 → 6 → NULL

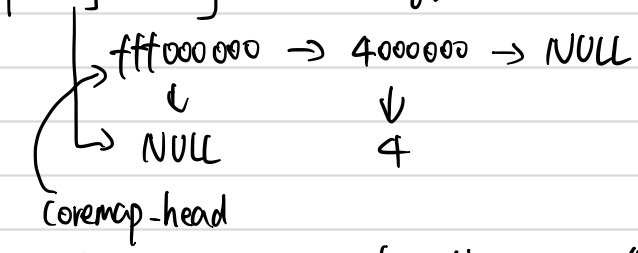
3: 4101100 → 3 → NULL

4: 4000000 → 4 → NULL

5: 41300ef → 5 → NULL

6: 4001000 → 2 → 6 → NULL

NULL has higher priority than any number



Reference 4001000 reference-string: index 0: 4000000 → 4 → NULL

1: fff000000 → NULL

2: 4001000 → 6 → NULL

3: 4101100 → 3 → NULL

4: 4000000 → 4 → NULL

5: 41300ef → 5 → NULL

6: 4001000 → 6 → NULL

Updated occurance-string

coremap_head : fff000000 → 4001000 → 4000000 → NULL

| ↓ ↓ ↓

Updated coremap_head NULL 6 4

↓ ↓

NULL NULL

Reference 4101100 :

reference_string index 0 : 4000000 → 4 → NULL

1 : fff000000 → NULL

2 : 4001000 → 6 → NULL

3 : 4101100 → NULL

4 : 4000000 → 4 → NULL

5 : 41300ef → 5 → NULL

6 : 4001000 → 6 → NULL

Evict coremap_head : fff000000

Bring in 4101100

coremap_head : 4101100 → 4001000 → 4000000 → NULL

↓ ↓ ↓

NULL 6 4

↓ ↓

NULL NULL

Now vaddr is placed here since NULL is greater than any number by our rule

.... AND SO ON

Time complexity analysis: Let n be # of referenced (line_number)
Let m be memsize

opt_init: Reads all referenced vaddr = O(n)
Malloc a hashmap of size m and assign every entry to NULL = O(m)
Total running time: O(m+n) ≈ O(n) if m<<n

opt_evict: Change pointer of coremap_head and returns = O(1)
Total running time: O(1)

opt_ref: Change pointer of occurance_sequence = O(1)
Add referenced vaddr to coremap_head linked list = O(m)
Total running time: O(m)

The overall time complexity seems pretty efficient, amortized running time is also low for each function.
But need large heap_size for mallocing data_structure O(n) space.