und wir stecken mitten im thema cash ist und wir müssen uns jetzt gedanken machen was passiert wenn ich in den cash schreibe und lesen ist harmlos weil da verändere ich ja nichts am Moment, wo ich schreibe, habe ich natürlich das Problem, dass ich den Cache mit dem Hauptspeicher konsistent bzw. kohärent halten muss. Was der Unterschied ist zwischen Konsistenz und Kohärenz, besprechen wir noch. Gut, also was passiert? Wenn ich auf ein Cache erstmal lesen zugreife, dann schaue ich erstmal nach, ist das Datum im Cache oder nicht und gleichzeitig Wenn ich aber feststelle, das Datum ist gar nicht im Cache, dann breche ich diese Aktion natürlich ab, weil da kann ich ja nichts auslesen. Und ansonsten gleichzeitig lese ich den Hauptspeicher und hier breche ich natürlich ab, wenn ich das Datum im Cache finde. Warum mache ich das gleichzeitig? Ganz einfach, um keine Zeit zu verlieren. Ich mache dazu jetzt ein bisschen unnötige Arbeit, die ich dann abbrechen muss, aber ich habe keine Zeit verschwendet, weil beide Aktionen gleichzeitig laufen. beim schreiben ist es natürlich ein bisschen schwieriger wegen der konsistenz problematik gut wie geht's los also erste aktion ist erst mal wieder der tag vergleich sie erinnern sich als letztes mal da haben wir ja das mapping besprochen vom hauptspeicher auf dem cache und über die tags kann ich feststellen ob so ein cache block eingelagert ist oder nicht wenn er ich das Datum zum Prozessor schicken, kann das Datum modifizieren und kann das Datum in den Cache praktisch wieder zurückschreiben und muss das natürlich dann auch in den Hauptspeicher zurückschreiben, damit eben die beiden konsistent bleiben. So, wenn ich jetzt aber das Datum nicht finde, wenn ich also ein Miss habe, dann ist die Vorgehensweise im Wesentlichen die gleiche, der einzige Unterschied ist, bevor ich anfange, habe ich ja quasi den Hit und dann kann ich also mit dem Schritt 2a weitermachen, kann ganz normal, wie wenn ich einen Hit hätte, arbeiten. Die Frage ist jetzt, wie attackiere ich das Problem der Cache-Consistenz und dafür gibt es eben unterschiedliche Strategien und die wollen wir im Folgenden mal ein bisschen genauer betrachten und dann auch miteinander vergleichen. Es geht ja wie immer um Performance auch. Erstmal schauen wir uns, wie oft habe ich denn überhaupt Schreibzugriffe, also wie wichtig ist denn das Ganze Wenn wir uns das jetzt nochmal anschauen, normale Architekturen heutzutage sind riskartig, also Load-Store-Architekturen, dann brauche ich natürlich, jeden Befehl muss ich aus dem Speicher erstmal laden. Das heißt, ich habe 100% aller Befehle lösen ein Read aus, nämlich ein Instruction Read. Und dann von diesen Befehlen lösen 20% aus ein Load, also irgendwelche Daten werden aus dem Speicher geladen und 10% aller Befehle, die machen ein Store, schreiben also Speicher. Problematik natürlich die interessanten. Okay, also erste Strategie, die wir uns anschauen wollen, ist, ich mache einfach ein Write-Through, also übersetzt auf Deutsch könnte man auch durchgängiges Schreiben dazu sagen. Und wie ist die Vorgehensweise? Vorgehensweise ist, dass ich gleichzeitig bei jedem Schreibzugriff auf den Cache und auf den Hauptspeicher zugreife. Das heißt, jedes Datum, was geschrieben wird, wird in beiden Speichern aktualisiert und kann natürlich dann erst weitermachen, wenn der längere, länger dauernde Zugriff, und das ist ja immer der Hauptspeicherzugriff, wenn der abgeschlossen ist. Wir haben hier, sehen wir einfach nochmal die Vorgehensweise, wir haben die CPU möchte praktisch an der Adresse 3 das Datum B schreiben.

Also was passiert? Im ersten Schritt legt die CPU die Adresse 3 und das Datum B an die entsprechenden zugehörigen Busse an. Datenbus, die Adresse kommt auf den Adressbus, die Daten kommen auf den Datenbus. Wenn ich jetzt einen Cache-Hit habe, dann wird das alte Datum A durch das neue Datum B überschrieben und das ist also der einfache Fall und wenn ich einen Cache-Miss habe, dann wird ein neuer Eintrag im Cache erzeugt auf jeden Fall, also das heißt das Datum kommt beim Schreiben drin und wenn dann alter Eintrag steht, der wird dann einfach überschrieben. Und gleichzeitig im Schritt 3 überschreibt der Hauptspeicher das alte Datum, also es wird in jedem Schreibprozess sowohl im Cache als auch im Hauptspeicher aktualisiert und die CPU muss halt warten, bis dieser Schreibvorgang abgeschlossen ist. Das ist eine relativ einfache Strategie, ich sage einfach, ok, schreiben passiert einfach immer in beiden, also sowohl im Cache als auch im Hauptspeicher. dass das natürlich zwar eine sehr einfache Strategie ist, die ist auch hardwaretechnisch einfach zu machen, allerdings ist sie nicht so ganz performant, weil ja jedes Mal der Prozessor warten muss, bis der Schreibvorgang im Hauptspeicher abgeschlossen ist. Man kann das ein bisschen abfedern, indem man die Strategie leicht verändert, dass man also ein Buffered Write-Through macht. Das heißt also, dass ich die Daten für den Hauptspeicher eben in den Puffer schreibe und sobald diese schreiben, der Puffer ist natürlich schneller als der Hauptspeicher kann die CPU weiterarbeiten und gleichzeitig, also nebenläufig, wird der Inhalt des Puffers in den Hauptspeicher übertragen. Das macht das Ganze ein bisschen schneller und behält doch noch die Einfachheit der Strategie bei. Ein bisschen aufwendiger als Strategie ist das sogenannte Zurückkopieren, Write Back. Und was mache ich? Ich brauche eine Änderungsmarkierung. ein Bit im Cache für jeden Eintrag und dieses Bit heißt Dirty Bit und dieses Bit, das zeigt mir an, ob jetzt ein Datum im Cache-Block modifiziert wurde oder ob es eben nicht modifiziert wurde, vorher schon. Und wenn immer ich natürlich einen modifizierten Block ersetzen will, dann weiß ich, okay, dieser Block hat sich im Cache verändert und ziemlich sicher ist diese erstmal das Datum im Hauptspeicher aktualisieren und kann dann erst den sogenannten Dirty Block ersetzen. Das heißt unser Algorithmus für diese Vorgehensweise ist wieder Schritt 1, also die CPU möchte praktisch das Datum 3, die Adresse 3 verwenden um das Datum B in den Cache zu schreiben. Das heißt die CPU legt 3 als Adresse auf dem Bus und B als Datum auf dem Bus. So, wenn ich jetzt habe, dann wird das Datum mit B überschrieben und B wird gesetzt und Ende des Schreibprozesses. Fertig dann. Also ich überschreibe einfach praktisch im Cache den Block, der da vorher drin gestanden ist. Wenn ich jetzt einen Cache miss habe, dann wird ein neuer Eintrag mit drin und wenn das Dirty Bit nicht gesetzt ist, dann wird der dort befindliche Eintrag einfach überschrieben. Das heißt wenn hier vorher war 6yd also Text 6, Inhalt yd, das wird einfach überschrieben im Fall eines Cache Miss. So wenn ich jetzt aber ein Cache Miss habe mit einem ich den dort befindlichen Eintrag nicht einfach überschreiben. Das heißt, ich habe hier einen Cache Miss, ich möchte hier einen Eintrag machen, hier finde ich das gesetzte Dirty Bit, das heißt also dieser Block 6 wurde verändert, ich muss also diesen Block 6 erst in den Hauptspeicher zurückschreiben und dann habe ich praktisch Platz, um an dieser Stelle das neue

Tag 3 zu schreiben, mit dem neuen Datum zu schreiben. Das heißt also, mit der Adresse 6 und das Datum Y erstmal in den Hauptspeicher zurück kopiert wird, Schritt 3C. Im Schritt 4C wird eben der Hauptspeicher aktualisiert, das heißt dieses Y wird zurückgeschrieben in den Hauptspeicher und jetzt kann die Cache-Logik praktisch im Cache diesen Block hier ersetzen, den vorher Block 6 ersetzen durch den Block 3 und damit ist der Schreibprozess beendet. führt natürlich dazu, also die Strategie Zurückkopieren oder Write Back führt natürlich dazu, dass zeitweise eine Inkonsistenz herrscht, weil eben eine Zeit lang im Cache unter dieser Adresse 6 ein anderes Datum steht als im Hauptspeicher. Also der Cache ist eben aktuell und der Hauptspeicher ist noch nicht aktualisiert und deshalb muss ich halt immer aufpassen, dass wenn ich diesen Block hier verdränge, dass der Hauptspeicher aktualisiert wird. Problem muss die Cache-Logik lösen. Es löst nicht der User, sondern das wird auf Hardware-Ebene gelöst. Dafür muss ich mir eine Logik bauen, die das Ganze vollautomatisch vornimmt. So, jetzt vergleichen wir mal die beiden Stripe-Strategien miteinander. Wir haben also die Write-Through-Strategie, haben wir ja schon gesagt, die ist relativ einfach und hat den Vorteil, dass Cache und Hauptspeicher zu jeder Zeit konsistent sind. Das ist insbesondere bei Multiprozessor-Systemen natürlich von großem Vorteil. andererseits wird eine hohe buslast erzeugt warum weil jedes schreiben auch in den hauptspeicher geht und deshalb der speicher bus für jedes schreiben belastet anders bei der anderen strategie schreibe ich erstmal nur in den cache und es geht natürlich deutlich schneller das heißt die der hauptspeicher bus ist natürlich deutlich weniger belastet so wie sieht es jetzt aus mit anderen strategien mit der ride back strategie zurück kopieren da ist jetzt die klar weil der hauptspeicher bus nicht für jeden schreibzugriff genutzt wird auch das ist natürlich wichtig für multiprozess systeme und sie sehen schon wieder wir machen hier so ein kleines spannungsfeld auf wir haben praktisch zwei kriterien für die leistung beide kriterien sind wichtig aber die beiden kriterien sind eben in den unterschiedlichen strategien und ich kann nur eine dieser beiden strategien verwenden also muss ich jetzt wieder der mir hier geboten wird wichtiger ist der eine oder andere so okay writeback ist natürlich schneller als writethrough ja weil ich eben nicht warten muss bis das datum im hauptspeicher aktualisiert ist sondern nur warten muss bis es im cache aktualisiert ist aber es ist natürlich technisch deutlich anspruchsvoller man kriegt eine kleine ahnung davon dass es anspruchsvoller ist wenn man sich die algorithmen betrachtet ja der algorithmus für writethrough ist einfacher als der Algorithmus für Writeback und diesen Algorithmus muss ich ja in Hardware realisieren und dementsprechend kann man sich glaube ich einfach vorstellen, dass die Hardware hierfür ein bisschen deutlich anspruchsvoller ausfällt. Gut, welche ist letztendlich die beste Strategie? Das muss ich für meine Zwecke entscheiden, indem ich simuliere, indem ich Benchmarke, die unterschiedlichen Strategien miteinander vergleiche und dann die für mich bessere nehme. was man nicht so ganz allgemein entscheiden kann was jetzt die beste lösung ist das ist wo packe ich den cache jetzt eigentlich hin ja packe ich den cache zwischen mmu und hauptspeicher so wie das hier unten gezeigt ist oder packe ich den cache praktisch direkt nah an den prozessor und Was ist der Unterschied zwischen den beiden Sachen eigentlich? Naja, ganz

einfach, es geht um die Adressierung. Prozessorintern arbeite ich immer mit virtuellen Adressen. Im Hauptspeicher arbeite ich immer mit realen Adressen. Und irgendwo muss ja die Umrechnung zwischen virtuellen Adressen und realen Adressen stattfinden. Und diese Umrechnung, genau das macht die Memory Management Unit, MMU. haben will ja und dann natürlich auf dem weg zum hauptspeicher aus den virtuellen adressen reale adressen mache das ist also hier spricht man vom virtuell adressierten cache und die mmu sitzt dann genau zwischen cache und hauptspeicher und hier unten da spreche ich jetzt vom real adressierten cache warum weil direkt auf dem weg zum cache schon die mmu dazwischen sitzt und eben reale Cache-Adressen umrechnet und die große Frage ist jetzt wieder, welche dieser beiden Ansätze ist der beste. Gut, schauen wir uns mal die Vor- und Nachteile der beiden Ansätze an. Zuerst mal den virtuell adressierten Cache. Die Lösung ist schnell, weil die Cache-Adresse unabhängig von der Adressübersetzung ist. Also virtuell adressierter Cache, da bin ich hier oben, ein bisschen braucht um die adressen umzurechnen zur verzögerung sagt der nachteil ist klar die virtuellen adressen sind in der regel deutlich länger als die realen adressen sie erinnern sich grundlagenregenarchitektur da haben wir das besprochen wie man aus virtuelle adressen reale adressen macht und dementsprechend habe ich natürlich größere also brauche ich mehr speicherplatz im cache weil ganz einfach meine adressen länger sind als die die so, der nächste Problem ist, wenn jetzt jeder Prozess seinen eigenen virtuellen Adressraum besitzt ja, dann muss ich natürlich dafür sorgen, dass die Daten, die im Cache stehen auch erstmal in den Hauptspeicher geschrieben werden, wenn ein Prozesswechsel stattfindet das heißt, ich muss bei jedem Prozesswechsel einen Cache-Flush machen Das kann ich natürlich vermeiden, indem ich meine Cache-Adressen noch ein bisschen länger mache Das heißt, außer dass ich hier mit virtuellen Adressen arbeite, hafte ich noch so ein Tag mit an, das die PID beinhaltet So dass dann praktisch jeder Prozess dann im Cache seinen eigenen virtuellen Adressraum hat Das ist erstmal eine gute Lösung, braucht zwar ein bisschen Speicherplatz, aber Speicher ist ja nicht mehr ganz so teuer wie früher, kann man sich leisten Problem, weil ich ab und zu mit Kommunikationsvariablen arbeite, also mit Semaphore-Variablen zum Beispiel und die werden dann auf diese Art und Weise zu unterschiedlichen virtuellen Adressen gemappt, obwohl die im Hauptspeicher an der gleichen virtuellen Adresse liegen. Das heißt, da habe ich jetzt ein neues Problem, was ich dann irgendwie hardware-technisch natürlich zum beispiel von der festplatte außen da die memory access fahren also in dem erfahren und wenn ich jetzt mir von der festplatte daten in den hauptspeicher hole ja dann kriegt der cash in der regel davon nichts mit und jetzt habe ich hier natürlich andere daten als hier weil ich ja quasi den cash umgangen habe und über die memory access mir meinen hauptspeicher inhalt eben der Hauptspeicher konsistent bleibt. So, das sind also Vor- und Nachteile, wenn ich mit dem virtuell adressierten Cache arbeite. Auf dieser Folie, das sind wir jetzt beim real adressierten Cache, also Anordnung ist CPU, dann kommt die Memory Management Unit, dann kommt der Cache, dann kommt der Hauptspeicher. Und hier ist das Sicherstellen der Kohärenz einfacher, weil eben keine Rückübersetzung nötig ist, also Cache und Hauptspeicher sprechen quasi die gleiche Sprache. und der real adressierte Cache

hat sich in der Praxis als günstiger für Multiprozessor Systeme erwiesen, eben aufgrund dieser Kohärenzproblematik, dass die einfacher zu lösen ist. Allerdings, was ich jetzt brauche, ist natürlich eine sehr schnelle Übersetzung von virtuellen in reale Adressen, weil die Memory Management Unit ja jetzt zwischen CPU und dem sehr schnellen Cache sitzt zu beschleunigen und hierfür gibt es einen Trick bzw. damit es überhaupt funktioniert, hat man den Translation Look Aside Buffer eingeführt. Dieser Translation Look Aside Buffer, der soll hier nur kurz behandelt werden, der wird ausführlich behandelt in der Vorlagen-Grundlagen-Betriebssysteme. Aber nächste Folie gehen wir ganz kurz darauf ein, wie das funktioniert. indem ich parallel zur Adressübersetzung schon mal einen Cache-Zugriff mache und mit Offset-Bit arbeite, die ich nicht übersetzen muss und gleichzeitig dazu die Tag-Bits übersetze. Schauen wir uns einfach mal das Beispiel an für die Translation Look-Aside-Buffer, wie das funktioniert. Also ich komme hier vom Prozessor mit einer virtuellen Adresse und diese virtuelle Adresse, die hat natürlich eine virtuelle Seitenadresse und der normale Weg ist der blaue Weg. dass die Memory Management Unit aus dieser virtuellen Seitenadresse eine reale Seitenadresse macht und über diese reale Seitenadresse kann ich dann im Hauptspeicher auf die reale Adresse zugreifen. Also der blaue Weg, das wäre der normale Weg. Was wir jetzt als Abkürzung dazwischen bauen, ist der sogenannte Translation Look-Aside Buffer. Das heißt, ich merke mir einfach bei den Zugriffen auf meinen Cache, und schreibe in einer kleinen Tabelle da rein die Seite, die ich mir geholt habe und welche virtuelle und welche reale Adresse gehört dazu. Ich baue mir also im Betrieb eine kleine Tabelle auf und die Tabelle hier ist natürlich wesentlich kleiner als die große Seitentabelle, mit der die Memory Management Unit arbeiten muss. Dann baue ich mir eine schnelle Hardware für eine assoziative Suche. und wenn also jetzt ein Zugriff gestartet wird von der Wettbewerbsadresse pd, dann läuft beides gleichzeitig an. Also das wird erstmal an die MMU geschickt, die MMU schaut schon mal nach, ob sie hier übersetzen muss und gleichzeitig schaue ich im Translation Lookup Side Buffer. Dieser rote Weg, der ist deutlich schneller, weil diese Tabelle deutlich kleiner ist und ich hier eine schnelle assoziative Suche implementiert habe dann breche ich den Weg hier ab. Dann gehe ich also diesen schnellen Weg und habe damit eine sehr schnelle Übersetzung von virtueller Adresse in reale Adresse. Das heißt also, ich mache eine kombinierte Adressübersetzung, sowohl der Translation Look-Aside Buffer wird durchsucht, als auch die Page Table und falls ich eben im Translation Look-Aside Buffer erfolgreich bin, Nun wollen wir ein bisschen näher nochmal betrachten unser Cache-Con-Kohärenz-Problem. Wir haben vorhin gesagt, dass sowohl die CPU als auch eine I.O. Einheit, also zum Beispiel eine Festplatte über DMA, also Direct Memory Access, können auf den Hauptspeicher zugreifen. Das heißt also, ich kann über CPU auf das Datum zugreifen und ich kann auf diesem Weg über Direct Memory Access auf dieses Datum zugreifen. führen dass ich hier ein anderes datum stehen habe als hier das heißt cache und hauptspeicher können unter gleicher adresse unterschiedliche daten enthalten und das ist natürlich immer schlecht das heißt was wenn wir uns jetzt mal vorstellen was passieren kann erstmal arbeitet die cpu mit dem cache und überschreibt das datum b adresse 3 im cache und

mit dem writeback methode hier lediglich das dirty bit gesetzt. Bei writeback schreibe ich ja nur im cache und erst wenn dieser block verdrängt wird, wird der hauptspeicher aktualisiert. so der zweite schritt der startet ist eben eine dma task wird ausgeführt und die führt auch dazu dass an der gleichen adresse 3 auf die platte geschrieben wird und jetzt haben wir das problem wenn eben nur der hauptspeicher weil es am Cache vorbei aktualisiert wurde So und was ist jetzt dann der Unterschied zwischen Kohärenz und Konsistenz? Kohärenz heißt, das ist die Eigenschaft eines Speicherverwaltungssystems welches garantiert, dass der Lesezugriff auf ein Wort immer den Wert des letzten Schreibzugriffes auf dieses Wort zurückgibt Das heißt immer dann, wenn der Prozessor ein Wort liest, bekommt er das aktuellste Datum Das stelle ich mit Kohärenz einfach sicher dann ist diese Eigenschaft der Fall. Das heißt, ein Lesen liefert mir immer wirklich das aktuelle Datum. So, diese Forderung ist nicht ganz so hart wie die Konsistenz. Weil Konsistenz heißt, das ist der Systemzustand, in dem alle Kopien eines Wortes im Hauptspeicher und Cache identisch sind. Also Konsistenz ist eine Nummer härter. die wir vorher hatten bei der Write-Back-Strategie praktisch. Hier haben wir auf der Folie vorher die entsprechende Strategie, die zu dem fehlerhaften Zustand führt. Und deshalb ist es so, dass ich mit so einer Strategie arbeiten kann, die nicht immer konsistent ist, aber sie muss auf jeden Fall kohärent sein. Gut, also bei der Konsistenz müssen alle Kopien immer identisch sein. Das ist bei der Kohärenz nicht zwingend gefordert. lese, dass ich dann wirklich das aktuelle Datum bekomme. Das heißt, wenn ich konsistent bin, bin ich auf jeden Fall kohärent. Also Konsistenz ist ein bisschen härter als Kohärenz, dementsprechend impliziert die Konsistenz die Kohärenz. Und die Kohärenz, die kann oder muss zwangsläufig natürlich vorübergehend inkonsistente Zustände akzeptieren. Also ich kann inkonsistent werden, kein Problem, solange das nicht zum fehlerhaften Systemzustand ich eben nicht falsche Werte verarbeite. Das muss ich sicherstellen, Minimum. Weil sonst habe ich ein Problem, sonst habe ich einen fehlerhaften Systemzustand. Ich muss mir also ein Protokoll einfallen lassen, das mir auf jeden Fall die Kohärenz sicherstellt. Und was mache ich? Ich beobachte einfach alle Cache-Zugriffe, ich mache also ein Monitoring, ich beobachte alle Cache-Zugriffe und reagiere entsprechend. jetzt zwei unterschiedliche Methoden, prinzipiell unterschiedliche Methoden. Ich kann diese Beobachtungen in einem zentralen Verzeichnis verwalten und in diesem Verzeichnis da steht es dann praktisch drin, Informationen über Standort und Zustand aller Kopien, mit denen ich in den Caches arbeite. Also Sie können sich vorstellen, dass dieses zentralisierte Verzeichnis natürlich schon Informationen über alle Cache-Kopien da drin überwacht werden. Die andere Methode ist die sogenannte Snoop-Method. Hier höre ich ständig den Bus ab und schaue praktisch, welche Daten laufen über den Bus, welche Cache-Blöcke laufen über den Bus, weiß Bescheid, wie praktisch die Cache ist, welcher Inhalt da drin ist und durch geeignete Maßnahmen stelle ich sicher, dass die Konsistenz erhalten bleibt. entweder invalidieren oder in den Hauptspeicher zurückschreiben. Ich muss also aufgrund des Protokolls Maßnahmen anwerfen, die dafür sorgen, dass die Konsistenz erhalten bleibt. So, wir werden gleich Beispiele dazu machen über beide Methoden, also über das Directory Method und über das Snoop

Method, wie ich sowas in Hardware dann auch realisieren kann. Vorher noch ein paar Begriffe. Also es geht jetzt um Cache-Kohärenz-Protokolle Es geht eben darum, dass die Caches und der Hauptspeicher organisiert aktualisiert werden. Das heißt, dass die Kohärenz auf jeden Fall erhalten bleibt. Und es gibt dafür einmal ein Write-Update-Protokoll. Das heißt, in dem Moment, wo ich eine Kopie neu beschreibe, also ich ein Write mache, werden alle anderen Kopien aktualisiert auf jeden Fall. auf jeden Fall bei jedem Schreiben ausgelöst. Dann spricht man vom Write-Update-Protokoll. Das ist eine Möglichkeit. Die andere Möglichkeit, die ich habe, ist, dass ich einfach, wenn ich eine Kopie verändere, alle anderen für ungültig erkläre. Dass also immer die letzte Änderung nur erhalten bleibt. Und dann spricht man vom Write-Invalidate-Protokoll. Das heißt, vor einer Veränderung einer Kopie werden alle anderen Kopien außer Kraft gesetzt. Also zwei Strategien. Ich aktualisiere alle Kopien, dann Write Update oder ich sage, okay, alle anderen Kopien sind ab jetzt ungültig, weil ich eben einen neuen Wert da reinschreibe. Dann ist es Write Invalidate. So, bevor wir uns jetzt die Protokolle im Genauen anschauen, müssen wir uns einfach über Zustände und verwendete Abkürzungen nochmal kurz unterhalten. also als endlichen Automaten beschreiben und Sie wissen ja, endliche Automaten brauchen einerseits Zustände und andererseits Zustandsübergänge und dieser endliche Automat, der soll die Zustände von Cache-Blöcken reflektieren und so ein Cache-Block, der kann jetzt drei unterschiedliche Zustände haben. Der eine Zustand ist der Zustand U, U steht für Uncached, das heißt kein Prozessor verfügt über den Cache-Block derzeit nur im Hauptspeicher und der Hauptspeicher ist natürlich aktuell, ist klar. Dann gibt es die Möglichkeit des Zustands Shared, abgekürzt mit S. Shared heißt, dass eben dieser Cache Block A sich befindet in einem oder mehreren Caches von den Prozessoren. Der Hauptspeicher ist aktuell und natürlich sind alle Kopien des Cache Blocks A konsistent. Das ist also, die haben alle den gleichen Wert, alle Kopien, dann ist er Shared. EM, EM steht für Exclusive Modified, das heißt ein Prozessor verfügt über diesen Cache Block und hat diesen Cache Block auch verändert und der Hauptspeicher ist nicht aktualisiert. Und das Problem, das wir jetzt haben bei unserer Directory Method, dass das Directory jetzt Zustand befindet. Also Exclusive ist ja ein bisschen schwächer. Exclusive heißt, ich habe den nur alleine und das Direktere bekommt nicht mit, wenn ich den verändere. Und deshalb wird es zusammengefasst zu einem Zustand, also Exclusive Modified. Ich gehe mal Worst Case davon aus, dass ich diesen Block schon verändert habe. Gut, weil einfach dieser Zustandsübergang von Exclusive nach Modified, das passiert ja nur im eigenen Cache und geht nicht über den Systembus. kommt das Directory davon nichts mit. Okay, dann haben wir unsere Abkürzungen für Zugriffe auf die Cache-Blöcke. Da haben wir einfach ein ganz normales Read. Und Read heißt, dass bei einem Read-Miss im Cache der Prozessor eine Read-Anforderung auf den Bus sendet und auf den angeforderten Cache-Block aus dem Hauptspeicher wartet. Also Read-Hit ist klar, wenn ich den Cache-Block schon habe, dann lese ich den natürlich direkt aus meinem Cache. ich ein read miss habe dann fordere ich den block vom hauptspeicher und bekomme den dann auch vom hauptspeicher readx ist ein write miss im cache dann wird eine readx anforderung auf den bus

geschickt und auch hier wird natürlich der angeforderte cacheblock übermittelt vom hauptspeicher an den prozessor und da ich aber darauf schreibe muss der natürlich in allen anderen caches werden alle anderen Kopien invalidiert mit einem Invalidate-Signal. Und dann gibt es noch Upgrade. Beim Upgrade ist es so, ich finde, ich habe einen Bright Hit in meinem eigenen Cache, das heißt, ich finde den Blog, den ich schreiben will, im eigenen Cache und muss dann natürlich, wenn ich den Blog jetzt verändere durch Schreiben, muss ich wieder alle anderen Kopien invalidieren. Das heißt, ich muss auch wieder so ein Invalidate ausführen. endlichen Automaten, also in eine Finite State Machine überführen. Das heißt, wir haben hier unsere drei Zustände U, E, M und S und wir haben dann die Zustandsübergänge durch diese Pfeile hier dargestellt und die müssen wir natürlich beschriften entsprechend mit den Anforderungen. Und letztendlich ist dieser Graph hier nichts anderes als eine Übersetzung von dieser heißt ich schaue immer in welchem Zustand bin ich, da habe ich drei Möglichkeiten, U, EM und S und dann schaue ich mir wie wirken die Anforderungen, die ich machen kann, ich kann also ein Read oder ReadX vorliegen haben, wenn ich in U bin, was passiert, wenn ich in U bin, dann bin ich ja direkt auf dem Hauptspeicher und jede Anforderung, die auf den Hauptspeicher ab und wenn nur einer erstmal den Block holt, dann befindet sich dieser Block im Exclusive Modified Zustand. So, das heißt der Folgezustand hier ist immer EM. Man sieht es auch, es gibt nur einen Pfeil von hier nach hier und das beschreibt mir dann eben den Folgezustand. So, jetzt ausgehend von diesem Zustand EM, da habe ich jetzt mehrere Möglichkeiten. es kommt ein Read und wenn dann ein Read kommt, dann muss ich das jetzt natürlich, habe ich mitbekommen, ein anderer will eben auch diesen Blog. Also ich bin jetzt da nicht mehr der exklusive Eigentümer dieses Blogs, sondern der andere will den Blog. Dementsprechend bekommt der andere diesen Blog auch geliefert und der Zustand wechselt jetzt natürlich dieses Blogs von Exclusive Modified zu Shared. nur einen Prozessor zugeordnet, sondern mehrere Prozessoren zugeordnet. Dementsprechend habe ich hier einen Folgezustand Shared und jetzt habe ich eben den Cache-Block entsprechend im Folgezustand Shared. So, wenn ich jetzt diesen Block schreibe, wenn ich also ein Read-X auf diesen Block mache und bin im Exclusive-Modified-Zustand, dann muss ich natürlich alle anderen Kopien invalidieren Blog wird geschrieben und es gibt in dem Sinn keinen Folgezustand, weil eben die Invalidierung auch an den Eigentümer geschickt wird. So, wenn ich jetzt im Zustand Shared bin und habe ein Read, dann wird ganz einfach die operation ändert ja nicht an der konsistenz oder kohärenz das heißt der block wird ausgeliefert und ich bleibe in dem zustand das ist genau wie bei dem exclusive modified bleibe ich ja auch mit dem readx in diesem zustand ich löse ja das problem dass ich hier verändere indem ich alle anderen kopien invalidieren das heißt ich kann hier munter darauf schreiben der zustand ändert sich nicht solange es kein anderer prozessor macht solange ich das mache bleibt es bei exclusive zustand genau lesen im schert ist kein problem schreiben im schert ist natürlich ein problem das heißt wenn jemand auf dem schert blog schreiben will ja dann bekommt er diesen schert blog zum schreiben allerdings ändert sich dann der zustand zu exclusive modified weil eben der blog neu beschrieben wird und alle kopien die derzeit existieren die

werden invalidiert alle Share invalidiert und dem Anforderer auf diesen Blog wird gesagt, okay, er kann aktualisieren und das wird eben durch ein Reply-Signal gemacht. Hier unten ist nochmal die Legende, also Read ist ein Read-Miss im eigenen Cache, ReadX ist ein Write-Miss im eigenen Cache und ReplyD, das heißt, die Daten werden geschickt, also ReplyData, da wird ein Blog gesendet und Reply ist einfach so ein Signal, das da gesendet wird. Ich habe jetzt praktisch mein Cache-Kohärenz-Protokoll in die Fine-It-State-Machine gepackt und Sie erinnern sich vielleicht, dass das eigentlich immer das Hauptproblem ist, diese Fine-It-State-Machine zu entwickeln oder so eine Tabelle zu entwickeln, weil jetzt kann ich alle Eingänge und alle Ausgänge und alle Zustandsübergänge meiner Schaltung identifizieren, die praktisch dieses Protokoll dann zügig umsetzt. Das war also die Möglichkeit, indem ich so ein Direct-Rename baue, also so ein Verzeichnis, wo eben alles über die Cache-Blöcke drinsteht, also zentral verwaltetes Verzeichnis mit allen Informationen über alle Blöcke. Die andere Möglichkeit ist die verteilte Möglichkeit, ist für symmetrische multiprozessoren die geeignete das heißt wir haben hier speicher gekoppeltes system wir haben einen system bus also das ist ein bisschen lästig hier normalerweise drückt man control und dann kommt der laser und wenn man da eine zehntel sekunde versetzt drückt dann kommt die nächste folie also da ist powerpoint noch ein bisschen verbesserungswürdig egal Ein Systembus, ein global geteilter Speicher und wir greifen über verteilte Caches, das heißt jede CPU hat ihren eigenen Cache, greifen wir über den Systembus auf diesen Speicher zu. So, und zwischen Cache und Systembus, da hängen wir jetzt diesen Snooper und der Snooper, der macht ein Monitoring, also eine Aufzeichnung von allem, was da jetzt passiert, was da abgeht zwischen Caches und Speicher. Das heißt, der Snooper vergleicht die Adressen auf dem Bus mit den Adressen im Cache und stellt eben fest, okay, es passiert ein externer Schreibzugriff, was muss ich dann tun? Die lokale Kopie wird ungültig, weil jemand anders aktualisiert ein Datum, das bei mir im Cache liegt. Also muss ich das natürlich als ungültig deklarieren. Und wenn ich einen externen Schreib- oder Lesezugriff habe und die lokale Kopie verändert wurde, muss ich natürlich erstmal dafür sorgen, dass der Hauptspeicher aktualisiert wird. Das heißt, die lokale Kopie wird erstmal in den Hauptspeicher zurückgeschrieben und dann kann sie von dem anderen entsprechend aus dem Hauptspeicher geholt werden. Das ist das Protokoll, das ich abarbeiten muss. Wenn ich jetzt so ein Multiprozessor-System habe, dann ist das Ganze natürlich ein bisschen umfangreicher und deshalb hat man sich dann Gedanken gemacht, wie man das jetzt am besten lösen kann das sogenannte MESI-Protokoll erfunden. MESI, die Abkürzung, kommt von den vier Zuständen, mit denen ich arbeite. Und dieses MESI-Protokoll, das wollen wir uns jetzt mal ein bisschen genauer anschauen. Wir haben also für jeden Cache-Block vier Zustände, zweimal zwei Zustände. Und zwar habe ich hier eine kleine Tabelle, da haben wir zeilenmäßig Modified und Unmodified stehen und spaltmäßig Exclusive und Shared. jetzt genau diese vier zustände es gibt den zustand exclusive modified abgekürzt mit m und es gibt den zustand exclusive unmodified abgekürzt mit e es gibt den zustand invalid und das heißt das datum ist shared aber als invalide deklariert worden und den zustand Also Zeilen- und Spaltbeschriftung ist klar, Exclusive

heißt, Blog befindet sich nur in diesem Cache, Shared heißt, Blog befindet sich in mehreren Caches, Modified heißt, der Blog wurde eben überschrieben und Unmodified heißt, der Blog wurde nur gelesen, aber nicht überschrieben. So, und außer diesen vier Zuständen mäßig, ja, kann man hier schön lesen, gibt es noch Signale für den Snooper, das heißt, es gibt ein Invalidate-Signal, Dann gibt es ein Shared-Signal, das heißt ein Cache-Block ist ab sofort Shared. Und dann gibt es ein Retry-Signal, das heißt der Cache-Block wird gerade geschrieben. Aber ich will praktisch die neueste Kopie des Cache-Blocks, deshalb muss ich abwarten, bis er im Hauptspeicher aktualisiert ist. Und dementsprechend muss ich dann nochmal zugreifen, wenn das Datum im Hauptspeicher aktualisiert ist. Das ist also dieses Retry. Ich will die neueste Kopie, aber die befindet sich an einem Cache, also muss die erst in den Hauptspeicher und dann muss ich warten, bis sie im Hauptspeicher ist und dann kann ich einen Retry machen, also nochmal drauf zugreifen. Machen wir mal ein einfaches Beispiel dafür, ein MESI-Protokoll-Beispiel. Wir haben also jetzt einen symmetrischen Multiprozessor mit zwei Prozessoren. Wir sprechen genau von diesen MESI-Zuständen, also Exclusive Modified, Exclusive Unmodified, Shared Unmodified, Invalid. wir haben hier den Hauptspeicher wir haben hier die Caches für den Prozessor 1 die Caches für den Prozessor 2 und hier im Kommentar steht was wir gerade tun wollen ok im ersten Schritt fordert der Cache vom Prozessor 1 das Datum A an das heißt also in diesem Schritt der Cache Block eingelagert in den Prozessor 1 mit dem Datum A über diesen Cashblock verfügt, bekommt dieser Cashblock natürlich den Zustand Exclusive Unmodified, also ein E. So jetzt fordert der P2 natürlich das gleiche Datum an. So der P2 bekommt auch dieses Datum A, allerdings ist der jetzt natürlich nicht mehr Exclusive, sondern das Datum A wird für beide Caches, also Jetzt schreibt der Prozessor 1 auf das Datum A im Cache. Das heißt, aus A wird natürlich A'. Das heißt auch, dass das natürlich jetzt Exclusive Modified wird, weil er verändert dieses Datum ja. Und damit wird natürlich die Kopie, die hier im Prozessor 2 liegt, als Invalide erklärt. Das heißt, ich habe also mehrere Aktionen. von A in Cache P2 und Cache P1 sendet Retry an den Cache P2 und Cache P2 wartet. So, aber der, äh, okay, also der Kommentar hier, der ist ein bisschen Quatsch, den müssen wir nochmal überarbeiten, sorry, neue Folie, der Kommentar ist Quatsch, was passiert ist ganz einfach, A geschrieben wird in P1, dann wird A in P2 als Invalide erklärt. So, ja genau, das ist diese Zeile hier. Was hier steht, was ich vorgelesen habe, das bezieht sich jetzt auf die nächste Aktion. Die nächste Aktion ist nämlich, dass das Datum A im Cache P2 jetzt neu gelesen werden soll. einen Zustand modified vom Datum A im Cache P1 und der Cache P2 fordert jetzt das Datum A neu an. So, was passiert? A ist ja nicht mehr aktuell, sondern das ist ja A', aber A' liegt im Cache von P1. Und deshalb bekommt der Cache P1 es mit, dass jetzt der P2 wieder auf A zugreifen will. Und der sagt, okay, momentan habe ich das Datum, aber ich habe es verändert. Das heißt, du musst ein bisschen warten. Das ist dieses Retry. Also der Cache P2 wartet. Was Cache jetzt P1 macht, er schreibt das Datum in den Hauptspeicher zurück. Macht also ein Shared draus und schickt es in den Hauptspeicher zurück. Und wenn es im Hauptspeicher aktualisiert ist, dann kann der Cache P2 das aus dem Hauptspeicher holen

und holt sich also das Datum A'. Das heißt also, wir haben jetzt das Datum wieder Shared und beide verfügen jetzt über die aktuelle Kopie. Es ist immer ein bisschen leicht verwirrend. Was mich jetzt verwirrt hat, ist, ich muss das hier noch in gleicher Farbe machen. Also diese Aktion hier, das ist quasi eine Aktion. Lesen von Datum A in Cache B2. Und das sind die vier Teilschritte, die ich machen muss, um diese Aktion zu realisieren. Und das Ergebnis dieser Aktion ist, dass das Datum dann wirklich in beiden Caches liegt. Gut, man kann also schon bei den ganzen Aktionen ein bisschen den Überblick verlieren. Und deshalb muss ich das wieder in eine Finanztate Machine packen. wo dann alle Zustände und alle Zustandsübergänge praktisch drin sind. Das ist also jetzt der vollständige, also nicht erschreckend, das schaut jetzt natürlich ein bisschen unübersichtlich aus, das ist der vollständige MESI-Zustandsübergangskraft für die Cache-Blöcke. Wir werden im Einzelnen die Zustände auf den nächsten Folien einfach nochmal durchgehen. Klar ist, glaube ich, MESI, die vier Zustände, die wir haben, unmodified und shared unmodified. Das sind unsere vier Zustände, zwischen denen wir hin und her operieren. Wir haben ein Dirty Line Copyback, wo ich praktisch dann den Hauptspeicher aktualisiere. Wir haben eine Invalidate Transaction, wo ich Blogger als Invalide deklariere. Wir haben ein Read with Intent to Modify als Aktion. Also ich möchte gern schreiben, vorher muss ich es natürlich lesen. und wir haben eine Cache Line Fill Operation. So, und welche Aktionen führen wir auf den Caches aus? Also die harmlosen Aktionen sind immer die Leseaktionen, also Read Hit, ein Read Miss auf Shared oder ein Read Miss auf Exclusive, dann haben wir ein Write Hit, dann haben wir ein Write Miss und dann haben wir ein Snoop Hit on a Read und ein Snoop Hit on a Write oder Read with Intent to Modify. Und diese Aktionen hier befinden sich natürlich als Beschriftung auf den entsprechenden Kanten meiner Finite State Machine. Gehen wir jetzt einfach mal die einzelnen Zustände nacheinander durch und die Aktionen, die in diesen Zuständen auftreten können und was im Protokoll dann abgearbeitet werden muss. Das heißt, wir sind jetzt erstmal im Zustand hier unten, Exclusive Modified. Cache und Modified er wurde nach dem Laden verändert. Also ich habe quasi eine Inkonsistenz zwischen Hauptspeicher und Cache, weil ich eben ein Datum verändert habe, aber exclusive nur hier verändert habe. So was passiert also wenn ich jetzt einen lokalen Lese- und Schreibzugriff habe, also ein Read-Hit oder ein Write-Hit, da passiert natürlich überhaupt nichts. Ich bleibe in diesem Zustand. Ich habe die letzte aktuelle Kopie, die kann ich natürlich als Modified gekennzeichnet ist. Also da, das ist das Einfachste, da passiert gar nichts. Der Zustand bleibt unverändert und der Bus bekommt auch von diesen Aktionen nichts mit. Also diese Aktionen, die werden nicht nach außen getragen, weil der Block eh schon als Modified deklariert ist. So, jetzt meldet mir mein Snooper einen erschnüffelten Lesezugriff in einem anderen Cache. Also ein anderer Cache will darauf zugreifen. und der andere Cache braucht natürlich das aktuelle Datum. Das heißt, erstmal müssen wir mit SHR, also genau mit diesem Pfeil hier, den Zustand wechseln, nämlich nach Shared Unmodified. Der laufende Lesezugriff im anderen Cache, der muss unterbrochen werden, weil die aktuelle Kopie liegt bei mir. Ich muss erstmal diesen Cache-Block in den Hauptspeicher zurückschreiben, das ist dieser Pfeil

nach unten, über das Retry-Signal und das Shared-Signal dann den Zeitpunkt, an dem er jetzt den im Hauptspeicher aktualisierten Block praktisch sich holen kann. Gut, wenn ich einen erschnüffelten Schreibzugriff habe, also hier haben wir einen erschnüffelten Lesezugriff, wenn ich einen erschnüffelten Schreibzugriff habe, dann muss ich natürlich zu Invalidate überwechseln, und zwar mit SAW wechsle ich hier zu Invalidate. Das heißt, erstmal wird der laufende Schreibzugriff unterbrochen, weil ich ja die exklusive Kopie habe. Ich schreibe das Ganze in den Hauptspeicher zurück und ich mache ein Retry-Signal als Signal an den anderen, dass er jetzt diesen Block verändern kann. Und alle anderen Kopien werden natürlich invalidiert. Das ist dann dieser Invalid-Übergang. So, was sind die Möglichkeiten, wenn ich im Explosive Unmodified bin? Explosive Unmodified heißt, den Cache-Block, den ich gerade betrachte, den gibt es nur als Kopie in meinem lokalen Cache. Also der ist nicht verändert. So, wenn ich einen Read-Hit drauf mache, ganz einfach, passiert natürlich gar nichts, weil Lesen verändert nicht, Konsistenz, Kohärenz kein Problem an dieser Stelle. hier munter lesen und bleibe in diesem Zustand. Wenn jetzt jemand anders, wenn ich diesen Cache Block jetzt verändere, dann, also ich habe einen Right Hit, dann wechsle ich natürlich von Exclusive Modified zu Exclusive Modified. Also ich bin immer noch der Einzige, der mit diesem Block arbeitet, aber ich habe diesen Block natürlich verändert und bin dementsprechend jetzt im neuen Zustand, Folgezustand, Exclusive Modified. mein Snooper meldet mir jemand anders, also ich bin in Exclusive Unmodified und mein Snooper meldet mir jemand anders, will diesen Blog lesen, dann gehe ich natürlich mit SHR, also Snoop Hit on Read, wechsle ich in den Zustand Shared Unmodified, es ist ja, Lesen führt ja nicht zur Modifizierung des Blogs, aber er ist nicht mehr Exclusive, sondern er ist jetzt Shared. So, und wenn es jetzt einen Schreibzugriff auf diesen Cash-Blog gibt, Dann habe ich also ein Snoop Hit on Write. Dann wird natürlich dieser Block invalidiert. Der wechselt also den Zustand Invalidate, weil ja jemand anders auf diesen Block schreibt. Ich habe ihn ja nicht modifiziert, also brauche ich den nicht in den Hauptspeicher zurückschreiben. Aber er wird natürlich invalidiert, weil jemand anders auf diesen Block schreibt. So, nächster Zustand. Ich befinde mich in Shared Unmodified. und da haben wir wieder ein Snoop-Hit on Read oder ein Read-Hit, da passiert wieder gar nichts, solange das Ding unmodifiziert ist, also unverändert ist, kann ich natürlich lesen und bleibe im gleichen Zustand, ändert sich also nichts. weil hier einer auf diesen Block geschrieben hat und ich muss natürlich alle anderen möglicherweise existierenden Kopien invalidieren. Das heißt, ich muss diese Invalidate Transaction auslösen, damit eben alle anderen Kopien invalidiert werden. anderen wieder invalidieren mache, also einen Zustandsübergang zu invalidate. Okay, letzter Zustand, den wir betrachten müssen, ich befinde mich im Zustand Invalide, das heißt, der betrachtete Cashblock ist entweder ungültig oder im lokalen Cash nicht vorhanden und wenn jetzt auf diesem Cashblock ein Lesezugriff stattfindet, also ein Read Miss Shared oder ein Read Miss Exclusive, also klar, es muss ein Miss sein, weil der eben nicht vorhanden ist oder ungültig ist, dann wechsle ich von Invalide über zu, je nachdem, Shared oder Exclusive, also mit dem Read Miss Shared wechsle ich natürlich zu Shared und mit dem Read Miss Exclusive

wende ich natürlich hier zu Exclusive Unmodified. Ich muss natürlich den Cashblock laden, das heißt ich muss einen Cashline Fill auslösen, der Cashblock Außenhauptspeicher erstmal geladen wird. Ist ja momentan nicht vorhanden oder ungültig. So, wenn ich auf diesen Cashblock jetzt schreibe, dann gehe ich natürlich von Invalid auf in den Zustand Exclusive Modified. Ich muss natürlich immer, wenn ich schreibe, alle anderen Kopien invalidieren und eben so ein Read with Intent to Modify hier auslösen, das sind also die vier Zustände des MESI-Protokolls resultieren in diesen doch relativ komplexen Automaten man muss sich einfach nochmal wirklich die einzelnen Zustände jeweils vornehmen und überlegen, was passiert genau mit diesen Cash-Blöcken und ich glaube, so eine Darstellung ist zwar komplex, aber immer noch schöner als irgendeine Tabelle mit Nullen und Einsen, die letztendlich aus diesem Zustandsübergang natürlich hergestellt wird und auf die dann die Logik die dann eben dieses MESI-Protokoll realisieren kann. Und bei der Cache-Größe sieht es jetzt eben so aus, dass wir Kontrollinformationen zusätzlich noch im Cache natürlich mit speichern müssen, bei den einzelnen Blöcken dabei. Das heißt, unsere Cache-Größe setzt sich zusammen aus einem Anteil für die Daten, die ich speichern muss, also Data Storage, plus die Tags, die ich natürlich brauche. Sie erinnern sich an das Mapping, Hauptspeicher Cache, Tags, die gespeichert werden müssen und dann natürlich irgendwelche Kontrollinformationen, die ich brauche, also zum Beispiel Dirty Bits und ähnliches. Das heißt, also die Größe für den Data Storage ist einfach die Anzahl der Cache-Blöcke, SCB ist die Anzahl der Cache-Blöcke mal die Blockgröße, dann haben wir natürlich für jeden Cache-Block haben wir so einen Tag, was wir mitspeichern müssen und für jeden die wir mitspeichern müssen. Und für den Data Storage ist es normalerweise so, dass ich dafür sehr schnelle SRAMs benutze, also statische Speicherbauteile, die eben deutlich schneller sind als DRAMs. Und der Tag- und Control-Storage, das sind extra Module, die eben auch entsprechenden Zugriff erlauben. Und die Technologie, heutzutage hohe Integration, erlaubt es natürlich, dass ich alles auf einem Chip integriere, bzw. für Level 1 Cache ist das Ganze ja auf dem Prozessorchip integriert. Die Performance ist klar, aufgrund von Caches erhöhe ich natürlich meine Performance, aber warum erhöhe ich die Performance? Naja, wir wissen, dass die Ausführungszeit ist Anzahl der Befehle mal CPI mal Taktzeit und CPI Strich hier, das ist das CPI plus Memory Delay letztendlich senkt der Cash entweder die Taktzeit, ich kann also schneller takten, oder das Memory Delay wird natürlich kleiner, weil Weight States gespart werden. Allerdings, wenn ich natürlich nicht auf den Cash zugreife, dann erfahre ich eine gewisse Strafe, die Miss Penalty, und das heißt, das Memory Delay berechnet sich natürlich aus der Missrate mal dieser Miss Penalty, speicherzugriffen pro befehl die ich dann ausrechnen kann und die missrate die ich natürlich auch entsprechend ausrechnen kann das heißt wir könnten jetzt mal zwei unterschiedliche architektur miteinander vergleichen mit und ohne cache wie sieht es dann aus also wir haben eine architektur a ohne cache und eine architektur b mit cache basisdaten ist der taktzyklus ist 50 nanosekunden der zugriff auf den hauptspeicher dauert 200 also kleiner als 50 Nanosekunden. Ich habe einen CPI von 1,5, also Cycle Span Instruction von 1,5, das heißt im Mittel erledige ich 1,5 Instruktionen pro Takt

und ich habe eine Speicherzugriffsrate von 1,3, das heißt für jeden Befehl hole ich ungefähr 1,3 Worte im Mittel vom Speicher. Ich habe eine Hitrate von 95%, das heißt meine Missrate beträgt 5% auf dem Cache nicht im Cash finde, habe ich eine Miss-Penalty von 10. So, wenn wir jetzt diese ganzen Zahlen einsetzen, für eine Architektur ohne Cash, dann habe ich hier die Ausführungszeit, berechnet sich aus die Anzahl der Befehle, Entschuldigung, mein Tagzyklus, 1,5 ist mein CPI plus 1,3, das ist der Zugriff auf den Speicher 3 zusätzliche Zyklen, weil der Zugriff auf den Speicher dauert 200 Nanosekunden, meine Zykluszeit ist aber 50 Nanosekunden, habe ich also 3 zusätzliche Zyklen, die ich auf den Speicher warten muss für jeden Befehl und das geht hier multiplikativ mit ein. Das heißt, meine Ausführungszeit ist Ni mal Tc mal 5,4. So, jetzt stelle ich die gleiche Formel auf für die Architektur mit Cache das im Normalfall finde ich alles in Cache, das heißt hier multipliziere ich mit 1, allerdings habe ich eben bei 5% aller Zugriffe in Cache Miss und muss dann mit den entsprechenden Strafzyklen auf den Hauptspeicher zugreifen und selbst das führt aber immer noch dazu, dass meine Ausführungszeit weniger als halb so groß ist, das heißt ich bin mehr als doppelt so schnell. Ja, also ungefähr, wenn ich die zwei Zeiten miteinander vergleiche, kann ich den Speedup berechnen und der Speedup liegt bei 2,5. Das ist nur ein anschauliches Beispiel, aber klar ist, dass Zeiten, die ich damit verbringe, dass der Prozessor auf dem Speicher wartet, das sind Zeiten, in denen ich nur Wärme produziere und nichts Sinnvolles arbeite. minimieren möchte, die der Prozessor mit Warten verbringt. Und ich könnte ohne eine Cache-Hierarchie, könnte ich moderne Prozessoren nicht ausreichend schnell mit Daten und Befehlen versorgen, weil sie ansonsten die meiste Zeit eben in Waitstates verbringen würden. Gut, es ist auch klar, dass ich immer dran schleife, und was ich beobachten kann ist natürlich, dass der Cache erst dann wirkt, wenn ich das erste Mal auf so einen Cache-Block zugegriffen habe. Dann wird ja der Cache-Block eingelagert vom Hauptspeicher in den Cache und eine Idee könnte ja sein, dass ich versuche spekulativ herauszufinden, was könnte denn der nächste Cache-Block oder der erste Cache-Block sein. Das heißt, ich lade den Cache schon, bevor überhaupt die Daten im Cache genutzt werden. Prefetching heißt also, nutze das Prinzip der zeitlichen oder örtlichen Lokalität von Daten und lade Daten, bevor sie genutzt werden. Es ist natürlich eine Kunst, da die geeigneten Vorhersagen für das Prefetching aufzustellen. Das heißt, ich will natürlich auch nicht allzu viel unnötige Arbeit machen. Es ist jetzt nicht ganz so schlimm, wenn man mal einen Cashblock umsonst lädt, aber wenn man natürlich hauptsächlich Cashblocke umsonst lädt, die dann überhaupt nicht gebraucht werden, dann geht diese Strategie natürlich nicht auf. Prefetching Strategien realisieren. Ich kann versuchen einfach zu beobachten wie auf dem Speicher zugegriffen wird und das beobachte ich mit Hardware, mache dann Prediction Bits und versuche dann eine halbvolle vernünftige Vorhersage zu treffen um dann Blöcke vorzuladen oder ich erkenne irgendwelche Muster wenn ich Laufschleifen habe und diese Muster nutze ich dann um entsprechend sinnvoll Die andere Möglichkeit, die ich habe, ist bei unserer Strategie, ist ja immer so, dass Speicherzugriffe blockiert werden, wenn ich ein Cache miss habe, durch ein Cache, damit Konsistenz und Kohärenz erhalten bleiben. Zugriffszeit mit Berechnungen zu überdecken. Das heißt, in

der Zeit, in der ich am Speicher warte, führe ich andere, in Anführungszeichen, sinnvolle Befehle aus. Ich muss da natürlich immer auf Datenabhängigkeit achten, die dürfen nicht verletzt werden. Solange die nicht verletzt werden, kann ich natürlich Berechnungen gleichzeitig durchführen in der Zeit, in der ich auf dem Hauptspeicher warten muss. Aber auch hier muss ich natürlich eine Datenabhängigkeitsanalyse und der korrekte ablauf muss sichergestellt bleiben das heißt ich darf durch meine strategien die ich mache um die performance zu verbessern darf ich natürlich keine fehlerhaften system zustände generieren das ist also oberste priorität dass alles korrekt abläuft und die nächste priorität ist dass es dann eben möglichst performant abläuft und da kann ich mir alle möglichen auch teilweise skurrilen sachen überlegen muss dann eben durch test durch benchmarking durch feststellen, ob das gute oder schlechte Strategien sind.