Guten Morgen meine Damen und Herren, es geht weiter mit der Vorlesung Rechnerstrukturen. Wir haben das letzte Mal das Kapitel Datenpfad beendet und wir wollen uns heute ein bisschen die Speicherhierarchie anschauen, um das thema cash um dieses thema cash genauer zu verstehen ist natürlich gut wenn sie eine grundahnung haben von der von neumann architektur wie wir sie ja schon in der grundlagen der rechner architektur besprochen haben und sie sollten auch so eine grobe ahnung haben wie so ein speicherzugriff vom prozessor aus abläuft worum geht es in dem kapitel wir möchten ihnen ein richtig gutes ein Cache in der Architektur eingebettet ist, warum ich denn überhaupt Caches brauche und warum die Speicherhierarchie, die ich heutzutage finde, so komplex ist. Dafür wollen wir uns erstmal eine kleine Motivation holen, uns ein System anschauen, wie so ein Cache integriert ist und dann wollen wir uns auch mal anschauen, warum es überhaupt arbeiten es geht also um das prinzip der lokalität von daten die es uns eben deutlich dieses prinzip erleichtert es deutlich mit caches zu arbeiten dann speicher hierarchie und kosten ja so ein cache ist klein und schnell aber teuer deshalb muss man sich natürlich darüber unterhalten wie baue ich die hierarchie auf wie viel speicherplatz gebe ich in den cache wie viel speicherplatz gebe ich in den hauptspeicher und das wird letztendlich auch über den kostenfaktor dann habe ich natürlich immer das problem des mapping das heißt welche blöcke lagere ich in meinem cache ein welche blöcke verdränge ich wieder aus dem cache dann suche ist natürlich ganz wichtig einige beispiele und natürlich ist keine lösung ohne probleme das heißt wir werden uns genauer die erfolge von ersetzungs und schreibstrategien anschauen dann wenn ich platzierte und wenn wo ich die daten hin schreibt außerdem wie platziert ich meinen cash also was kommt on chip was kommt auf chip also auf board und insbesondere bei multiprozessor systemen ist die cash kohärenz natürlich ein richtig großes problem da gibt es ein protokoll dafür das mese protokoll das wenn wir aber voraussichtlich dann als in der nächsten vorlesung besprechen und dann klar rechner strukturen habe ich jetzt immer Das machen wir dann gegen Ende dieses Kapitels. Okay, hier nochmal das Bild von unserem Datenpfad. Und wo finden wir jetzt in diesem Datenpfad eigentlich Caches? Auf den ersten Blick ist klar, finden wir Caches hier in unserem Instruktions-Cache. Aus dem Instruktions-Cache holen wir unsere Instruktionen, laden die Instruktionen. Und hier hinten haben wir unseren Daten-Cache. speichere ich Daten oder hole mir Daten aus dem Speicher. Wenn man so will, dann haben wir ja hier unser Register-File und dieses Register-File ist eigentlich, wenn man so will, Cache Level 0, das heißt, das ist ja am nächsten an der ALU dran und hier liegen also die Daten, die direkt verarbeitet werden und wir haben ja schon beim Register-File gelernt, dass es natürlich toll ist, dass das Register-File am liebsten sehr sehr groß haben möchte auf der anderen seite habe ich natürlich auch probleme wenn ich das sehr groß mache weil zum beispiel wenn ich ein prozesswechsel habe muss ich ja register retten und je mehr register ich habe desto größer ist der aufwand und so ähnlich verhält es sich auch mit dem cash es gibt für alles eine richtig gute größe und es gibt was was zu klein ist und es gibt was was zu groß ist und was eben eine gute größe ist das wollen wir schauen, wie man das ermitteln kann unter Umständen. Gut, das ist jetzt hier ein Blick auf eine real

existierende Architektur. Das ist eine Fallstudie zum IA64. Das war also Ende der 90er Jahre, als die X86-Architektur schon ein bisschen an die Leistungsgrenzen gelangt ist, hat man sich natürlich im Hause Intel überlegt, ja wie kann es denn weitergehen in Zukunft? Bleibe ich bei Zisk-Architekturen oder gehe ich noch einen Schritt weiter geht so very large instruction world architekturen ja was ist denn die die architektur der zukunft und intel gemeinsam mit julia packard haben eigentlich große hoffnung gehabt dass der schritt in richtung very large instruction world dass das der richtige schritt ist ich meine die geschichte hat gezeigt dass zumindest kommerziell dieser schritt nicht so ganz erfolgreich war aber Wir haben hier oben beginnen unseren Instruction Cache, dann haben wir so eine Instruction Queue, die Architektur Very Large Instruction Word bedeutet ja, dass ich mehrere Maschinenbefehle gleichzeitig in Bearbeitung habe, also hier bis zu 6 Instruktionen habe ich gleichzeitig in Bearbeitung und die werden dann natürlich entsprechend auf die Register gemappt, die mir zur Verfügung stehen und damit überhaupt 6 Instruktionen gleichzeitig laufen können, brauche ich natürlich eine Vielzahl von parallelen Units zur Verarbeitung. Das heißt, ich habe hier drei Branch Units, ich habe vier Integer Allos, ich habe zwei Floating Point Units, die eben alle parallel liegen und gleichzeitig bedient werden können. So, der Versuch ist natürlich hier immer, sechs Instruktionen gleichzeitig am Laufen zu halten. Ja, da brauche ich natürlich eine gewisse Parallelität dafür. bei der einfachen pipeline war die wir im datenpfad besprochen haben das ist jetzt hier natürlich beliebig kompliziert das ganze und dann habe ich natürlich noch mal register setze hier 128 integer register 128 floating point register und die sind wiederum verbunden mit dem data cache cache on chip des weiteren haben wir noch einen dual ported l2 cache ja hier stehen dann instruktionen und befehle drin also sowohl der instruktions cache speist sich aus diesem l2 cache als auch der data cache das ist alles on chip und off chip ist dann der l3 cache und system und buskontrolle heutzutage gibt es durchaus systeme wo auch der l3 cache noch on chip ist zum großen Hauptspeicher über den Systembus. Was mir ganz gut gefällt an diesem Schaubild hier, wenn Sie sich jetzt nochmal die Phasen vergegenwärtigen, in denen wir unseren Befehlszyklus aufgeteilt haben, also in Instruction Fetch, Instruction Decode, Execute, Writeback und Memory. Also dieser Ablauf hier, der spiegelt sich natürlich ganz deutlich in dieser Architektur wieder, wenn ich von oben nach unten schaue. also zum Beispiel spekulative Ausführung von Instruktionen, die ich zwangsläufig brauche, um die Parallelität zu nutzen. Und dafür braucht es noch ein paar Tabellen, also so eine Advanced Load Address Table, wo ich vielleicht spekulativ Daten holen kann. Dann gibt es Data Translation Look-Aside Buffer, die ich dazu brauche, also DTLB steht hier. Und Instruction Translation Look-Aside Buffer, auch das steht hier. Also das sind alle Sachen, die brauche ich für die spekulative Programmausführung. Und das würde jetzt einfach in dieser Vorlesung ein bisschen zu weit führen, das komplett auszurollen. Und wir haben deshalb in unserer Vorlesung Grundlagenbetriebssysteme ein Kapitel, das sich eben mit solchen Mechanismen beschäftigt. Hier nochmal ganz kurz die groben Kenntaten vom Intel IA64, der auch unter der Bezeichnung Ethanium läuft. instruktionen die wir holen wollen also sechs fetches gleichzeitig die ausgeführt werden dann

haben wir jeweils 128 register für integer und floating point wir haben insgesamt neun execution units die zur ausführung bereit stehen und die ganze maschine kann auch im simt betrieb fahren das heißt single instruction multiple data also ich kann einen befehl gleichzeitig auf mehreren daten hier geschrieben, ein Schritt in die falsche Richtung, weil eben die Geschichte gezeigt hat, also das ist diese Entwicklung hier Ende der 90er Jahre statt, da war noch der Richtungsstreit zwischen CISC und RISC und VLW, dieser Streit ist kommerziell entschieden also die RISC Architekturen haben da eindeutig gewonnen und wenn Sie sich eben moderne Prozessoren anschauen, dann haben die alle RISC Architektur und den Datenpfad, den wir Risk Architektur. Okay, so, wie können oder warum können uns Caches jetzt helfen, schneller auf Daten zuzugreifen? Warum brauche ich erstmal überhaupt Caches? Ich brauche so eine komplexe Speicher-Hierarchie, weil die Prozessoren schneller schnell geworden sind, als die Speicher. Das heißt, der Geschwindigkeitsgewinn, den die Prozessoren erzielt haben über die letzten 30, 40 Jahre, ist um vielfaches höher als der Geschwindigkeitsgewinn, den ich bei Speichern erzielt habe. Dementsprechend ist es immer schwieriger, meine Prozessoren ausreichend schnell mit Daten zu versorgen und der normale Hauptspeicher ist einfach zu langsam. Also der reicht mir nicht aus und dann können wir sagen, na gut, warum mache ich denn noch DRAM als normalen Hauptspeicher, warum mache ich nicht den ganzen Speicher mit SRAM, also mit schnellem Speicher, richtig schnellem Speicher, und da kommt natürlich dann wieder das liebe Geld ins Spiel, das heißt so einen Gigabyte großen oder inzwischen fast Terabyte großen Hauptspeicher mit SRAM aufzubauen, das ist dann noch ein bisschen teuer und das lohnt sich für eine normale Rechnerarchitektur nicht, so ein Rechner muss ja auch bezahlbar bleiben. vom Registerpfeil bis zum Hauptspeicher auf. Warum kann ich das überhaupt? Warum können diese kleinen Speicher sinnvoll arbeiten? Das ist jetzt die Beobachtung 1, die ich da habe. Wenn ich meine Programme und das Speicherverhalten meiner Programme beobachte durch Simulationen oder durch Messungen, dann werde ich feststellen, dass sich Programme oft sehr lokal in Zeit und Raum verhalten. Was heißt das jetzt? ein Datum im Programm verwende, auf das eben gerade zugegriffen wurde, dann ist die Wahrscheinlichkeit groß, dass in naher Zukunft wieder auf das gleiche Datenelement zugegriffen wird. Das ist also zeitliche Lokalität. Räumlich ist so, denken Sie, wie Sie Ihre Datenstrukturen speichern, also zum Beispiel Arrays und dann ist es natürlich so, dass auf den Nachbarn eines Datenelements zugegriffen wird. Und zwar bezieht sich das auf Daten und auf Instruktionen. Bei den Instruktionen ist es auch klar, wenn ich jetzt so ein sequentielles Programm habe, das ich von oben nach unten lese, dann kommt ein Befehl nach dem anderen und die Befehle stehen in aufeinanderfolgenden Speicherplätzen. Das heißt, wenn ich mir dann aus dem Speicher einen ganzen Block in den Cache lade, dann ist natürlich die Wahrscheinlichkeit groß, dass die nächste Instruktion, die da kommt, dann schon im Cache drin liegt. Das Ganze wird natürlich durch Loops. Im Idealfall liegt der ganze Loop im Cache. Also zum Beispiel hier haben wir so einen Loop, der beginnt hier und hier ist das Ende des Loops, das springt dann wieder hierher zurück. Und wenn also diese 1, 2, 3, 4 Instruktionen, wenn die alle 4 im Cache liegen und der Loop läuft länger, dann brauche ich also eine

längere Zeit lang nicht auf den Hauptspeicher zugreifen, vorladen und dann eben hier im cache bleiben bei daten sieht es ähnlich aus wenn wir eben so eine array struktur haben dann ist die wahrscheinlichkeit groß dass wenn wir so ein array durchlaufen dass eben die daten lokal sind und hier ist es noch mal für dieses kleine assembler programm aufgezeichnet also wir haben store wir haben move increment addition decrement branch also mit dicken sprung der zurückgeht zum loop wenn denn die bedingungen erfüllt ist das eine Laufschleife, dann haben wir noch einen Move und noch einen Jump. Und wir sehen hier, dass eben innerhalb des Loops ständig auf Register B und Register C zugegriffen wird. Das heißt, die sind natürlich sowohl zeitlich als auch örtlich, also räumlich lokal hier. weil eben davon auszugehen ist, dass diese Befehle hintereinander ausgeführt werden. Aber sobald ich hier in diesem Loop bin, habe ich beides, zeitliche und räumliche Lokalität. Gut, das stimmt uns natürlich jetzt ein bisschen optimistisch. Wenn es tatsächlich der Fall ist, dann lade ich so ein Stück in meinen Cache und dann brauche ich eine längere Zeit nicht auf den Hauptspeicher zugreifen. dafür dass ich Caches brauche und nicht alles mit schnellen Hauptspeicher mache ist einfach die Beobachtung 2, dass richtig schneller Speicher eben auch richtig teuer ist. Andersherum ausgedrückt kann ich sagen je langsamer der Speicher ist desto größer kann er natürlich sein und wir unterscheiden praktisch in der normalen Rechner Architektur so drei Hauptstufen das eine bei der CPU sehr nah an sind die register die nächste stufe ist der cache speicher also deutlich schneller nicht ganz so großer speicher für daten und instruktionen und dann hier oben kommt der hauptspeicher so und hier ist mal eine größenrelation die heutzutage natürlich ein bisschen sich verschoben hat sagen wir mal insgesamt um den faktor 1000 vielleicht wie das halt in der informatik so ist wenn ich nicht aktualisieren dann ist alles um faktor 1000 schneller größer billiger nehmen sie es wie sie wollen aber einfach nur mal um die relation zu zeigen ja also die register das sind ein paar bytes die ich habe also 500 bytes das hat sich jetzt nicht allzu groß verändert deutlich verändert hat sich natürlich die cache größen ja damals waren 64 kilobyte heute ist man also keine ahnung eben heute auch nicht mehr ein gigabyte groß sondern ist bis zu einem terabyte groß und die festplatten die ich da an oder die ja die solid state disks die sind auch mehrere terabyte groß inzwischen geschwindigkeit die absoluten zahlen sind auch nicht so wichtig sondern halt die grobe relation die auch heute im wesentlichen noch gilt also die register sind natürlich noch mal Beispiel hier und dann der Cache ist eben um den Faktor 100 schneller als der Zugriff auf den Hauptspeicher und das ist dann nochmal deutlich schneller als der Zugriff auf die Festplatte. Diese ganzen Zahlen befinden sich natürlich immer in Bewegung, das heißt der Hauptspeicher ist heutzutage noch ein bisschen schneller geworden. zwischen festplatte und hauptspeicher nicht und auch preislich zwischen festplatten hauptspeicher nicht noch geschwindigkeitsmäßig also das heißt da wird vielleicht noch eine hierarchie stufe in zukunft eingezogen das also vor der festplatte noch so ein in vram nicht das wird sich zeigen aber generell gilt natürlich der trend ja also je näher ich an der cpu bin desto schneller ist der speicher und desto teurer ist der speicher ja hier einfach noch mal die die preisentwicklung dargestellt hat im Jahr 1988. Da gab es also noch Lochkarten, gibt es heutzutage natürlich nicht mehr.

Damals hat man auch gedacht, optischer Speicher ist die Lösung aller Probleme. Backup war auf Magnetbändern, dann gab es eben Festplatten und dann gab es noch ein paar exotische Technologien, also Bubble Storage und dann gab es natürlich den Hauptspeicher und dann den schnellen Speicher, die hierarchie und die entsprechenden zugriffszeiten drauf also hier haben wir zugriffszeiten aufgezeichnet klar je weiter oben desto langsamer und hier haben wir den preis in dollar pro bit aufgezeichnet und da ist natürlich klar je weiter rechts desto teurer sind wir ja das ist ein negative exponent hier gut ich habe dann mal letztes jahr mal nachgeschaut wie denn die preise heutzutage sind also wenn man eine SSD schaut, dann sind wir ungefähr bei 10 Cent pro Gigabyte, das heißt bei 1,25 mal 10 hoch minus 11 Dollar pro Bit, das heißt wir sind hier von 1988 bis 2019, also 30 Jahre später, sind wir vom Faktor 10 hoch minus 2 oder 10 hoch minus 3 auf den Faktor 10 hoch große veränderung von 10 hoch minus 3 nach 10 hoch minus 11 ist eine veränderung um 10 hoch 8 ungefähr also 10 hoch 6 ist die million das heißt es sind mehr 100 millionen fach billiger ist das ganze was das die festplatten betrifft und was den dieren betrifft das sind wir auch um den faktor million Hier ist nochmal eine Statistik Memory & Storage Preises, das ist die Preisentwicklung von 55 bis 2010. Auch hier haben wir wieder eine exponentielle Skala. Hier ist der Preis in US-Dollar pro Megabyte und es ist klar über die Zeit betrachtet, dass diese Preise eben extrem gefallen sind. Es gibt glaube ich kaum eine Technologie weltweit, wo ich solche Wachstumsraten habe wie in der Informatik. gesagt also jede folie die ich mal zehn jahre nicht anfasse muss ich irgendwo ein faktor zwischen 1000 und 1 millionen drauf packen um dann wieder die aktuellen zahlen zu liefern also man könnte eigentlich so alle folien wo irgendwelche zahlen draufstehen in alle fünf jahre erneuern aber das ist die absolute zahl ist glaube ich nicht so wichtig wichtig ist dass diese trends eben hier sind und wichtig ist immer festzustellen ob ein trend noch anhält dass die Technologie gesättigt ist und wir sehen gerade eben auf dem Speichermarkt, dass die Technologie alles andere als am Ende ist. Es gibt neue Technologien mit dem NV-RAM, die dann das ganze Bild noch weiter verändern werden. So, Konsequenz aus der ganzen Geschichte ist klar, dass ich häufig benutzte Daten und Befehle identifizieren muss, dass ich die möglichst nah am Prozessor halten muss, und teure Speicheraufbau und dass ich für selten benutzte Daten, dass ich die auf dem großen und langsamen Speicher auslagere. Deshalb ist ja ein Backup zum Beispiel durchaus auf einem Magnetband, weil ich eben da nur im Notfall darauf zugreifen muss, wenn ich also irgendwelche Daten verloren habe und diese Magnetbande halt im Preis pro Byte extrem billig sind und sie deshalb verwenden kann. beginnen mit dem registern dann primary cache meistens on chip secondary cache ist entweder on chip oder off chip dann der hauptspeicher ist dann meistens eben off chip oder immer off chip eigentlich und dass dieser speicher hier der ist power on speicher das heißt der inhalt des und der andere speicher also alles was auf festplatten steht auf cds dvds solid state disks bleibt mit information erhalten auch wenn die spannung abgeschaltet wird diese speicher hier unten sind natürlich entsprechend langsamer je weiter oben desto schneller je weiter unten desto größer was jetzt neu ist ist eben diese nv ram technologie non volatile ram und der und externen Speicher ein. Non-volatile

heißt eben, dass die Information auch erhalten bleibt, wenn ich Power-Off habe. Gut, also, diese Argumentation brauche ich, um einfach zu begründen, warum ich jetzt diesen ganzen Aufwand auf den nächsten Folien treibe und mich mit Caches beschäftigen muss. den Prozessor, auf der anderen Seite den Speicher und ich hole mir immer alle Daten vom Speicher. Das wäre natürlich in Hardware einfacher, das ist auch einfacher zu denken, aber es funktioniert halt nicht, weil ich es mir einfach nicht leisten kann, den Speicher so schnell aufzubauen. Auf der anderen Seite ist es ja so, dass ich den Zugriff auf den Cache komplett über die Hardware steuere, das heißt, als Benutzer kriege ich davon ja nichts mit. Ich Vielleicht der Optimierer des Compilers macht sich Gedanken um den Cache, aber ich als Programmierer habe damit eigentlich nichts zu tun. Aber als Rechnerarchitekt muss ich natürlich wissen, wo liegen denn die Probleme, wenn ich so ein Cache betrachte. Okay, schauen wir uns mal ein aktuelleres Beispiel an, die Core i7 Architektur. Schauen wir uns die Speicherlatenz an. noch also ich takte mit 2 GHz das heißt ein Clock Cycle sind ungefähr 0,5 Nanosekunden. So wenn ich jetzt ein Datum im Cache vorfinde also wenn ich ein Hit habe dann habe ich im L1 Cache eine Hit Latency von ungefähr 3 Clock Cycles also ich brauche 3 Clock Cycles Zeit um auf diesen L1 Cache zuzugreifen und ich habe ein Speicherwort pro Clock Cycle Durchsatz wenn ich also einmal den Aufbau geschafft habe in drei Clockcycles, dann schaffe ich ein Wort pro Clockcycle als Durchsatz. Wenn ich dann auf einen L2-Cache zugreife, dann haben wir hier also schon mal einen deutlichen Faktor dazwischen, dann sind es nicht mehr drei Clockcycles-Latenz, sondern es sind 14 Clockcycles-Latenz und beim Durchsatz bin ich dann ein bisschen schneller, weil ich dann zwei Worte pro Clockcycle-Durchsatz schaffe, indem ich eben solche Blocktransfers durchführe. Das heißt also, der L1-Cache, der ist natürlich wichtig für einen ganz schnellen Zugriff, dass ich also ganz schnell Daten hole und in meine Register laden kann. Also da geht es darum, Einzelzugriffe zu optimieren. Das schafft man, indem eben diese Zugriffslatenz sehr kurz ist und dieses, dass ich nur ein Wort pro Glockseil gedrückt habe, das stört mich nicht weiter, weil ich sowieso meistens Einzelzugriffe habe. Der Speicherverkehr zwischen L1- und L2-Cache ist im Wesentlichen Blocktransfers. ganz so wichtig, wie groß die Zugriffslatenz ist, sondern wichtiger ist der Durchsatz, weil ich immer ganze Blöcke zwischen den Speichern hin und her schicke. Deshalb, daher kommen eben diese Zahlen. Okay, das ist also die Hit Latency. Ich finde das Datum, eben das ich brauche im Cache, die Miss Latency, also wie viel Strafzyklen ist, wenn ich das Datum nicht im Cache finde. Das heißt, ich cache aber also l1 miss ich habe es aber im l2 cache hits l2 dann habe ich ungefähr zehn zyklen strafstrafe ja das ist von 3 nach 14 ungefähr das sind die strafzyklen die ich habe wenn ich jetzt das datum nicht im l2 cache finde und ich muss auf den hauptspeicher zugreifen ja dann steigert sich das auf 165 zyklen die Das heißt also, ich kann mir jetzt ausrechnen die Cache Bandbreite, die Bandbreite ist 8,5 Bytes pro Zyklus, die ich da transferieren kann und die Speicherbandbreite, also wenn ich so einen Desktop Rechner betrachte, dann bin ich heutzutage bei der aktuellen Speichertechnologie irgendwas im Gigabyte Bereich pro Sekunden und Socket und im Server Bereich bin ich ein Langsamer bin ich bei 3,5 GB

pro Sekunde und so. Wenn Sie sich die leistungsstarken Prozessoren anschauen, dann ist das natürlich deutlich zu wenig, um wirklich in jedem Zyklus mehrere Befehle zu lesen und die dazugehörigen Daten zu verarbeiten. Deshalb brauche ich eben den Cache. So, was ist jetzt das Grundprinzip des Caches? Wir betrachten nämlich die höhere Ebene, das ist die, die näher am Prozessor ist und die nächste Ebene, die niedrigere Ebene, das ist die, die etwas weiter entfernt, also einen Schritt weiter entfernt ist vom Prozessor. Wir haben also hier unseren Prozessor, wir haben hier die oberste Ebene, das ist zum Beispiel der Cache und wir haben hier die Ebene drunter, das ist der Hauptspeicher. praktisch im cache und im hauptspeicher nach dem datum ist er gerade braucht wenn er es im cache findet dann bricht er den hauptspeicherzugriff ab nimmt natürlich das datum aus dem cache wenn er es nicht im cache findet dann greift er auf dem hauptspeicher zu aber dann holt er sich nicht das eine datum aus dem hauptspeicher sondern dann transferiert er gleich den ganzen block eine lokalität der daten dass er damit gleich in den block daten transferiert die vielleicht im nächsten oder übernächsten schritt gebraucht werden das ist wenn man so will ist es nicht bekannt spekulatives holen von speicher in der hoffnung dass durch die daten lokalität dann beim nächsten zugriff auf das datum das daneben liegt der schon im cache eingelagert ist und das funktioniert in der realität eigentlich ganz gut dass ich diesen mechanismus wiederholen kann dass ich also nicht nur einen cache und den hauptspeicher habe sondern dass ich Level 2, Level 3 Cache und den Hauptspeicher habe, ja das ist einfach nur immer wieder dieses Bild hintereinander ausgeführt. Das heißt, für unsere Betrachtungen genügt es erstmal, wenn ich nur zwei Ebenen betrachte. Und was auch wichtig ist, klar, dass die niedrigere Ebene immer eine gesamte obere Ebene beinhaltet. Dieser folgende Cache-Strategie lässt sich natürlich immer daran messen, wie viele Zugriffe finden erfolgreich im Cache statt, im Vergleich oder in Relation zur Gesamtanzahl an Speicherzugriffen. zugriffen. Bei einer Rate kann ich maximal 1 erreichen, also wenn RH gleich 1 ist, dann habe ich vollen Erfolg, dann sind alle Speicherzugriffe aus dem Cache. Dementsprechend ist die Missrate, also die Zugriffe, die nicht im Cache stattfinden, wo ich auf den Hauptspeicher zugreifen muss, ist 1 minus RH. Das sind natürlich Metriken, die ich als Leistungsindikator verwenden kann. Ich Also was kostet es mich, wenn ich jetzt auf den Cache zugreife und was kostet mich, wenn ich eben nicht auf den Cache zugreifen kann, sondern das Datum aus dem Hauptspeicher holen muss. Also diese Latenz, die wird eben nicht in absoluten Zeiten meist angegeben, sondern eben in Zackzyklen, die ich brauche. Ich kann also mit Hilfe dieses Leistungsindikators eine mittlere Speicherzeit ausrechnen. multipliziere die Hitrate mit der Latenz auf dem Cache und dann addiere ich dazu die Missrate multipliziert mit dem Speicherzugriff auf dem Hauptspeicher. Ich kann diesen Ausdruck dann noch ein bisschen vereinfachen, dass ich sage, okay, die mittlere Zugriffszeit ist die Hitzeit plus nicht im Cache finde. Also DAC plus die Missrate multipliziert mit den Strafzyklen. Und die Misspenalty ist natürlich die Zeitdifferenz zwischen Zugriff auf den Hauptspeicher und Zugriff auf den Cache. Gut, machen wir dazu mal ein paar Beispiele. Ja, gerne. Wenn jetzt so ein Block transferiert wird, dann mit den Datenadressen, die auf diese transferierten Daten eigentlich verweisen?

Ja, das ist eine gute Frage. Ich glaube, das sehen wir dann im zweiten Teil der Vorlesung ein bisschen besser, wo wir ja diese Blocktransfers betrachten und auch die Adressrechnung ein bisschen betrachten. Das ist natürlich klar. Ich brauche eine Memory Management Unit, die dafür sorgt, dass diese Adressen alle richtig gestellt werden. Und diese virtuellen Adressen werden ja in reale Adressen umgerechnet und diese realen Adressen, die werden natürlich dann automatisch aktualisiert, sodass eben wenn der Prozessor jetzt auf das Benachbarte Datum zugreift, er automatisch natürlich dann im Cache das Ganze findet. Adressen auf die realen Adressen. Also beteiligt klar ist die Memory Management Unit, die im Prinzip diese Aufgabe hat und die Memory Management Unit muss natürlich die Caches entsprechend mit betrachten, das heißt diese Adressrechnung richtig stellen. Ist die Frage damit beantwortet? Zahlenbeispiele dazu machen und zwar wenn wir jetzt eine Hitrate von 98% haben, das heißt 98% aller Zugriffe sind erfolgreich in Cash wenn wir eine Zugriffszeit auf Cash von 15 Nanosekunden haben und wenn wir eine Miss-Penalty von 150 Nanosekunden haben, also 10 Strafzyklen, wenn wir das Datum aus dem Hauptspeicher holen müssen dann kommen wir auf eine mittlere Zugriffszeit von 18 Nanosekunden, indem wir einfach diese Formel hier anwenden und das ausrechnen. Oder wenn wir das Ganze in Taktzyklen betrachten, wenn wir sagen, okay, die Zugriffszeit auf den Cache sind 15 Nanosekunden und Und der Taktzyklus in 15 Nanosekunden, genau, das ist der von da oben, das ist also Taktzyklus 15 Nanosekunden, ich habe hier einen Zugriff auf den Cash-Zugriffszeit, ich habe 10 Strafzyklen, das heißt die mittlere Zugriffszeit, gleiche Formel, folge vorher, ist 1,2 Taktzyklen. nur 2% Missrate habe, dann habe ich eine Performanceverschlechterung von 20%. Und deshalb muss ich also eben schon, wenn es irgendwie geht, darauf achten, dass meine Hitrate deutlich über 98% ist. Also heutige Systeme, die haben Hitraten 99, irgendwas Prozent, weil eben schon kleine ein bisschen schlechter ist, ergeben riesige Performanceverschlechterungen. Ergibt sich eben aus diesem Beispiel. Das ist so ein bisschen wie das Gesetz von Amdahl, wenn es um Parallelverarbeitung geht. Okay, so, dieses Prinzip, was wir jetzt für zwei Stufen betrachtet haben, das wird einfach wiederholt angewandt in der ganzen Speicherhierarchie. zwischen Cache der zweiten Ebene und dem Hauptspeicher, zwischen Hauptspeicher und Festplatte und dann von mir aus auch noch zwischen Hauptspeicher und dann mit so einem Flashspeicher dazwischen und Festplatte. Uns interessiert aber jetzt hauptsächlich eben die beiden Spieler Cache und Hauptspeicher und dafür wollen wir auch unsere Beispiele machen. Das Prinzip ist das gleiche, nur die Zahlen und die Hardware ist natürlich ein bisschen unterschiedlich dafür. wollen zwischen Cache und Hauptspeicher, dann werden wir fünf Probleme identifizieren als Rechenarchitekten, die wir angehen müssen, die wir lösen müssen. Das ist einmal das Mapping. Also ich habe gesagt, ich transferiere ganze Blöcke von den Hauptspeicher in den Cache und wohin packe ich jetzt einen Hauptspeicherblock im Cache? Also welche Mapping-Strategie fahre ich? Dann muss ich natürlich Suchstrategien fahren. ja im Cache jetzt diesen Hauptspeicherblock finden. Das war so ein bisschen das Problem, was auch gerade angesprochen wurde. Ich arbeite ja mit irgendwelchen Speicheradressen in meinem Programm und auf einmal ist dieser Speicherblock

nicht im Hauptspeicher, sondern im Cache. Und dann muss ich natürlich im Cache jetzt erstmal die richtige Adresse finden. Also die Identifikation des Blocks innerhalb des Caches finden. Dann ist es natürlich so, der Cache ist immer noch eine Teilmenge des Hauptspeichers. Ich habe da bestimmte Blöcke aus dem Hauptspeicher eingelagert. dann wird irgendwann der Cache voll sein und wenn der Cache voll ist und ich brauche jetzt neue Blöcke aus dem Hauptspeicher, muss ich mir eine Strategie überlegen, welche Blöcke verdränge ich für die neu zu ladenden Blöcke. Das sind Ersetzungsstrategien, wo ich mir Gedanken machen muss. Und was immer relativ unkritisch ist, wenn ich in den Caches nur lese, dann hole ich mir einen Block aus dem Hauptspeicher, lese den Block aus dem Cache, Gefährlicher wird es natürlich, da ich ja jetzt zwei Kopien der Daten habe, einen Block im Hauptspeicher, einen Block im Cache Und wenn ich jetzt im Cache Schreiboperationen mache, dann habe ich ja die Konsistenz verletzt Das heißt, wenn ich jetzt nur im Cache schreiben würde, dann habe ich im Cache unter der gleichen Adresse einen anderen Wert als im Hauptspeicher Das darf natürlich nicht passieren, das heißt, ich muss die Konsistenz sicherstellen kommt die Geschichte mit der Memory Management Unit. Die große Frage ist, wo findet jetzt die Übersetzung der virtuellen in die realen Adressen statt? Also adressiere ich meinen Cache virtuell oder adressiere ich meinen Cache real? Das sind unterschiedliche Strategien, wo ich mir eben überlegen muss, was ist günstiger? Wo platziere ich jetzt meine Memory Management Unit? Platziere ich die zwischen Prozessor und Cache oder zwischen Cache und muss dann eben die entsprechenden Adressrechnungen schnell und korrekt machen. Gut, nicht wundern, manchmal verwende ich das Wort Cashline, manchmal das Wort Cashblock. Das bedeutet in dieser Vorlesung zumindest das gleiche, also das ist ein Synonym. Ich werde versuchen hauptsächlich von Cashblocks zu sprechen. Wenn mir Cashline mal rausrutscht, übersetzen Sie es einfach mit Cashblock. jetzt das Mapping so kompliziert? Naja, es geht damit los, dass es grundsätzlich unterschiedliche Möglichkeiten gibt, Daten abzuspeichern. Es gibt erstmal die Location Method. Die Location Method ist das, was Sie vom Hauptspeicher kennen. Der Hauptspeicher ist eine Ansammlung von Bytes und diese Bytes sind einfach von 0 bis n-1 durchnummeriert. Das heißt, ich eine Bits für Adresse spendieren, weil ich einfach die vorgegebene Nummerierung habe. Ich merke mir also einfach nur die Nummer des Speicherplatzes und dann kann ich darauf zugreifen, weil die Daten liegen eben immer an der gleichen Stelle dann. Diese Location Method, als Drawers oder Frames bezeichnet, heißt also, dass ein Datenelement an einer bestimmten eindeutigen Stelle gespeichert ist und diese Speicherzelle wird anhand ihrer einzigartigen Adresse identifiziert. zwischen 0 und n-1 und ich brauche Adressen nicht explizit abzuspeichern. Dann gibt es die assoziative Methode, das ist ein bisschen anders und zwar ist es so, dass ich jetzt hier Nutzdaten habe und zu diesen Nutzdaten kommt ein sogenanntes Tag dazu und dieses Tag ist zu verstehen als eine Adresse. Und wenn ich jetzt dann nach den Daten suche, dann mache ich einen Tag-Vergleich. Dann suche ich also bis ich so ein Tag-Matching habe, das heißt ich suche und wenn ich das Tag gefunden habe, dann sind die Daten, die an diesem Tag dranhängen, die Daten, die ich brauche. Diese Methode der assoziativen Suche wird eben assoziative

Methode genannt und wenn wir uns jetzt mal anschauen, was verwenden wir wo, dann ist es so, dass ich beim Hauptspeicher, der ist einfach durchnummeriert, da habe ich immer die Location Method und beim Cache, das ist jetzt das Problem, teilweise miteinander vermischt, das heißt ein Teil davon ist eben die Location Method bei der Adressierung und ein Teil davon der Information zur Adresse steht in Tags und diese Tags muss ich dann jeweils durchsuchen. So, im Folgenden machen wir ein paar Beispiele und dafür brauchen wir auch die entsprechende Notation. Und zwar, wenn dann Groß A steht, dann steht dieses Groß A immer für Adresse Indizes sind Index 1 und Index 2 und der Index 1 bezieht sich eben auf die Location, auf den Ort, also wenn da ein M steht, M steht für Memory, dann bezieht sich das immer auf den Hauptspeicher, wenn da ein C steht, dann bezieht sich das auf den Cache und das zweite, der Index 2, der bezieht sich jetzt dann bei dem B auf die Blockadresse und beim S auf die Satzadresse. Also wir werden sehen, dass der Cache in Sätze So, das heißt, wenn dann A in B steht, dann ist es die Blockadresse im Hauptspeicher und wenn dann A CS steht, dann ist es die Satzadresse im Cache. Also erster Index Memory oder Cache und zweiter Index Blockadresse oder Satzadresse. Der zweite Parameter ist das Groß S. Das Groß S steht für Englisch Size. Und auch hier gibt es zwei Indizes. Der erste Index ist für den Hauptspeicher oder Cache, also M oder C. Zweiter Index gibt mir die Anzahl von Blöcken oder die Anzahl von Sätzen. Also SMB steht entsprechend für Anzahl der Blöcke im Hauptspeicher und SCS sind Anzahl der Sätze im Cache. Und SCB ist Anzahl der Blöcke im Cache. Also die kann man natürlich entsprechend variieren, diese Indizes. Vollassoziativ heißt, ich kann jeden Block aus dem Hauptspeicher beliebig an irgendeine Stelle im Cache stellen. Ich habe also hier meinen Hauptspeicher, ich habe hier meine Blöcke im Hauptspeicher. In dem Beispiel haben wir einfach 16 Blöcke im Hauptspeicher. Und für 16 Blöcke brauche ich 4 Bits, um die zu adressieren. Das sind also hier jetzt meine Blockadressen. Und was ich jetzt mache, ich kann beliebige Blöcke in meinen Cache einlagern. Ich nehme also jetzt hier den Block DC, den lagere ich hier im Cache ein. Und ich muss natürlich mitnehmen die Blockadresse, die 010, die speichere ich im Cache als Tag mit. Und jetzt, wenn ich auf diese Daten in DC zugreifen will, dann, da das hier ja nicht sortiert sind, diese Tag-Adressen, Tags durchsuchen, bis ich die 0010 finde und dann weiß ich, okay, hier stehen meine Nutzdaten. Das heißt also, das Mapping vom Hauptspeicher auf den Cache, das ist willkürlich. Ich kann einen beliebigen freien Block nehmen oder einen beliebigen Block verdrängen und lagere einfach den Block aus dem Hauptspeicher inklusive dessen Adresse hier ein. Die Adresse das Cache Set kommt hier überhaupt nicht vor, ich habe in diesem Cache nur einen Satz, nämlich der gesamte Cache ist ein Satz, also ACS, die Adresse des Cache Satzes brauche ich nicht, weil es nur einen Satz gibt, entsprechend die Anzahl der Cache Sätze ist 1, der Cache hier soll 8 Cache Blöcke haben, ich kann also hier beliebig 8 Blöcke beschreiben, Und der Hauptspeicher in dem Beispiel hier hat 16 Blöcke. Das heißt, der Cache ist halb so groß wie der Hauptspeicher. Und ich kann immer in etwa die Hälfte der Blöcke, oder ich kann die Hälfte der Blöcke Außenhauptspeicher im Cache einlagern. Ich kann die aber eben an beliebigen Stellen einlagern. Voll assoziativ. Das ist natürlich schön, wenn ich die

an beliebige Stelle schreiben kann. Was ist der Trade-off? bis ich den richtigen Block hier finde. Im Unterschied dazu gibt es einen sogenannten Direct Mapped Cache, also einen direkt abgebildeten Cache. Wir haben jetzt hier auch wieder unseren Hauptspeicher. Unser Hauptspeicher besteht aus 16 Blöcken, genau wie vorher. Was wir jetzt als Adressbits für den Cache. Das heißt, wir haben hier eben statt 4 Bitsstellen, haben wir 3 Bitsstellen im Cache, das sind die untersten 3 Bits, die wir verwenden und diese untersten 3 Bits, die adressieren uns erstmal den entsprechenden Cache-Satz. Das heißt also, der erste Index, der rote Index, der deutet auf den Cache-Satz. Und das heißt, in diesem ersten Eintrag vom Cache, da gibt es jetzt jetzt entweder dieser Block drinsteht, DA, oder dieser Block, DI. So, welcher dieser beiden Blöcke jetzt drinsteht, das unterscheide ich mit dem Tag, weil hier haben wir das Tag 0 und hier haben wir das Tag 1. Das heißt, diese Tags, die muss ich natürlich explizit wieder mitspeichern. Ich habe aber jetzt ein kurzes Tag, ich habe nur das Tag 0 oder 1. Das heißt also, wenn ich jetzt auf den Hauptspeicher zugreifen will, dann schaue ich erstmal im Cache, über die untersten 3 Bits identifiziere ich den Satz und dann muss ich nur schauen, vorhanden ist es das gleiche oder nicht es gibt also nur diese zwei möglichkeiten wenn es da ist es gut kann ich mit arbeiten wenn nicht dann muss ich halt den anderen block an dieser stelle einlagern das heißt dann wird da durch die ausgetauscht das heißt also das mapping ist relativ einfach nämlich die adresse im cache ergibt sich aus der hauptspeicheradresse modulo anzahl der dann muss ich über den Tag-Vergleich feststellen, ob jetzt der Block, den ich brauche, ob der eingelagert ist oder nicht eingelagert ist. Das heißt, wir haben also die Anzahl von Cache-Sets ist die Anzahl von Cache-Blöcken. Wir haben 8 Cache-Sets, somit 8 Cache-Blöcke. Die Anzahl der Hauptspeicherblöcke ist 16. Und zusammen mit dem Tag kann ich dann unterscheiden, ob der richtige Block eben eingelagert ist oder nicht eingelagert ist. Gut, warum ist es jetzt schlau, die oberen Adressbits für die Tags zu nehmen und die unteren Adressbits für die Satzadresse? Ja, denken Sie mal einen Moment darüber nach, warum das eine gute Idee ist. Ich könnte es ja auch anders machen. 3 Bits für die Satzadresse und das untere Bit für das Tag. Ist auch eine Möglichkeit. Die Antwort ist relativ einfach. Die Antwort ist eben die Eigenschaft der Datenlokalität. Ich will, dass benachbarte Blöcke im Hauptspeicher nicht im selben Satz im Cache landen. Nämlich, was passiert, wenn ich jetzt benachbarte Daten im Hauptspeicher, und die auch noch beschreibe, dann führt es zu einer ständigen Verdrängung im Cache. Also hier so ein kleines Beispiel für ein Matrix-Vektor-Produkt. Wir haben hier eine Matrix A, ein Vektor B, ein Vektor X und ein Vektor B. Ich multipliziere die Matrix A mit dem Vektor X und erhalte als Resultat B. Sie kennen den Algorithmus, Sie müssen hier ein Skalarprodukt bilden, also immer Zeile mit Spalte multiplizieren, das gibt mir dann das entsprechende Element im Vektor B. Und wenn ich also jetzt den Vektor B zum Beispiel, wenn ich den jetzt in, das sind hier zwei Cache-Blöcke, die ich brauche, durch unterschiedliche Farben signalisiert. Und wenn ich diese zwei Blöcke natürlich im Cache an die gleiche Stelle, also in den gleichen Satz mappe, dann habe ich das Problem, dass ich bei jedem Skalarprodukt diesen Block einlagern muss und dann diesen Block einlagern muss. unterschiedliche Cachesets einzulagern, dann muss ich die einmal

laden und habe die dann für das ganze Matrix-Vektor-Produkt eingelagert und brauche die nie mehr verdrängen. Deshalb ist es eben wichtig, dass benachbarte Blöcke im Hauptspeicher nicht im selben Satz im Cache landen, weil die würden sich dann immer gegenseitig verdrängen, wenn ich auf die benachbarten Daten zugreife. Das heißt für dieses Beispiel, wenn zwei Blöcke für Vektor X im selben Satz im Cache Deshalb ist das eben eine gute Idee. Ok, das Beispiel vorher war einfach Satz-Assoziativ und dieses Beispiel hier, das ist jetzt zweifach Satz-Assoziativ. Also was machen wir? Die unteren Bits, wieder die roten Bits, die verwenden wir um den Satz zu adressieren. Der zeigt also auf dem richtigen Satz und die oberen zwei Bits, die verwenden wir jetzt für ein Tag. Was ich machen kann bei dieser Organisation, ich habe jetzt meinen Cache so organisiert, dass ich in jedem Satz zwei Blöcke halten kann. Potenziell kann ich aber in jedem Satz vier unterschiedliche Blöcke vom Hauptspeicher einlagern. und hier das sind die vier potenziellen blöcke die ich im ersten satz einlagern kann oder im satz 0 einlagern kann und dementsprechend kann eine beliebige mischung aus diesen vier blöcken hier vorne in den ersten beiden stellen eingelagert werden ich muss dann natürlich über das tag vergleichen mal schauen welche blocke tatsächlich eingelagert ist und zur not wenn ich da und de eingelagert dann muss ich an einen dieser beiden verdrängen durch diesen blog die welchen verdränge das ist dann die entsprechende strategie die wir noch betrachten müssen das heißt also meine haupt speicheradresse die teile ich jetzt hier auf in zwei bits für die satzadresse und in zwei bits fest tag die satzadresse nicht mehr eindeutig fest wo das im cache liegt und die tag adresse gelagert ist und auf welchen ich zugreifen muss. Das ist jetzt zweifach satz-assortiv. Und was ist jetzt voll-assortiv? Haben wir schon besprochen, dass jeder Hauptspeicherblock einen beliebigen Blockframe im Cache nutzen kann, das heißt die komplette Adresse wird im Tag gehalten. Und Direct Map, direkt abgebildet, ist eben genau das Gegenteil, dass jeder Hauptspeicherblock Übers Techno unterscheiden, ist er jetzt eingelagert oder nicht eingelagert. Okay, hier nochmal die Kürzel, also AMB-Blockadresse im Hauptspeicher, ACB-Blockadresse im Cache, SCB-Cache-Größe in Blöcken und es gilt beim direkt abgebildeten Cache immer die Abbildungsvorschrift oder die Map-Vorschrift. Die Adresse vom Cache-Block ergibt sich aus Adresse vom Hauptspeicher, Modulu, Anzahl der Cache-Blöcke. Ich muss natürlich nur die Texte im Cacheblock jeweils vergleichen und für ein Direct Map gibt es nur zwei Möglichkeiten, ja oder nein. Okay, Satzassosiativ, allgemein ausgedrückt, jeder Hauptspeicherblock ist eindeutig mit einem Satz assoziiert. Das sind die unteren Bits, die mir die Satzadresse geben und so ein Satz kann eben bis zu M Blockframes enthalten Blockframes innerhalb des Satzes ist beliebig. Das heißt, was jetzt hier an der ersten Stelle steht, das ist nicht vorgegeben. Also hier kann entweder DA stehen oder DE stehen oder DI oder DM. Also dieses Mapping hier oben, die zwei Plätze sind zu vergeben. Und welche dieser vier Blöcke auf diesen beiden Plätzen landen, das ist beliebig. Das ist damit gemeint. Das heißt also, dass das Mapping von Blockframes innerhalb eines Satzes beliebig ist. Und was bedeutet jetzt M-Fach, Satzassortativ? Ich habe wieder die Kürzel wie vorhin. Und dann gilt, dass die Adresse des Cache-Satzes ist die Hauptspeicheradresse Modulo Anzahl der Cache-Sätze. Gut, man nennt

die Adresse des Cache-Satzes oder die Adresse des Cache-Blocks, nennt man auch eben Index. dass dieser Faktor M bestimmt sich aus Anzahl der Cache-Blöcke durch Anzahl der Cache-Sätze und M wird auch bezeichnet als Grad an Assoziativität. Ein vollassoziativer Cache mit M-Blöcken ist M-fach-assoziativ und ein direkt gemappter Cache, da habe ich ja keine Wahlmöglichkeiten, der ist einfach satzassoziativ. Das soll dieses Beispiel hier nochmal verdeutlichen. Wir haben also hier den Hauptspeicher, der besteht hier aus 32 Blöcken. Und wenn ich jetzt einen vollassoziativen Cache habe, dann kann dieser Block, im Beispiel mit der Nummer 12, kann der an der beliebigen Stelle im Cache landen. Das ist vollassoziativ, weil ich die komplette Adresse als Tag nehme und ich muss dann halt immer den kompletten Cache durchsuchen, wo dieser Block gelandet sein könnte. Wenn das jetzt Direct Mapped ist, dann habe ich eine einfache Modulo-Operation für die Adresse, dann sage ich also die Adresse vom Cache Block ist eben 12, ist die Speicherstelle hier, Modulo, ich habe 8 Cache Blöcke, Modulo 8 und das ist 4, das heißt der landet auf jeden Fall an der Stelle 4 und da muss ich ihn suchen und es gibt zwei Möglichkeiten, entweder er ist es oder er ist es nicht. und die Adresse 12, die kann jetzt entweder im ersten oder im zweiten Block des Cache-Sets 0 landen. Warum, wie komme ich drauf? Ich habe hier die Adresse 12, die muss ich jetzt Modulo 4 rechnen, weil ich 4 Cache-Sätze habe, dann kommt die 0 raus, also lande ich im Cache-Satz 0 des Cash-Blocks, die kann jetzt 0 oder 1 sein, das muss ich jetzt über das Tag unterscheiden und dann sehe ich, ob der Block eingelagert ist oder nicht eingelagert ist. Also man muss hier noch zwei Tag-Vergleiche durchführen. So, das ist jetzt natürlich super, dass ich so viele unterschiedliche Möglichkeiten habe und da stellt sich natürlich sofort die Frage, warum denn überhaupt diese ganzen Möglichkeiten betrachten, ist denn nicht eine dieser drei Möglichkeiten die beste und ich konzentriere mich auf eine dieser drei Möglichkeiten. ja leider ist dem nicht so ja es kommt immer wie immer ein bisschen drauf an was ich gerade machen will also da ich ja zum beispiel aus dem instruction cache im wesentlichen lese und block transfer mache kann da die lage natürlich ganz anders sein als beim data cache wo ich lese und schreibe drauf und auch nicht nur blöcke sondern auch einzelne daten lese und schreibe das heißt die wahrheit wie immer in der rechner architektur nicht nur da die simulationen herausfinden welches ist für mich die beste lösung gut man kann es auch mit natur gesetz begründen es gibt nämlich wenn man sein eigenes zimmer anschaut gibt es ein naturgesetz das gilt dass der aufwand für die platzierung also der aufwand fürs aufräumen plus dem aufwand den ich betreiben muss um irgendwas zu suchen heißt wenn ich meine sachen aufräume dann finde ich sie sofort habe natürlich damit arbeit beim aufräumen wenn ich in chaot bin und alles einfach irgendwo hin schmeiße ja dann habe ich unter umständen ein bisschen aufwand beim suchen gesamtaufwand ist konstant also das ist eine gute begründung für den spruch wer sein zimmer aufräumt ist nur zu faul zum suchen ja gut also Adresse und diese Hauptspeicher Adresse werden wir aufteilen einmal in den Block Offset, weil ich ja innerhalb des Blocks adressieren muss dieser Block Offset wird immer der gleiche sein im Hauptspeicher und im Cache dann habe ich ein paar Bits in der Adresse die brauche ich für den Index und ein paar Bits in der Adresse die brauche ich für das entsprechende Tag,

den Index brauche ich um Und um den richtigen Block zu finden, da brauche ich eben das Tag. Gut, also Block Offset ist die Wartadresse innerhalb des Blocks, ist identisch zwischen Hauptspeicher und Cache für die Beispiele, die wir betrachten. Index ist die Adresse des Satzes im Cache und das Tag brauche ich eben, um den richtigen Block zu finden. Jetzt kann man sich dann ausrechnen, wie groß die einzelnen Sachen sind. aus F, aus dem Mapping Faktor, den sehen wir gleich auf der nächsten Folie und zwar eben Logarithmus Dualis davon, weil wir kodieren den ja binär, dann haben wir der Index, ergibt sich natürlich aus der Anzahl der Cache Sets und davon Logarithmus Dualis, also die Anzahl der Cache Sets, so viel Bits brauche ich für den Index dann und den Block Offset, das ist natürlich entsprechend die Hauptspeicheradresse, die Länge der Hauptspeicheradresse von den anderen komponenten abgezogen also das was an bits übrig bleibt für die folgenden beispiele machen wir es meistens so wir gehen von der 32 bit architektur aus die byte adressiert ist das heißt in so einem block da steht im wesentlichen meistens ein wort drin erstmal für um der einfachkeit halber wir brauchen also zwei bits für den block offset mit zwei bits kann ich 4 byte adressieren und vier mal acht ist 32 habe ich also ein wort mit 32 bit daher bestimmt sich das muss nicht so sein, das ist aber eben auf den nächsten Seiten für die Beispiele so. So, den Mapping Faktor F, den kann ich berechnen ganz einfach, indem ich die Anzahl der Hauptspeicherblöcke durch die Anzahl der Cacheblöcke dividiere und multipliziere mit dem Assoziativitätsgrad, also mit dem M, was wir auf den Folien vorher schon berechnet haben, das heißt die Formel m. Das ist ein Maß dafür wie verschiedene Hauptspeicherblockframes zu jedem Blockframe im Cache gemappt werden. Und die Division von Anzahl der Hauptspeicherblöcke durch Anzahl der Cacheblöcke, dieser Quotient, der wird auch als Cache Coverage Degree bezeichnet. Wir haben 32 Hauptspeicherblöcke und wollen das Ganze mal als M-fach assoziativen Cache auslegen. Der Cache Coverage Degree ist leicht auszurechnen, das ist 4. Also der, wenn man so will, der Cache ist ein Viertel so groß wie der Hauptspeicher. m multipliziert. Gut. Also wenn wir jetzt voll assoziativ sein zu wollen, also 8-fach assoziativ, wir haben 8 Cache-Blöcke, also 8-fach assoziativ, wollen wir vergleichen mit Direct Map, das heißt mit 1-fach Satz-Assoziativ und mit 2-fach Satz-Assoziativ. m gleich 1, m gleich 2. So und das setzen wir jetzt einfach in diese Formeln ein Die Anzahl der Cache Sätze ist eine einfache Beziehung. Das ist die Anzahl der Cache Blöcke dividiert durch die Assoziativität. Das heißt wir haben hier 8 Blöcke, wir sind 8 fach assoziativ. Also im Fall vom 8 fach Satz Assoziativen kommt dabei ein Cache Set raus. Also es ist nur ein Cache Set. Wenn ich Direct Map bin, dann habe ich natürlich 8 Cache Sets. Also die gleiche Division wieder, das m ist 1, 8 durch 1 ist 8. Und hier bin ich zweifach Satzassizativ, ich muss also die Anzahl der Cache-Blocken durch 2 dividieren, kommt also 4 Cache-Sets raus dabei. So, wie komme ich jetzt auf die Adresse, also welches Cache-Set muss ich verwenden, um den Hauptspeicherblock abzulegen? was ich hier ausgerechnet habe ich habe also in dem fall hier modulo 1 in dem fall hier modulo 8 und in dem fall hier modulo 4 das heißt ich kann ganz einfach mir die adresse des cache sets ausrechnen aus der hauptspeicheradresse durch die entsprechende modulo operation so jetzt kann ich noch den mapping factor f

ausrechnen der ist smb dividiert durch also anzahl der blöcke im Das heißt also hier haben wir 32 für F, hier haben wir 4 für F und hier haben wir 8 für F. Und ja klar, wie viele Bits brauche ich jetzt um 32 Möglichkeiten zu kodieren? Dann brauche ich 5 Bits, um 4 Möglichkeiten zu kodieren brauche ich 2 Bits und um 8 Möglichkeiten zu kodieren brauche ich 3 Bits. Also ich kann hier eben die Größen ausrechnen, wie die Felder belegt sind. Okay, machen wir mal ein konkretes Beispiel dazu. Konkretes Beispiel ist hier, wir haben hier einen Hauptspeicher von 64 KB, der Cache soll 2 KB groß sein. Also ein Block ist 16 Byte groß. Das heißt also, ich habe 2 hoch 4 Bytes in den Blöcken, ich habe 2 hoch 7 Blöcke, 128 ist 2 hoch 7, und ich habe 4096 gleich 2 hoch 12 Blöcke im Hauptspeicher. 12 Blöcke, mein Cache hat 2 hoch 7 Blöcke. Der Cache Coverage Degree ist einfach 4096 durch 128 ist 32 und das ganze soll voll assoziativ sein, das heißt ich habe nur einen Cache Satz. Die Anzahl der Cache Blöcke sind 2048 Bytes dividiert durch also 2 Kilobyte dividiert Also 128, das ist mein M, was ich brauche. Und mein Coverage Degree ist SMB durch SCB mal M, ist also 32 mal 128, ist also entsprechend 2 hoch 12. Gut, das heißt also, ich brauche 12 Bits beim Vollassoziativen Cache, brauche ich 12 Bits für das Tag, ich brauche kein Bit für den Index. und ich mache die suche ausschließlich über die tagsuche gut insgesamt ist natürlich die hauptspeicheradresse 16 bit breit weil ja noch der block offset immer dazu kommt und ja der block offset kann man eben hier ablesen 2 hoch 4 bytes minus 12 ist 4, kommt man dann natürlich auch auf 16 Bit. Gut, als Bild schaut es natürlich so aus, dass ich habe jetzt auf dieser Seite hier den Cache, ich habe auf dieser Seite hier den Hauptspeicher. Mein Cache hat 128 Blöcke, der Hauptspeicher hat 4096 Blöcke, ich habe 12 Bits für das Tag, 0 Bit für den Index und 4 Bits für den Block Offset. Und das heißt, so ein Block, den ich hier im Hauptspeicher habe, Der Block 0, der wird natürlich nach einer eindeutigen Vorschrift hier zugeordnet, eben über das Tag dann, über das Tag adressiert in die entsprechende Stelle meines Caches. Das heißt, die gesamten 12 Bits der Hauptspeicherblockadresse werden eben zu Tag Bits für den Cache und der Nachteil ist natürlich, ich muss hier können also beliebige Blöcke landen, wie ich will, ich kann die auch beliebig verdrängen. Wenn ich also eine Adresse suche, muss ich eben alle 128 Tags, die gerade eingelagert sind, vergleichen und schauen, ob das Tag, das ich aus dem Hauptspeicher habe, mit einem dieser Tags übereinstimmt. Wenn ja, dann habe ich den Block gefunden. Wenn nein, dann muss ich mir den Block aus dem Hauptspeicher holen und hier irgendwo einlagern. Ich kann den aber beim Vollassoziativen Cache eben an beliebiger Stelle einlagern. Ich habe also nichts fest adressiert im Cache. merken. Gut, wenn ich das Ganze jetzt Direct Map mache, gleiches Beispiel, also Hauptspeicher 64 KB Cache 2 KB, Block 16 Byte Größe, kann ich jetzt wieder alles ausrechnen, Cache Coverage Degree, die Anzahl der Cache Sets sind jetzt 128 Cache Sets, also wie viele Bits brauche ich, um 128 Cache Sets zu adressieren, 2 hoch 7, brauche also 7 Bits jetzt für meinen Index fünf bits ja wir haben insgesamt zwölf bits hier vorne die verbleibenden fünf bits die brauche ich natürlich auch fürs tag man könnte sich das aber auch über die formel ausrechnen wieviel bits ich fürs tag brauche indem er eben diese f-formel anwendet und man sieht wenn man das ausrechnet dass dann eben auch 25 rauskommt also

fünf bits fürs tag block offset ist genau wie vorhin bleibt dass jetzt im obersten Cache Set, also im 0. Cache Set praktisch, da wird eingelagert der Block 0 und alles, was mit Modulo 128 0 ergibt. Das heißt, eben entsprechend der Block 128, genau stimmt, Block 128, Block 256, 128 gleich 0 ergibt, genau, richtig. Also die hellblauen Blöcke hier, die werden alle hier im 0. Cache Set eingelagert, dann eben hier die rosa Blöcke, die werden im letzten Cache Set eingelagert. Also ich mache eine einfache Rechnung, indem ich eben Modulo 128 nehme, das heißt mein Index hier, also die Index Bits, die hier eben auf den entsprechenden Cache Set verweisen, habe ich noch 5 Bits für die Tags, dann muss ich hier nachschauen, ob der entsprechende Block tatsächlich drin steht. Da habe ich also mehr die Möglichkeit nachzuschauen, ja oder nein, mehr Möglichkeiten gibt es nicht. Es gibt also nur einen Tagvergleich, entweder der Block ist jetzt im Cache oder ist nicht im Cache. liegt eben dazwischen also zum beispiel bei zweifach satzassosiativ gleiches das ist jetzt hier ein anderes beispiel wir haben jetzt 16 megabyte hauptspeichergröße 16 megabyte sind 2 hoch 22 blöcke ja wir haben aber 24 bit adressen deshalb weil ein block sind 4 bytes also brauche ich noch mal zwei bit und unser Cache ist jetzt 2 mal 32 KB groß, also insgesamt 2 hoch 14 Blöcke groß. Ich will das Ganze zweifach satzassoziativ machen, also die Cache-Blöcke sind 2 hoch 14, mein M ist 2, zweifach satzassoziativ, jetzt kann ich dann meinen Mapping-Faktor berechnen, der Mapping-Faktor ist 2 hoch 9, also 512, und über diesen Mapping-Faktor bestimme ich natürlich die Anzahl der Bits für das Tag, Das heißt, wir haben jetzt diese 9 Bits brauchen wir für das Tag und die verbleibenden 13 Bits, die verbrauchen wir für den Index. Also die Summe aus dem Ganzen muss natürlich 24 Bits ergeben. 2 Bits sind festvergeben für den Block Offset und 13 und 9 kann ich mir über diese Formeln berechnen. Was ich jetzt, wie ich diese Hauptspeicheradresse in Cache-Adressen praktisch umwandle. Bild, wie die entsprechenden Hauptspeicherblöcke im Cache eingelagert werden. Wir haben hier wieder unseren Cache mit 2x32 KB, wir haben hier unseren Hauptspeicher mit den entsprechenden Blöcken und wir sehen eben, dass die Indexbits in der Hauptspeicheradresse verwendet werden, die weisen eindeutig auf einen Satz im Cache und die Tag Bits, die wir haben, diese Tag Bits, die werden dann mitgegeben, die werden an die Daten angeheftet und ich muss jetzt hier einen Tag Vergleich machen, welcher Block eingelagert ist und ich kann also jetzt sehen, ok, hier ist der Block mit dem Tag 1FF eingelagert, hier ist der Block mit dem Tag 001 eingelagert, einen anderen Block, also zum Beispiel den hier mit 7FFC Index einlagern möchte, dann muss ich einen von diesen beiden verdrängen. Weil der Index 7FFC verweist eindeutig auf dieses Cache Set und für dieses Cache Set habe ich zwei Blöcke und wenn eben der neue Block da nicht vorhanden ist, dann muss ich einen dieser beiden Blöcke verdrängen. tags müssen immer durchsucht werden und ein vergleich angestellt werden also index liefert satzadresse im cache jeder satz enthält zwei blöcke das heißt ich muss bis zu zwei tags vergleichen um die hauptspeicheradresse zu verifizieren gut das beispiel hier zeigt wie die neue blockanforderungen also ich will jetzt hier diesen block einlagern beide plätze sind belegt im cache das heißt ich muss mich entscheiden welchen von diesen beiden blöcke ich verdränge kann dann den Block da unten einlagern. Den markierten Block einlagern. So, die große Frage

ist jetzt natürlich, welchen dieser beiden Blöcke soll ich denn verdrängen? Was bringt mir am meisten Performance, was schadet am wenigsten? Bevor wir die Verdrängungsstrategien betrachten, noch ein kleines Beispiel zum 80386-Cache. Also das hat es mal tatsächlich gegeben. jugendlichen alter wahrscheinlich gar nicht mehr kennen das war so einer der ersten pc ist die ich gehabt habe also ende der 80er jahre und der hatte immerhin schon 16 megabyte hauptspeicher also 22 blöcke ein wort war 4 bei groß der cash war 64 kilobyte groß da rückt cash und ich habe also der Adresse für den Index und 8 Bits für das Tag und der Aufbau war jetzt so, dass ich habe hier meinen Prozessor, der Prozessor spuckt mir die Hauptspeicheradresse aus, auf die ich, mit der ich ihm auf den Hauptspeicher zugreifen will, allerdings geht das Ganze natürlich über den Cache, das heißt es wird erstmal im Cache abgegriffen diese Adresse, sind, dann liefert natürlich der Cache die Daten und wenn sie nicht im Cache vorhanden sind, dann kommen diese Daten direkt aus dem Hauptspeicher. Um das zu entscheiden, brauche ich eine Cache Control Logic, die diese Vergleiche eben durchführt entsprechend. Ich habe also hier mein Tag Target, also meine Adresse, Zieladresse wird verglichen mit den Tags im Cache. Wenn ich dieses Tag nicht im Cache finde, dann muss ich die Daten aus dem Main Memory holen. Sie sehen, es passiert vollkommen in Hardware. Also als Programmierer habe ich mit dieser Anordnung nichts zu tun. Das Einzige, was ich natürlich als Programmierer beeinflussen kann, ist, wie lege ich meine Daten ab. Also ist es zum Beispiel für mich günstiger, eine Matrix zeilenweise oder spaltenweise zu speichern. der Matrix anstellen will, ob ich halt zeilenweise oder spaltenweise darauf zugreifen kann. Also damit kann ich das ganze Verhalten natürlich indirekt beeinflussen, indem ich das möglichst günstig anlege für den Cache, aber dafür muss ich natürlich Detailkenntnisse zur Hardware haben und normalerweise ist es so, dass die Optimierer und die Hardware das besser kann als der Programmierer. Okay, jetzt die Frage, Cache ist voll, ich muss aber Und welchen Block soll ich denn einlagern? Das heißt, welchen Block soll ich verdrängen? Und beim DirectMap ist es ganz einfach. Da gibt es keine Entscheidung, weil es ist klar, in welchem Cache-Block der Hauptsprecherblock landet. Das ist DirectMap, direkt abgebildet. Und beim Voll- und Satzassertif muss ich mir Gedanken machen, welcher wird voraussichtlich nicht mehr benötigt. Das ist jetzt wieder eine spekulative Geschichte. Ich kann jetzt Datenlokalität zu Hilfe nehmen. Ich kann sowohl zeitlich als auch räumlich das betrachten. Ich kann also sagen, ein Datum, das ich längere Zeit nicht gebraucht habe, werde ich vermutlich in Zukunft auch nicht mehr brauchen. Ob das stimmt oder nicht, das ist eine ganz andere Sache. Das ist Spekulation. Das ist ein bisschen Lotteriespiel. Da kann ich Glück oder Pech haben. Da kann ich nur erfahrungswerte Heuristiken zu Hilfe nehmen, um diese Entscheidungen zu treffen. Dementsprechend gibt es unterschiedliche Strategien. eine Strategie wäre FIFO, First In First Out das heißt der älteste Eintrag wird ersetzt also den auf den ich am längsten nicht mehr zugegriffen habe dann Least Recently Used der Block, ja der älteste Eintrag also was am ältesten praktisch eingetragen wurde da wird also mit markiert, wann wird der Block eingelagert beim Least Recently Used welchen Block habe ich denn seit längerem nicht mehr zugegriffen oder auf welchen Block habe ich am längsten

nicht mehr zugegriffen, dann wird dieser ersetzt. Dann kann ich mitmessen, wie oft wird auf den Block zugegriffen. Und dann kann ich eine Least Frequently Used Strategie fahren. Das ist der Block, auf den am seltensten zugegriffen wird, den ersetze ich. Das ist ja auch eine vernünftige Strategie, indem ich sage, okay, da wird nur ganz ab und zu mal drauf zugegriffen, also kann ich den auch locker mal verdrängen. einen zufällig gewählten Block widersetzt. Die ist natürlich am allereinfachsten zu realisieren. Ich zeige einfach zufällig auf einen Block, den haue ich raus und ersetze den. Und wir werden gleich sehen, dass diese einfache Strategie gar nicht so dumm ist im Vergleich zu den doch elaborierten, also sehr auffälligen anderen Strategien. Schauen wir uns aber erstmal so eine LRU-Strategie an. nehme den Block, der am längsten nicht mehr verwendet worden ist, zur Verdrängung. Das heißt, hier geht das Beispiel los, die Zeit läuft von links nach rechts. Das sind jetzt die Block-Frame-Adressen, die also im Cache vorliegen. Und initial ist die LRU-Blocknummer, also der Block, der am längsten nicht mehr verwendet wurde, der Block mit der Nummer 0. So, wie ist die Strategie? Wenn jetzt hier ein Block eingelagert wird, der gerade diese LRU-Nummer hat, ersetzen und zwar suche ich mir dann den Block der am längsten nicht mehr verwendet wurde. Also ich lagere hier die 0 ein, damit muss ich diese 0 hier ersetzen und dann schaue ich, okay der Block 3 wurde am längsten nicht mehr verwendet, also trage ich jetzt, verpasse ich den Block 3 die LRU Nummer. So jetzt weiß ich also der Block 3 ist LRU, solange ich jetzt Blöcke einlagere die nicht die Nummer 3 haben passiert gar nichts, so und jetzt lagere ich wieder die 3 an, jetzt haben wir also hier wieder identisch und das heißt also, ich habe inzwischen natürlich mitprotokolliert, auf welchen Block wurde am längsten nicht mehr zugegriffen, in der Situation war es die 2, in der Situation war es natürlich der Block 1, weil die 2 habe ich ja hier eingelagert, auf den am längsten nicht mehr zugegriffen wurde und dementsprechend, wenn ich jetzt die 3 einlagere, ja, nehme ich die 1 hier für die LRU-Blocknummer, ja, kann das hier natürlich auch so, ich die Markierung auflösen, die Markierung geht weiter nach der 0. Im nächsten Schritt lagere ich aber gleich die 1 ein, habe also hier wieder identisch, habe aber inzwischen das auf die 0 weitergeben, die LRU-Blocknummer, das heißt die nächste LRU-Blocknummer bekommt die 0, dann lagere ich die 3 ein, da verändert sich wieder nichts, die 0 hat diese Nummer, jetzt wird dann wieder die 0 eingelagert, wenn die 0 eingelagert wird, dann geht es nicht mehr genutzt wurde und wird entsprechend dann gesetzt. Also das ist die Strategie, das heißt ich muss immer protokollieren, welcher Block wurde am längsten nicht mehr genutzt und muss dann einem Block immer diese LRO-Blocknummer geben. Ist also nicht sehr aufwendig zu realisieren, aber ist doch Aufwand. Und jetzt wollen wir mal diesen Aufwand vergleichen mit dem Erfolg. Das heißt wir vergleichen einfach die Strategie mit der Random-Strategie. habe ich bei LRU mehr Aufwand, also random ist einfacher, da habe ich einfach eine Zufallszahl, hier muss ich protokollieren bei LRU und dann vergleiche ich unterschiedliche Strategien, also 2-fach Satz-Assoziativ, 4-fach Satz-Assoziativ und 8-fach Satz-Assoziativ und unterschiedliche und berechne dann die Missraten. Ich will ja die Missraten so klein wie möglich machen. Und was wir im Endeffekt im Ergebnis sehen, ist, dass diese Zahlen sich sehr ähneln, und dass natürlich

je größer der Cash ist, desto kleiner wird die Missrate. Das ist ja das, was wir eigentlich sehen wollen. Und Sie sehen hier zwischen 16 und 64 ist der Faktor 4, da wird also die Missrate deutlich verkleinert, ebenfalls um den Faktor 4 fast. 3. Während hier wird der Cash nochmal um den Faktor 4 vergrößert, dann sehen Sie, dass diese Verbesserung hier natürlich nicht mehr ganz so groß ist. Also es ist in Ihrer Glaube zu denken, wenn der Cash beliebig groß ist, dass das dann eben Aufwand im Verhältnis zu Ertrag steht. Der Ertrag wird immer kleiner, den ich erziele, deshalb gibt es hier irgendwo natürlich ein Optimum, wenn ich Aufwand und Ertrag betrachte. So, wenn wir jetzt die zwei Zahlen relativ ähnlich sind, die ich erziele und insbesondere, wenn der Cash größer wird, wird der Unterschied zwischen diesen beiden Zahlen immer geringer. Das heißt, gerade wenn ich jetzt diese Cashgröße anschaue, dann frage ich mich ja, lohnt es sich denn so eine komplizierte Strategie wie LRO zu fahren, wenn ich mit Random fast genauso effizient bin? Typische Frage des Rechnerarchitekten, Antwort ist einfach, wenn mein Rechner auf absolut Leistung getrimmt ist, dann lohnt sich jeder Aufwand, ich meinen Rechner in großen Stückzahlen möglichst billig verkaufen möchte, dann lohnt sich das Ganze natürlich nicht und dann mache ich das Ganze random. Okay, so, jetzt müssen wir noch betrachten das Problem des Schreibzugriffs. Wir haben ja ganz am Anfang schon festgestellt, dass wenn ich lese, dass es eigentlich kein Problem ist, es ist einfach. schaue ich einfach, ist das Datum im Cache oder nicht, also habe ich einen Hit oder einen Miss. Und wenn ich einen Hit habe, dann lese ich halt den Cache-Block-Frame und wenn ich einen Miss habe, dann lese ich den Hauptspeicher-Frame. Das Lesen des Caches und Hauptspeicher wird immer gleichzeitig angeworfen, damit ich keine Zeit verliere. Und wenn ich eben einen Miss feststelle, dann breche ich den Cache-Zugriff ab und wenn ich einen Hit feststelle, dann breche ich den Hauptspeicher-Zugriff ab. Vorsichtshalber beides und abbrechen kann man ja immer noch, einfach um keine Zeit zu verlieren. Aber lesen ist einfach, weil mit Lesen kann ich die Konsistenz eben nicht verändern. Schwieriger wird die Strategie, wenn ich Daten erneuere und was ich dann machen muss, ist klar, erstmal muss ich wieder über einen Tag-Vergleich herausfinden, okay, ist das Datum, das ich ich natürlich dieses Datum verändern, dann muss ich natürlich im Cache das entsprechend verändern und muss es im Hauptspeicher verändern. Und wenn ich einen Miss habe, dann hole ich mir erstmal den Block aus dem Hauptspeicher in den Cache und dann fahre ich hier ganz normal weiter, so wie vorher. Jetzt kann ich mir natürlich anschauen, wie oft dritten Schreibzugriffe auf, wie oft Lesezugriffe und dann fällt mir auf, ok, Lesezugriffe Brauche ich für jeden Befehl mindestens einen Structure Read. Das heißt, ich habe einen Lesezugriff auf den Instruktionsspeicher. Und dann habe ich deutlich mehr Load- als Store-Befehle. Also 20% aller Befehle sind Loads und 10% aller Befehle sind Stores. Das heißt also, die Schreibzugriffe sind deutlich seltener. Okay, jetzt muss ich mir eben unterschiedliche Schreibstrategien einfallen lassen, die wir miteinander vergleichen wollen. dauert, die mir noch verbleiben. Deshalb würde ich an dieser Stelle ganz gerne die Vorlesung wählen.