Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Ausarbeitung

Caches
Eine Einführung in Caches und ihre Verdrängungsalgorithmen

vorgelegt von

Robin Wannags

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang: Informatik
Matrikelnummer: 6948409

Betreuer: Kira Duwe

Hamburg, 10.03.2019

Contents

1 Einleitung

"Leer doch mal deinen Cache, dann funktioniert es wieder!"

Diesen Satz liest man oft in diversen Foren, als Lösungsvorschlag für unterschiedliche Probleme. Sei es, dass eine Internetseite nicht korrekt angezeigt wird oder in einem Online Spiel etwas nicht richtig funktioniert.
Doch was hat es mit diesem Cache auf sich und wieso hilft das Entleeren? Fragen wie diese möchte ich in dieser Arbeit beantworten, wobei ich zunächst auf das Thema 'Speicher' eingehen werde.
Computerspeicher ist einer der zentralen Punkte in einem modernen Computer. In ihm werden sowohl temporär, als auch dauerhaft Daten gespeichert, wodurch das Ausführen von Programmen ohne Speicher nicht möglich wäre. Das Problem an heutigen Computern ist allerdings, dass CPUs und GPUs immer schneller werden, Speicher hingegen kaum, was in einer stetig wachsenden Lücke zwischen diesen Komponenten resultiert.
In meiner Arbeit geht es darum, herauszuarbeiten, wie Caches in einem modernen Computer arbeiten und auf welche Herausforderungen sie stoßen.
Ich habe mich mit den verschiedenen Problemen beschäftigt, welche dabei auftreten könnten und Lösungen für diese vorgestellt, hauptsächlich in Hinblick auf Algorithmen zum Leeren des Cachespeichers.
Infolgedessen bin ich auf die Speicherhierarchie von modernen Systemen eingegangen, um diese in Bezug auf verschiedene Attribute miteinander zu vergleichen.
Ebenfalls habe ich den Unterschied zu virtuellem Cachespeicher herausgearbeitet und sinnvolle Einsatzmöglichkeiten für diesen aufgezeigt.
Um verschiedene Algorithmen zum Leeren von Caches vergleichen zu können, habe ich die Grundlagen von Cache-Hits und Cache-Misses erklärt, was ein optimaler Cache Algorithmus ist und wie man diesen nutzen kann, um Schwächen von anderen Algorithmen aufzuzeigen.
Diese Schwächen habe ich anhand von Workloads gezeigt, ebenso wie die Stärken von verschiedenen statischen Algorithmen und einem dynamischen Algorithmus.

2 Speicher Hierarchie und Caches

2.1 Speicher Hierarchie

Um zu verstehen, wozu Cache Algorithmen gebraucht werden, ist es wichtig, verschiedene moderne Speichermedien zu verstehen und ihre Unterschiede zu kennen.
Um verschiedene Speicher miteinander vergleichen zu können, wird Figur 2.1 benutzt. Diese gibt eine Speicherhierarchie an, die auf unterschiedlichen Eigenschaften und Preisen aufgebaut ist.

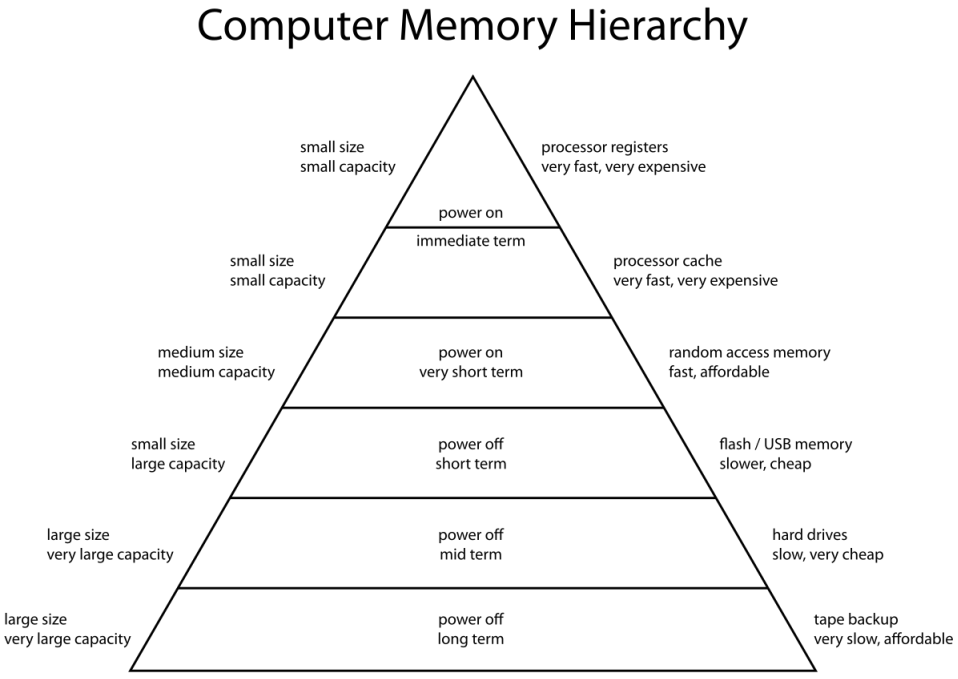

Figure 2.1: Speicher Hierarchie [6]

Anhand dieser Pyramide (Fig. 2.1) sieht man, in welchem Verhältnis Speicher in einem Computer vorliegt. Die Ausnahme bilden hierbei die Magnetbänder, da diese für den heutigen Gebrauch selten praktikabel sind. Sie werden dennoch mit aufgelistet, da vor allem Datenzentren auf ihnen große Mengen an Daten speichern können und das für einen geringen Preis.
Anzumerken ist, dass sich die nachfolgenden Werte um Durchschnittswerte, zum Zeitpunkt des Erstellens des Berichts (März 2019), handeln und es durchaus zu Abweichungen

kommen kann. Die Preise und Größen wurden der deutschen Amazon Seite entnommen.

Ganz unten in der Hierarchie befinden sich eben diese Magnetbänder. Ein durchschnittliches Magnetband fasst 6TB an Speicher, was etwa 4€ pro TB an Speicher sind. Ein weiterer Vorteil ist die lange Lagerzeit von 30 Jahren ([4] Seite 33), jedoch ist die Datenrate recht gering mit 280MB/s.
Einen Platz weiter oben sind HDD Festplatten mit einer durchschnittlichen Größe von 2TB bei etwa 30€/TB an Speicher. Die Lagerbarkeit ohne Strom ist mit bis zu 10 Jahren angegeben([4] Seite 33) und die Datenrate ist ebenfalls mit 200MB/s niedrig. Im Gegensatz zu Magnetbändern kann man jedoch wahlweise auf die Daten zugreifen und muss nicht vorerst das ganze Band durchlaufen, weswegen sie in modernen Computersystemen anzutreffen sind.
Direkt über den HDD Festplatten kommen die Flashspeicher, wie SSD Laufwerke und USB-Sticks. Eine SSD Festplatte ist durchschnittlich 500GB groß bei 160€/TB. Sie sind ebenfalls bis zu 10 Jahre ohne Strom lagerbar([4] Seite 33) und haben eine höhere Datenrate von durchschnittlich 600MB/s.
Nun folgen die flüchtigen Speicher. Diese haben allesamt keine Lagerzeit ohne Strom, da sie ihren Zustand nur mit Energie halten können.
An letzter Stelle des flüchtigen Speichers steht der RAM Speicher (auch Hauptspeicher im Computer genannt). Die durchschnittliche Größe eines RAM Riegels beträgt 8GB was einen Preis von 8500€/TB entspricht. Dafür steigt die Datenrate jedoch auch beträchtlich auf 3200MB/s.
Der zweitschnellste Speicher in einem Computersystem ist der Prozessorcache. Hier variieren die Werte sehr stark, aber ein durchschnittlicher Prozessor hat etwa eine Speichergröße von 9MB [7]. Einen Preis kann man leider nicht berechnen, da Cachespeicher nicht öffentlich verkauft wird und auch von einer Datenrate kann man nicht mehr sprechen. Jedoch hat der Cachespeicher eine Zugriffszeit von wenigen Nanosekunden.
Den schnellsten Speicher bilden die Register einer CPU. Auch hier können Größe und Preis nicht wirklich angegeben werden, aber die Zugriffszeit beträgt zwischen 3 und 4 Zugriffen pro Nanosekunde.
Wie man sehen kann, werden die Datenraten und die durchschnittlichen Größen von Speichern exponentiell kleiner, je weiter man in der Speicherhierarchie nach oben geht, der Preis wächst jedoch genauso schnell an.

2.2 Begriffserklärung: Cache

Für einen möglichst schnellen Computer wird also möglichst viel vom schnellsten Speicher benötigt, jedoch wäre dies beinahe unbezahlbar. Man muss also Kompromisse eingehen, einen Balanceakt zwischen Schnelligkeit und Preis.

Will beispielsweise der RAM-Speicher Informationen von einer HDD-Festplatte laden, so kann dieser die Daten mit 3200MB/s verarbeiten, die Festplatte kann die Daten jedoch

nur mit einer Datenrate von 200MB/s bereitstellen.
Dies führt zu einem sogenannten Bottleneck, der Computer wird ausgebremst.
Hier kommt Cachespeicher ins Spiel, denn dieser wird als Puffer genutzt, um die Geschwindigkeitsdefizite auszugleichen.
Beispielsweise sind Festplatten mit RAM Speicher ausgestattet, in dem oft genutzte Dateien liegen und somit schnell verarbeitet werden können. Dieser ist zwar oft nicht sonderlich groß, in etwa 256MiB, dies reicht jedoch oftmals schon aus, um ein starkes Ausbremsen des Systems zu verhindern.
Neben Festplattencache gibt es außerdem noch einen Prozessorcache, dieser gilt als Puffer zwischen RAM-Speicher und den Registern, sowie einen virtuellen Cache. Virtueller Cache ist kein dedizierter Cachespeicher, wie z.B. der Cachespeicher in einer Festplatte, sondern ein manuell angelegter Speicher in einer Festplatte, um lange Übertragungen von Datenbanken oder Webseiten zu verkürzen. Es handelt sich also um ganz normalen Festplattenspeicher, der vom System zur Speicherung von bestimmten Daten verwendet werden kann, denn selbst die 200MB/s von einer HDD-Festplatte sind meist schneller, als die Anbindung an eine Datenbank oder eine Webseite.
Der Nachteil an diesem Cachespeicher ist jedoch, dass er hin und wieder veraltete Daten bereitstellt und es zu Fehlern kommen kann.

3 Algorithmen zur Cacheverdrängung

Sofern nicht anders angegeben, bezieht sich dieses Kapitel größtenteils auf die Quelle [1]. Diese kann als Referenz für Behauptungen genutzt werden.

3.1 Cache-Algorithmen

Wie in 2.2 bereits beschrieben, will man für einen möglichst schnellen Rechner möglichst viel Speicher haben, der in der Speicherhierarchie weiter oben liegt. Dies ist jedoch nicht ohne immense Kosten möglich. Ein Programm, welches 100GB groß wäre, würde 850€ an RAM-Speicher kosten (Zahlen aus 2.1 entnommen), wenn man es komplett in diesem laden würde. Würde ein solches Programm direkt in den Prozessorcache geladen werden, dann wäre der Preis unbezahlbar, ebenso wäre es technisch nicht umsetzbar. Es mussten also andere Möglichkeiten erfunden werden, um das Ausbremsen des System zu verhindern.
An dieser Stelle kommen Cache-Algorithmen zum Einsatz.
Die Idee hinter den Cache-Algorithmen ist simpel, man nutzt den schnellen Speicher so effektiv wie möglich und löscht unwichtige Pages wieder aus diesem, während wichtige behalten werden, um ein Nachladen dieser Dateien zu verhindern. So spart man sich wertvolle Zeit, da die schnellen Speicher nicht mehr durch die langsamen ausgebremst werden würden.

3.2 Cache Hits und Misses

In der Theorie klingt dieser Ansatz vielversprechend, jedoch ist die Umsetzung schwierig. Das Problem ist, dass das System entscheiden müsste, welche Pages es am Längsten nicht mehr brauchen wird. Mit anderen Worten: Es muss in die Zukunft blicken und das ist nicht möglich.
Wird auf den Cache zugegriffen und die benötigte Page befindet sich noch in selbigem, nennt man dies einen Cache Hit.
Umgekehrt, wenn die benötigte Page sich nicht im Cachespeicher befindet, von einem Cache Miss.
Es gilt also die Anzahl der Cache Misses möglichst gering zu halten, denn ein Nachladen kostet wertvolle Zeit.

Ein Beispiel:
Angenommen die Zugriffszeit auf den Cache der Festplatte beträgt 100ns und die Zugriffszeit auf die Festplatte beträgt 10ms. Um zu berechnen wie gewichtig die Cache

Misses im Vergleich zu den Hits sind, berechnet man die Average Memory Access Time (AMAT).
Mit den oben genannten Werten ergibt sich folgende Rechnung:

$$90\%hit = 100ns + 0.1 * 10ms = 1.0001ms$$

Bei einer 90% Hitrate ruft man also 0.1 mal den langsameren Speicher auf. Es ist ebenfalls zu beachten, dass man immer vorher den Cache prüft, ob die gesuchte Datei sich bereits in diesem befindet.

$$95\%hit = 100ns + 0.05 * 10ms = 0.5001ms$$

Wenn man die Hitrate um 5% erhöht wird die AMAT halbiert.

$$99.9\%hit = 100ns + 0.001 * 10ms = 0.0101ms$$

Und je näher man sich an die 100% herantastet, desto näher kommt man der Zugriffszeit des Caches. Diese kann man jedoch nie unterschreiten, da man den Cache jedes mal aufrufen muss.

3.3 Optimaler Cache-Algorithmus

Mithilfe von Cache Hits und Misses kann man nun einen optimalen Cache-Algorithmus herausfinden. Doch was genau ist ein optimaler Algorithmus überhaupt?
Wie oben beschrieben, will man möglichst wenig Cache Misses haben, jedoch ist es in den meisten Fällen unmöglich eine 100% Hitrate zu bekommen, denn irgendwann muss man eine Page im Cache austauschen. Dies geschieht, wenn der Cache voll ist, eine neue Page jedoch unbedingt benötigt wird.
Optimal wäre es, wenn man dann eine Datei austauscht, die nicht wieder benötigt wird. Dies ist in der Praxis jedoch selten der Fall, deswegen wird die Page gelöscht, die für die längste Zeit, im Vergleich zu allen anderen Dateien, nicht wieder benötigt wird.

An einem Beispiel (Fig. 3.1) wird dies schnell deutlich:
Gegeben sei ein Cache, der drei Pages fasst, das System muss allerdings vier Pages benutzen.
In der ersten Zeile wird die Page 0 angefordert. Sie ist jedoch nicht im Cache, da der Cache anfangs leer ist. Somit gibt es bereits einen initialen Miss. Es muss jedoch keine Page ausgetauscht werden und sobald sie geladen ist, befindet sie sich im Cache.
In der zweiten Zeile passiert etwas ähnliches, denn auch Page 1 ist noch nicht im Cache und man bekommt einen weiteren Miss. Da der Cache noch nicht voll ist, kann sie problemlos geladen werden.
Ebenso bei Zeile 3 mit Page 2.
Als nächstes wird Page 0 benötigt, diese befindet sich bereits im Cache, weshalb ein erneutes Laden überflüssig wird. Die nächste Page 1 ist ebenfalls schon im Cache.

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

Figure 3.1: [1] Seite 3 - optimaler Algorithmus

Als sechstes wird nun Page 3 benötigt. Da sich im Cache nur Page 0, 1 und 2 befinden, ergibt sich hier ein Miss. Nun geht es darum, eine Page zu verwerfen, um Nummer 3 in den Cache laden zu können. Ein Blick in die Zukunft verrät, dass als nächstes die 0 benötigt wird, danach wieder die 3 und danach die 1. Somit wird Page 2 als letztes wieder gebraucht und vorerst aus dem Cache gelöscht.
Anschließend kommen wieder drei Hits mit den bereits vorausgesehenen Pages 0, 3 und 1 und nun wird wieder Page 2 benötigt.
Das System benötigt danach nur noch Page 1, deswegen kann die Wahl zwischen 3 und 0 zufällig getroffen werden.

Dies wäre der optimale Algorithmus für diesen Arbeitsprozess. Leider ist ein Blick in die Zukunft, wie bereits erwähnt, nicht möglich. Dieser optimale Algorithmus kann jedoch verwendet werden, um andere Algorithmen bewerten zu können. Bei sechs von elf getroffenen Pages ergibt sich eine Hitrate von 54,5%.

3.4 FIFO-Algorithmus

Der erste Verdrängungsalgorithmus basiert auf dem FIFO (First In First Out) Prinzip. Die Page, die zuerst geladen wurde, muss auch als erstes wieder verworfen werden.
Das selbe Beispiel (Fig. 3.2) wie beim optimalen Algorithmus, sieht mit dem FIFO Algorithmus wie folgt aus:

Zuerst ergeben sich wieder drei initiale Cache Misses, gefolgt von den beiden Cache Hits. Nun benötigt das System Page 3. Da Page 0 zuerst in den Cache geladen wurde, muss sie auch zuerst verworfen werden. Somit sind danach Page 1, 2 und 3 im Cache.

Access	Hit/Miss?	Evict	Resulting Cache State	
0	Miss		First-in→	0
1	Miss		First-in→	0, 1
2	Miss		First-in→	0, 1, 2
0	Hit		First-in→	0, 1, 2
1	Hit		First-in→	0, 1, 2
3	Miss	0	First-in→	1, 2, 3
0	Miss	1	First-in→	2, 3, 0
3	Hit		First-in→	2, 3, 0
1	Miss	2	First-in→	3, 0, 1
2	Miss	3	First-in→	0, 1, 2
1	Hit		First-in→	0, 1, 2

Figure 3.2: [1] Seite 5 - FIFO Algorithmus

Sofort danach wird allerdings wieder Page 0 benötigt. Diesmal wird Page 1 verworfen, da diese nun die am längsten im Cache gespeicherte Page ist. Der Cache besteht nun aus den Pages 2, 3 und 0 (in dieser Reihenfolge geladen).
Anschließend wird nun Page 3 gebraucht, die sich im Cache befindet und anschließend wieder Page 1. Diese befindet sich aber nicht mehr im Cache. Diesmal wird Page 2 verworfen. Der Cache beinhaltet nun 3, 0 und 1.
2 wird jedoch auch sofort wieder gebraucht, wodurch 3 verworfen wird. Als letztes wird noch einmal Page 1 benötigt, die sich noch im Cache befindet.
Mit dem FIFO Algorithmus ergibt sich eine Hitrate von 36,36%. Im Vergleich zu den 54,5% des optimalen Algorithmus ist dies nicht sonderlich viel.
Das Problem an FIFO ist, dass er nicht intelligent ist. Er passt sich nicht dem Code an, sondern läuft nach einem festen Schema ab.

3.5 Random-Algorithmus

Als nächstes wird der Random Verdrängungsalgorithmus getestet. Sofern nötig, werden Pages hier willkürlich verworfen.
Für eine hinreichende Vergleichsgrundlage, wird auch hier dasselbe Beispiel, wie bei den vorangegangenen Algorithmen verwendet und verglichen (Fig. 3.3):

Zum Start gibt es wieder die drei Misses und die zwei Hits.
Nun wird Page 3 benötigt. Da nicht ausreichend Platz im Cache zur Verfügung steht, entfernt man willkürlich eine Page. In diesem Fall wurde sich willkürlich für Page 0 entschieden.
Im nächsten Schritt wird jedoch klar, dass dies eine schlechte Wahl war, da Page 0

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	0	1, 2, 3
0	Miss	1	2, 3, 0
3	Hit		2, 3, 0
1	Miss	3	2, 0, 1
2	Hit		2, 0, 1
1	Hit		2, 0, 1

Figure 3.3: [1] Seite 6 - Random Algorithmus

gleich wieder benötigt wird. Dieses mal wird Page 1 entfernt, ebenfalls völlig zufällig ausgewählt.
Anschließend wird die 3 benötigt, welche schon im Cache ist und wieder die eben verworfene 1. Diesmal wird die 3 verworfen.
Die letzten beiden Aufrufe sind beides Hits.
Somit ergibt sich eine Hitrate von 45,45%. Deutlich besser als FIFO, jedoch noch immer nicht optimal.
Das Problem an diesem Algorithmus ist, dass seine Ergebnisse völlig zufällig sind. Sie könnten optimal sein, sie könnten aber auch katastrophal sein, denn auch dieser Algorithmus ist nicht intelligent.

3.6 LRU-Algorithmus

Der letzte vorgestellte Algorithmus heißt LRU (Least Recently Used). Er überprüft, welche Pages am längsten nicht genutzt wurden und entfernt diese bei Bedarf aus dem Cache. Somit passt er sich dem Code an und agiert damit "intelligent".
Es wird wieder dasselbe Beispiel genutzt(Fig. 3.4):

Anfangs ergeben sich wieder drei Misses um den Speicher zu füllen und zwei Hits, da sich diese Pages bereits im Cache befinden. Nun muss Page 3 geladen werden. Da sie sich nicht im Cache befindet, ist dies ein Miss und es muss nun eine Page aus dem Cache entfernt werden. Hierzu schaut man sich die zuletzt verwendeten Pages an. Zuvor wurde Page 1 benutzt und davor Page 0. Somit wurde Page 2 am längsten nicht benutzt und wird aus dem Cache entfernt.
Es wird nun Page 0 benötigt, gefolgt von 3 und 1, die allesamt bereits im Cache sind.

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		LRU→ 0
1	Miss		LRU→ 0, 1
2	Miss		LRU→ 0, 1, 2
0	Hit		LRU→ 1, 2, 0
1	Hit		LRU→ 2, 0, 1
3	Miss	2	LRU→ 0, 1, 3
0	Hit		LRU→ 1, 3, 0
3	Hit		LRU→ 1, 0, 3
1	Hit		LRU→ 0, 3, 1
2	Miss	0	LRU→ 3, 1, 2
1	Hit		LRU→ 3, 2, 1

Figure 3.4: [1] Seite 7 - LRU Algorithmus

Als vorletzte Aktion wird wieder Page 2 benötigt. Diese befindet sich nicht im Cache, wodurch es einen weiteren Miss gibt und eine Page ersetzt werden muss.
Erneut schaut man auf die letzten Aktionen und sieht, dass erst Page 1 genutzt wurde und davor Page 3. Somit muss Page 0 aus dem Cache entfernt werden, um Page 2 laden zu können. Zuletzt gibt es noch einen Hit.

LRU kommt somit auf eine Hitrate von 54,54% und hat mit diesem Beispiel ein optimales Ergebnis erzielen können.
LRU hat sich von den drei Beispielen also am besten geschlagen, da es die Vergangenheit der Zugriffe in Betracht zieht und anhand dieser entscheidet.
Natürlich ist er dafür auch wesentlich schwieriger zu implementieren als FIFO und Random. Nun gilt es herauszufinden, ob dies nur ein glücklich gewähltes Beispiel war oder ob LRU wirklich der perfekte Algorithmus ist.

4 Workloads

Durch Workloads kann das Verhalten von Algorithmen beobachtet werden. Ein Workload ist eine bestimmte Aufgabe, die durch einen Algorithmus X mal ausgeführt wird.

4.1 No-Locality Workload

Hier wird zuerst der No-Locality (Fig. 4.1) Workload vorgestellt.
In diesem Workload gibt es 100 verschiedene Pages, die 10000 mal abgefragt werden. Im Falle von No-Locality sind neue Pages immer zufällig gewählt. In Fig. 4.1 wird die Hit Rate durch die Y-Achse bestimmt, die X-Achse bezeichnet die Größe des Cachespeichers. Bei einer Größe von 100 ist die Hitrate natürlich 100%, wenn man nur 100 Pages hat (wenn man von initialen Misses absieht).

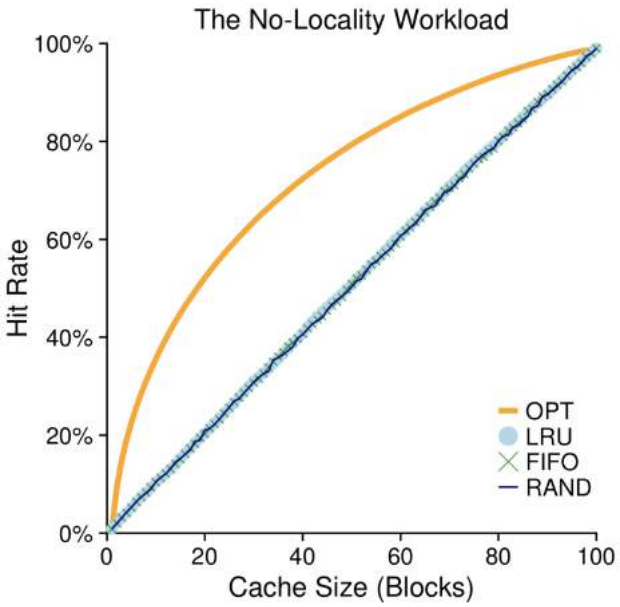

Figure 4.1: [1] Seite 9 - No-Locality Workload

Wenig überraschend verhalten sich in diesem Workload alle Algorithmen gleich. Das liegt daran, dass Random (RAND) und FIFO nicht intelligent vorgehen und LRU eine

Historie braucht, um richtig funktionieren zu können. Lediglich der optimale Algorithmus (OPT) erzielt durchgehend höhere Hit Raten, was darauf schließen lässt, dass keiner der drei Algorithmen optimal arbeitet.

4.2 80-20 Workload

In diesem Workload (Fig. 4.2) werden wieder 100 verschiedene Pages genutzt und diese 10000 mal abgefragt. Diesmal werden die Pages aber nicht zufällig gewählt, sondern es werden zu 80% immer die gleichen 20% der Pages genutzt.

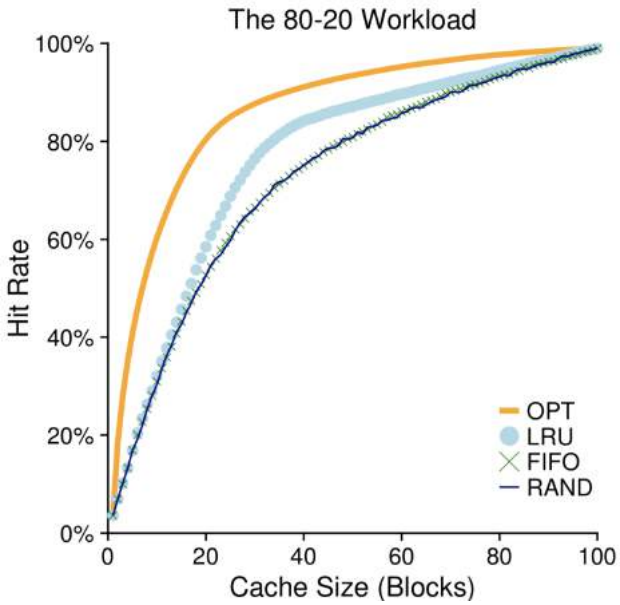

Figure 4.2: [1] Seite 10 -80-20 Workload

In der Grafik Fig. 4.2 sieht man deutlich, dass LRU hier besser arbeitet als FIFO und Random(RAND), denn da oftmals die selben Pages benutzt werden, kann LRU die ungenutzten leicht ausmachen und verwerfen, während die wichtigen immer im Cache bleiben. Optimal ist der Algorithmus trotzdem nicht, jedoch ist ein optimaler Algorithmus(OPT) in der Größenordnung auch eher Wunschdenken.
Überraschenderweise agieren auch Random und FIFO in diesem Workload wesentlich besser, obwohl sie eigentlich gar keine Strategie verfolgen. Das liegt daran, dass oftmals die gleichen 20% der Pages genutzt werden. Random hat so eine viel höhere Chance die "schlechten" Pages zu verwerfen.
Genau das gleiche Prinzip greift ebenfalls bei FIFO, auch wenn man es auf den ersten Blick vielleicht nicht bemerkt.

4.3 Looping sequential Workload

Der letzte Workload, der hier vorgestellt wird, nutzt diesmal nur 50 Pages, die aber ebenfalls 10000 mal abgefragt werden. Looping sequential ruft diese 50 Pages der Reihe nach auf und beginnt nach der letzten von vorn. Das bis die 10000 Wiederholungen durchlaufen wurden.

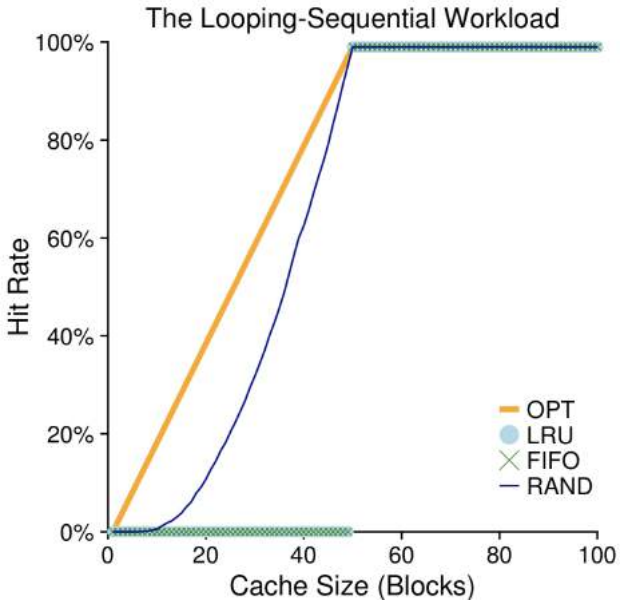

Figure 4.3: [1] Seite 11 - Looping Sequential Workload

Wie man in Fig. 4.3 sehen kann, sind LRU und FIFO komplett unbrauchbar, bis der Cache 50 Pages fassen kann. Danach funktionieren sie optimal. Tatsächlich unterscheiden sich die beiden Algorithmen in diesem Workload kaum, denn sie entfernen immer die ältesten Pages aus dem Cache und funktionieren somit ziemlich gleich. Diese ältesten Pages werden jedoch wieder früher gebraucht als die neuesten Pages, ohne Ausnahme. Somit hat man dauerhaft nur Cache Misses.
Random(RAND) kann diesen Workload noch relativ gut bearbeiten, aber auch hier reicht die Spanne von optimal(OPT) bis völlig unbrauchbar. In der Regel wird er aber besser funktionieren als FIFO und LRU.

4.4 Dynamische Algorithmen

Dynamische Algorithmen sind oftmals Verbindungen aus statischen Algorithmen. Beispielsweise ist der ARC (Adaptive Replacement Cache) ein Balanceakt zwischen LRU und MRU (Most Recently Used)(vgl. [5] Abschnitt: Adaptive replacement cache). Most Recently Used würde in dem Looping Sequential Workload hervorragend abschneiden, da die neuesten Pages gleich wieder verworfen werden.

ARC würde in diesem Fall ein Muster erkennen und für diesen Loop den Algorithmus auf MRU umstellen. Somit würden die ersten paar Iterationen in einer Loop zwar nicht gut funktionieren, der Algorithmus würde sich jedoch anpassen und dann ein hervorragendes Ergebnis abliefern.
Dynamische Algorithmen sind jedoch wesentlich komplexer zu implementieren als statische Algorithmen.

5 Zusammenfassung

Speichermedien werden nur langsam schneller, während die Prozessoren in den letzten Jahren immer schneller wurden.
Auch preislich gesehen ergibt es Sinn, nicht immer nur den schnellsten Speicher zu benutzen.
Caches sind eine gute Lösung, um diese Lücke in Geschwindigkeiten zu schließen. Doch Caches sind recht klein und müssen deswegen oft geleert werden. In der heutigen Zeit sind Algorithmen zum Verdrängen von überflüssigen Pages im Cache unumgänglich, um Systeme schneller zu machen.
Doch diese Algorithmen haben alle ihre speziellen Nischen, es gibt (noch) keinen optimalen Algorithmus in der Praxis. Selbst vielversprechende Algorithmen wie LRU werden in gewissen Arbeitsgebieten völlig unbrauchbar und ein Arbeiten ohne Cache wäre sogar schneller.
Jedoch können dynamische Algorithmen genau hier ansetzen und diese Extremfälle abschwächen, auch wenn ihre Implementation eine große Herausforderung ist.

Bibliography

[1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Beyond Physical Memory: Policies* March 2015

[2] Qi Zhu, Ying Qiao. *A Survey on Computer System Memory Management and Optimization Techniques* 2012

[3] Glass, G. and Cao, P. *Adaptive page replacement based on memory reference behavior* 1997

[4] Jakob Lüttgau, Michael Kuhn, Kira Duwe, Yevhen Alforov, Eugen Betke, Julian Kunkel, Thomas Ludwig. *Survey of Storage Systems for High-Performance Computing* 2018

[5] Wikipedia based on: Nimrod Megiddo and Dharmendra S. Modha. *ARC: A Self-Tuning, Low Overhead Replacement Cache* 2003

[6] Bildquelle: de.wikipedia.org/wiki/Speicherhierarchie

[7] https://www.amazon.de/Intel-Core-i5-8600K-60GHz-Boxed/dp/B0759FKH8K