Gruppe 15 | Tobias Schoch, Luis Nothvogel
Simulation wurde auf dem HTWG Container ausgeführt

22.2

FIFO:

```
./paging-policy.py --addresses=0,1,2,3,4,5,0,1,2,3,4,5 --policy=FIFO --cachesize=5 -c
```

Da FIFO die Abkürzung ist für "First in First out", wird bei FIFO immer die „älteste" Page entfernt, falls der Cache zu voll wird.

FIFO nutzt den kompletten Cache mit der Größe 5 aus um 5 Pages unterzubringen und hat dabei logischerweise 5-mal einen Miss, da die Pages nicht auf dem Cache liegen. Als jedoch eine sechste Page hinzukommt, wird Page „0" rausgeworfen, da diese als erste dem Cache beigetreten ist. Im nächsten Schritt wird erneut die Page „0" aufgerufen, die jedoch letzte Runde ausgestoßen wurde, weshalb wir erneut einen Cache Miss haben.

```
FINALSTATS hits 0   misses 12   hitrate 0.00
```

LRU:

```
./paging-policy.py --addresses=0,1,2,3,4,5,0,1,2,3,4,5 --policy=LRU --cachesize=5 -c
```

LRU steht für „Last-Recently-Used" und ist eine Policy, die bei einem vollen Cache die am längsten nicht benutzte Page rauswirft. Dabei geht die Policy bis zur vierten Page und speist alle mit einem Cache Miss in den Cache ein. Wenn nun der Cache voll ist, wird die Page rausgeworfen, die am längsten nicht verwendet wurde. In unserem Fall die 0. Da nun erneut die 0 eingelesen wird, wird nun die 1 rausgeworfen. Dieser Vorgang geht so weiter ohne einen einzigen Cache Hit.

```
FINALSTATS hits 0   misses 12   hitrate 0.00
```

MRU:

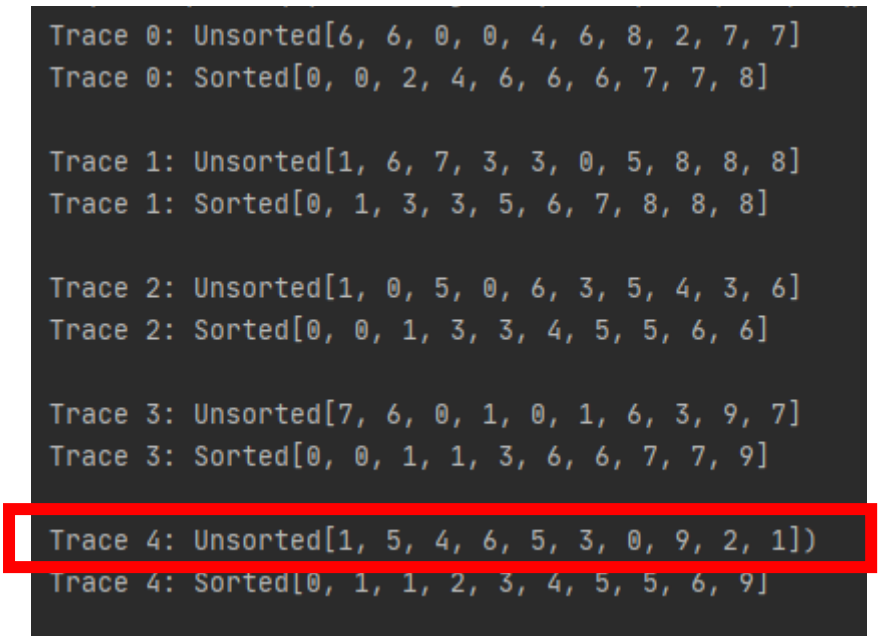
```
./paging-policy.py --addresses=0,1,2,3,4,5,4,5 --policy=MRU --cachesize=5 -c
```

MRU steht für "Most-Recently-Used" und ist genau das Gegenteil von einer LRU. Hier wird nämlich stets die neuste eingespeiste Page entfernt. Um den Cache nun wieder zu füllen, wird wieder bis zur fünften Page „4" eingelesen. Im Anschluss wird die 5 eingelesen in den Cache. Da der Cache nur eine Größe von 5 hat, muss davor noch eine Page entfernt werden. Bei MRU ist das die zuletzt benutzte Page, bei uns die „4". Danach wird die „5" eingelesen, und wiederrum die 4 entfernt. Dieser Vorgang kann sich unendlich oft wiederholen und hat stets keinen einzigen Cache Hit, sondern nur Cache Misses.

```
FINALSTATS hits 0   misses 8   hitrate 0.00
```

22.3

Wir haben in unserer Pythondatei insgesamt fünf verschiedene Traces ausgeben lassen und uns für Trace 4 entschieden:



Wir denken, dass die Trace insgesamt nicht gut performt, da es sehr viele unterschiedliche Pages gibt, von denen sich auch lediglich zwei wiederholen, ein solches Szenario kann aber dennoch sehr interessant sein.

FIFO:

Bei FIFO kann man es gut bereits am Array ablesen, indem man stets alle fünf aufeinanderfolgenden Pages vergleicht, ob eine Page mehrfach vorkommt. Dies geschieht bei der fünf. Ansonsten gibt es jedoch keine Cache Hits mehr.

1 Hit, 9 Misses = 10% Hitrate

LFU & MFU:

LFU und MFU gehen auf die Häufigkeit einer Page ein. „Least-frequent-use" und „Most-frequent-use". Dabei gibt es zwischen den beiden einen bedeutenden Unterschied. Die Pages die am häufigsten kommen, werden bei LFU in der Kick-Reihenfolge als letztes stehen. Bei MFU hingegen, wird die Page die als häufigstes vorkommt als erstes gekickt.

LFU:

				1	1 hinzugefügt	-	Miss
			1	5	5 hinzugefügt	-	Miss
		1	5	4	4 hinzugefügt	-	Miss
	1	5	4	6	6 hinzugefügt	-	Miss
	1	4	6	**5**	5 gefunden	-	**Hit**
1	4	6	3	5	3 hinzugefügt	-	Miss
4	6	3	0	5	0 hinzugefügt	1 Kick	Miss
6	3	0	9	5	9 hinzugefügt	4 Kick	Miss
3	0	9	2	5	2 hinzugefügt	6 Kick	Miss
0	9	2	1	5	1 hinzugefügt	3 Kick	Miss

MFU:

				1	1 hinzugefügt	-	Miss
			1	5	5 hinzugefügt	-	Miss
		1	5	4	4 hinzugefügt	-	Miss
	1	5	4	6	6 hinzugefügt	-	Miss
	5	1	4	6	5 gefunden	-	**Hit**
5	1	4	6	3	3 hinzugefügt	-	Miss
1	4	6	3	0	0 hinzugefügt	5 Kick	Miss
4	6	3	0	9	9 hinzugefügt	1 Kick	Miss
6	3	0	9	2	2 hinzugefügt	4 Kick	Miss
3	0	9	2	1	1 hinzugefügt	6 Kick	Miss

LRU:

				1	1 hinzugefügt	-	Miss
			1	5	5 hinzugefügt	-	Miss
		1	5	4	4 hinzugefügt	-	Miss
	1	5	4	6	6 hinzugefügt	-	Miss
	1	4	6	**5**	5 gefunden	-	**Hit**
1	4	6	5	3	3 hinzugefügt	-	Miss
4	6	5	3	0	0 hinzugefügt	1 Kick	Miss
6	5	3	0	9	9 hinzugefügt	4 Kick	Miss
5	3	0	9	2	2 hinzugefügt	6 Kick	Miss
3	0	9	2	1	1 hinzugefügt	5 Kick	Miss

1 Hit, 9 Misses = 10% Hitrate
LRU hat hier zufälligerweise gleichviele Cache Hits wie FIFO. Die einzige Änderung im Vergleich zu FIFO ist nach dem Hit, da LRU die Page 5 verwendet hat, und diese damit „recent" ist, wird diese wieder ganz vorne eingeordnet im Cache.

MRU:

				1	1 hinzugefügt	-	Miss
			1	5	5 hinzugefügt	-	Miss
		1	5	4	4 hinzugefügt	-	Miss
	1	5	4	6	6 hinzugefügt	-	Miss
	1	4	6	**5**	5 gefunden	-	**Hit**
5	1	4	6	3	3 hinzugefügt	-	Miss
1	4	6	3	0	0 hinzugefügt	5 Kick	Miss
4	6	3	0	9	9 hinzugefügt	1 Kick	Miss
6	3	0	9	2	2 hinzugefügt	4 Kick	Miss
3	0	9	2	1	1 hinzugefügt	6 Kick	Miss

MRU, also „Most-recent-use" agiert, fast genauso wie LRU in der Tabelle oben. Der kleine aber feine Unterschied ist, dass MRU die 5 nicht an den Anfang gesetzt hätte, sondern ganz ans Ende und somit beim übernächsten einlesen einer Page, die Page „5" gelöscht hätte.
1 Hit, 9 Misses = 10% Hitrate

Optimal:

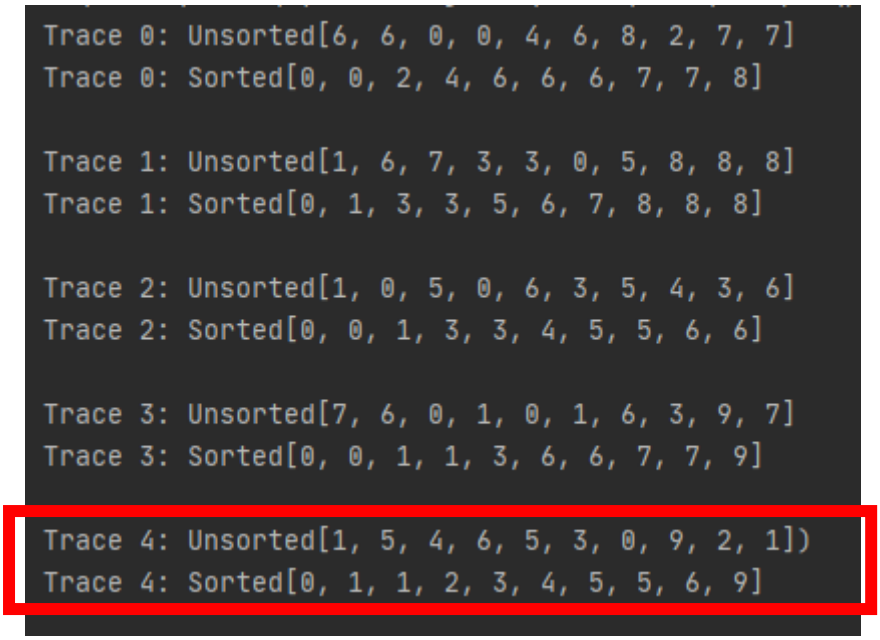
				1	1 hinzugefügt	-	Miss
			1	5	5 hinzugefügt	-	Miss
		1	5	4	4 hinzugefügt	-	Miss
	1	5	4	6	6 hinzugefügt	-	Miss
	1	**5**	4	6	5 gefunden	-	**Hit**
1	5	4	6	3	3 hinzugefügt	-	Miss
1	5	4	6	0	0 hinzugefügt	0 Kick	Miss
1	5	4	6	9	9 hinzugefügt	9 Kick	Miss
1	5	4	6	2	2 hinzugefügt	2 Kick	Miss
1	5	4	6	2	1 gefunden	2 Kick	**Hit**

2 Hit, 8 Misses = 20% Hitrate
Da die „Optimale Policy“ die Zukunft predicten kann, hebt sie sich die 1 auf, da sie weiß, dass die Page noch kommen wird. So kann sie noch im Vergleich zu den Anderen einen Cache Hit mehr gewinnen.

Random:

Die „Random Policy“ kann sowohl gut, als auch schlecht sein. Je nachdem welche Pages sie zufällig rauswirft, gibt es mehr oder weniger Cache Hits, aber maximal 2, denn die Optimal Policy hat bewiesen, dass 2 das Maximum ist, dass man mit unserer Trace erreichen kann.

22.4



Da wir bisher bereits mit der „örtlichen" Lokalität gearbeitet haben, wollen wir noch mehr die Temporale Lokalität verbessern, indem wir unsere Pagereihenfolge sortieren. Dadurch werden die Pages nacheinander drankommen und somit werden eventuell bei manchen Policys mehr Cache Hits logischerweise erzielt.

Wir haben diese temporale Lokalität mit der Pythonfunktion sort() gelöst. Diese sortiert ein Array nach der Größe.

LRU:

Mit unserer Neuanordnung performt LRU besser als noch davor.
2 Hit, 8 Misses = 20% Hitrate
Die Hitrate hat sich verdoppelt, da nun auch die eins einen Cache Hit erfährt.

				0	0 hinzugefügt	-	Miss
			0	1	1 hinzugefügt	-	Miss
			0	**1**	1 gefunden	-	**Hit**
		0	1	2	2 hinzugefügt	-	Miss
	0	1	2	3	3 hinzugefügt	-	Miss
0	1	2	3	4	4 hinzugefügt	-	Miss
1	2	3	4	5	5 hinzugefügt	0 Kick	Miss
1	2	3	4	**5**	5 gefunden	-	**Hit**
2	3	4	5	6	6 hinzugefügt	1 Kick	Miss
3	4	5	6	9	9 hinzugefügt	2 Kick	Miss

Random:

Random kann nur gleich gut oder schlechter werden als LRU, da LRU mit 2 Cache Hits bereits den bestmöglichen Ausgang für unser Beispiel hat. Im schlimmsten Falle, werden es 0 Hits und im Besten Falle 2 hits. Daher ist in diesem Beispiel LRU wesentlich zuverlässiger und besser als Random.

Clock:

Mit Clock pointer immer am neuesten Element startend (am rechtesten)
() = Use/Reference bit
_ = clock pointer

				0(1)	0 hinzugefügt	-	Miss
			0(1)	1(1)	1 hinzugefügt	-	Miss
			0(1)	**1(1)**	1 gefunden	-	**Hit**
		0(1)	1(1)	2(1)	2 hinzugefügt	-	Miss
	0(1)	1(1)	2(1)	3(1)	3 hinzugefügt	-	Miss
0(1)	1(1)	2(1)	3(1)	4(1)	4 hinzugefügt	-	Miss
0(0)	1(0)	2(0)	3(0)	5(1)	5 hinzugefügt	Kick 4	Miss
0(0)	1(0)	2(0)	3(0)	**5(1)**	5 gefunden	-	**Hit**
1(0)	2(0)	3(0)	5(0)	6(1)	6 hinzugefügt	Kick 0	Miss
2(0)	3(0)	5(0)	6(0)	9(1)	9 hinzugefügt	Kick 1	Miss

2 Hits, 8 Misses = 20% Hitrate
Bei clockbits > 2, kommt es drauf an wie das ganze implementiert ist. Wir gehen davon aus, dass damit mehr Reference bits dazu kommen, also nicht nur 0 oder 1. Denn wenn die 2 genauso behandelt wird wie eine 1 dann verhält es sich genau gleich. Dann würde die 2 auf 0 gesetzt werden usw.
Aber wenn man sagt, dass bei clockbits > 1, nur z.B. -1 gerechnet wird, dann würde man ein anderes verhalten bekommen. Dann würden nämlich Pages die mehr als 1-mal nochmal referenziert worden nicht so schnell rausfliegen, da diese dann einen höheren Clockbit hätten. Dies hätte dann wahrscheinlich zu Folge das bei Traces von vielen sich wiederholenden Zahlen die Hitrate steigt bzw. gleichbleibt. Bei unserem Beispiel ändert dies jedoch nichts, da wir keine Zahl haben die mehr als 2-mal vorkommt.

Bsp.
Mit Clock pointer immer am neuesten Element startend (am rechtesten)
() = Use/Reference bit
_ = clock pointer

				0(1)	0 hinzugefügt	-	Miss
			0(1)	1(1)	1 hinzugefügt	-	Miss
			0(1)	**1(1)**	1 gefunden	-	**Hit**
		0(1)	1(1)	2(1)	2 hinzugefügt	-	Miss
	0(1)	1(1)	2(1)	3(1)	3 hinzugefügt	-	Miss
0(1)	1(1)	2(1)	3(1)	4(1)	4 hinzugefügt	-	Miss
0(1)	**1(2)**	2(1)	3(1)	4(1)	1 gefunden	-	**Hit**
0(0)	1(1)	2(0)	3(0)	5(1)	5 hinzugefügt	Kick 4	Miss
1(1)	2(0)	3(0)	5(1)	6(1)	6 hinzugefügt	Kick 0	Miss
1(0)	9(1)	3(0)	5(1)	6(1)	9 hinzugefügt	Kick 2	Miss
1(1)	9(1)	3(0)	5(1)	6(1)	1 gefunden	-	**Hit**

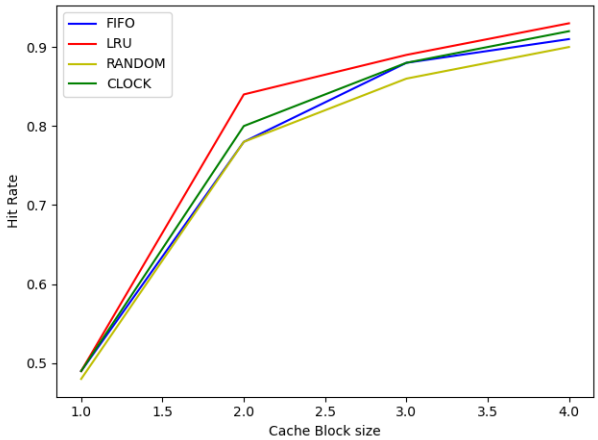
3 Hits, 8 Misses = 27% Hitrate

22.5

Wir haben das ganze im Container ausgeführt. Deswegen haben wir auch keinen Graphen für die Optimal & Unoptimal policy, da dies sehr lange gedauert hat. Der Container ist ein 32 Bit System, daher kann bekommen wir einen virtuellen Adressraum von 4GB also 2^32 Byte. Dies ergeben 32 Bit und mit „getconf PAGESIZE" bekamen wir 4 KB als Page Size raus. Daraus ergibt sich eine Virtual Page Number von 20 bit. Damit bleiben 12 Bit für den Offset übrig. Darum müssen wir dann wie im Buch erwähnt das ganze um 12 stellen nach rechts schifften. Alle neu übersetzten VPNs haben wir einer txt file untergebracht damit wir dies dann mit dem Simulator mit -f einfach einlesen können. Daraus hat sich der folgende Graph gebildet.
Der Aufruf war immer:

./paging-policy.py -f ./vpns.txt -p „(FIFO, LRU, RAND, CLOCK)" -C (1 – 4) -c

(natürlich immer nur ein Wert von den Klammern auf einmal)



Anhand des Graphs kann man sehen das wir etwas mehr als 4 GB an Cache brauchen damit man Hitraten von bis zu 100% erhält.