Okay, das nächste Kapitel ist überschrieben mit Komponenten superskalare Prozessoren. Also es geht natürlich davon aus, dass wir das Kapitel vorher halbwegs verstanden haben. haben wir ja gerade darüber gesprochen und jetzt geht es wirklich darum, ein bisschen detaillierter zu verstehen, was diese Superskalare Architektur für eine Auswirkung hat auf die Pipeline, also auf den Datenpfad. Was muss ich alles beim Datenpfad beachten, um so einen Superskalaren Prozessor gut auszulasten und neben Datenabhängigkeiten haben wir natürlich unsere Kontrollabhängigkeiten, größte problem von solchen architekturen eigentlich branch probleme sind die ich lösen muss wenn ich so eine architektur überhaupt halbwegs performant am laufen haben will dann muss ich mir irgendwas bezüglich sprüngen einfallen lassen wir werden ganz normal uns unseren datenpfad anschauen was wir alles beachten müssen was wir verändern müssen das heißt der der größte anteil dieses kapitels besteht eigentlich darin die probleme uns anzuschauen beim Fetch, also insbesondere wie kann ich Sprungvorhersage gut machen, wie kann mir Predication helfen, um hier möglichst viel Parallelität reinzubringen und natürlich auch eine spekulative Ausführung. Die nächste Phase ist die Dekodierung, Instruction Decode Phase, die ist eher weniger kritisch, Registerumbenennung, um irgendwelche Konflikte zu lösen. die datenabhängigkeiten gelöst in dieser phase ja dann habe ich die execute phase für die befehlszuweisung die ist mehr oder weniger harmlos bis auf das wir halt viele execution units parallel am start haben also diese befehlszuweisung dafür brauche ich jetzt natürlich eine issue unit oder ein hardware scheduler und dann werden diese befehle ausgeführt und da ich auch out of order ich bis jetzt nicht brauchte nämlich diese retirement phase um erstens mal auszusortieren welche befehle habe ich spekulativ ausgeführt ich gar nicht mehr brauche dann schmeiße ich die weg und um eine richtige reihenfolge meines programs wieder herzustellen und natürlich gibt es da auf dem weg probleme also insbesondere hier am ende wenn ich dann mehrere execution units gleichzeitig nutze und da gibt es halt hardware verfahren die diese konflikte Scoreboarding dazu und auch die Ideen von Thomas Sulo und Datenflussideen. Okay, also wir sind, das ist ja auch im Bild bei der Aufzeichnung unser rechtes Standbild, wir sind in der superskalaren Architektur unterwegs, das heißt hier ist unser Datenpfad von links nach rechts und wir haben eben unsere 5 Phasen und wir wollen uns jetzt erstmal wenn ich so eine superskalare Architektur möglichst schnell mit Befehlen versorgen will. So, also was wir jetzt bekommen in der Instruction-Fetch-Phase als Ergebnis praktisch, was hinten rauskommt, das ist nicht ein Befehl, den ich geholt habe und den ich in die nächste Phase gebe, sondern das ist ein Block von Befehlen. Diesen Befehlsblock hole ich vom Instruction-Cache, das heißt wir gehen von vornherein aus, getrennten Instruktions-Cache und getrennten Daten-Cache. Warum? Weil so ein Instruktions-Cache natürlich deutlich einfacher zu handeln ist als ein Daten-Cache, weil ich im Wesentlichen lesend auf diesen Instruktions-Cache zugreife. Ich verändere ja keine Befehle, sondern ich lese diese Befehle im Instruction-Fetch und verarbeite die weiter und deshalb kriege ich dann eigenen Cache dafür. Gut, also damit ich eben auf der Ebene schon mal Konflikte vermeide, zwei Speicher dafür. Ganze wie gesagt wird als Harvard Architektur bezeichnet. Gut, also Instruction Cache ist

einfacher, weil er Read-Only ist. So, ja typische Instruction Cache Größe, wenn das Ganze On-Chip ist, ist man irgendwo im Kilobyte Bereich, jetzt vielleicht ein bisschen größer, teilweise wahrscheinlich schon im Megabyte Bereich und dieser Cache ist normalerweise entweder Direct Mapped oder zwei- bis vierfach Satzassortativ. Ich und auch auf die Übungen, wo sie dann diese Assoziativität besprochen haben. Also je höher Assoziativ, desto mehr muss ich suchen. Direct Mapped habe ich so einen direkten Verweis. Auf der anderen Seite bin ich mit Assoziativ natürlich ein bisschen flexibler. Also das ist immer so die Frage, was ist für meine Zwecke jetzt die beste Architektur? Das ist eine Aufgabe für einen Rechnerarchitekten, der muss das halt entscheiden, anhand von Untersuchungen, Simulationen und Benchmarking. so ein Cache-Block jetzt mal mir anschaue, den ich hole, der ist in der Regel 32 Byte-Breit, auch das kann sich natürlich ändern, je nach Architektur. Und wenn ich jetzt von einer 32-Bit-Architektur ausgehe, dann sind das 8 Instruktionen. Also nicht jetzt sagen, okay, wir haben doch jetzt immer bloß noch 64-Bit-Architekturen, keine 32-Bit-Architekturen, das ist eigentlich egal, das Prinzip bleibt ja das gleiche. Und wir haben eben diese Vorlesung, damit es keine Verwirrungen gibt beim Erhöhen vom Befehlsziel usw., durchgängig für 32-Bit-Architekturen die Beispiele alles gemacht. Damit sie nicht durcheinander kommen. Aber natürlich ist dieses Prinzip genauso gültig für 64-Bit-Architekturen. So, was ist jetzt das Problem? Das Problem ist, wenn ich jetzt so einen ganzen Block hole und ich habe irgendwo einen Sprung in diesem Block, also entweder einen Branch am Anfang oder in der Mitte oder irgendeinen Jump, der irgendwo hingeht in der Mitte eines Cache-Blocks, dann überspringe ich ja die anderen Instruktionen, die da drinstehen und dann kann ich die wegschmeißen. Das heißt, das Blockholen beinhaltet das Problem, dass da Befehle jetzt möglicherweise dabei sind, die ich gar nicht brauche, aber meine ganze Architektur ist natürlich darauf ausgerichtet, dass ich diese Befehle gleichzeitig verarbeite. Das heißt also, wenn das passiert, dann habe ich nicht genügend Befehle, um die Pipeline voll auszulasten, weil ich da einfach Befehle geholt habe, die ich nicht brauchen kann. die nutzbare Befehlsbandbreite deutlich kleiner als die Zuweisungsbandbreite und das führt zu Pipeline-Stalls. Pipeline-Stalls ist kein ganz neues Thema und wir wissen, dass jetzt in der nicht skalaren Architektur Pipeline-Stalls schon wehgetan haben und wir einige Verrenkungen in der Hardware gemacht haben, um Pipeline-Stalls gar nicht erst auftreten zu lassen. Wir konnten letztendlich auf Hardware-Ebene alle Stalls bis auf einen, der so zwischen Load und Add auftaucht konnten wir alle Pipeline Stalls lösen im sequentiellen Fall. So jetzt haben wir den parallelen Fall und da tut eine Pipeline Stall natürlich noch mehr weh. Das heißt wenn da mehrere Befehle in Bearbeitung sind und die Pipeline stallt dann wird es richtig teuer und dementsprechend muss ich dafür sorgen dass ich genügend Befehle in petto praktisch habe die ich meiner Pipeline Und da ist so eine Daumenregel, die da mal aufgestellt wurde, dass wenn ich so einen 8-fach superskalaren Prozessor habe, mit einer einfachen Instruction Fetch Stufe, dass ich da dann 4 Befehle pro Zackzyklus maximal zuweisen kann. Die anderen 4 Befehle gehen irgendwo verloren. schon im Instruction Cache ein bisschen was vorzubereiten. Das heißt, dass ich nicht nur einen Block lese, sondern dass ich zwei Blöcke lese und dass ich

dann versuche, genügend Befehle aus diesen zwei Blöcken rauszuholen, die ich dann irgendwo gleichzeitig ausführe. Außerdem, was man natürlich zusätzlich macht, ist ein Prefetch von Befehlen. Das heißt, dass ich einfach spekulativ Befehle hole, die ich denke, dass demnächst ausgeführt werden anstelle, den ich schneller bearbeiten kann als den Cache. So, wenn ich solche Strategien fahre, dann kann ich natürlich in das Problem laufen, dass ich die Lokalität von Befehlen ausnutze und dass das nicht unbedingt dynamisch auch der Fall ist. Das heißt, ich mache so eine statische Zuordnung, ein Befehl kommt nach dem anderen. so eine Laufschleife oder Sprünge habe. Das heißt im normalen Instruction Cache habe ich hier meine Befehle, dann werden hier Befehle ausgeführt durch einen Sprung und dann komme ich hierher wieder zurück und wenn ich das jetzt beobachten kann und feststelle, ja was hindert mich daran eigentlich dieses Stück hier virtuell lokal einzubauen, dass ich einen zusammenhängenden Block habe, den ich dann hole. Dafür muss ich das natürlich wissen, deshalb heißt das ganze Dynamics Traces an und wenn ich diese Situation feststelle, ja dann baue ich das einfach so zusammen, damit es in Zukunft quasi ein Block ist. Er wird einfach, so ein Trace Cache wird parallel zum Instruction Cache betrieben, das heißt er beobachtet erstmal was da passiert und er stört auch weiter nicht. Er kann nur gut tun, wenn er das Ganze feststellt. Also während der Befehlsausführung läuft der Trace Cache mit und dann, wenn er irgendwelche solche Muster feststellt, dann baut er das halt richtig zusammen ganz wichtig haben wir gesagt sind die Sprünge, es sind ein Problem wir müssen also unsere Sprungvorhersage so genau wie möglich machen und wenn wir das nicht schaffen dann wird sich das in der Performance deutlich auswirken wenn ich natürlich also Sprünge nicht vorhersagen ist das eine Thema Wenn ich aber spekulativ Sprünge ausführe und mache das falsch, dann ist das natürlich noch ein größeres Problem. Ich muss nämlich dann dafür sorgen, dass dieser falsch ausgeführte Sprung keinen Nebeneffekt hat. Das heißt, ich muss einen Rollback durchführen und so ein Rollback kann sehr teuer sein. Also Beispiel Pentium oder Alpha, ältere Prozessoren klar, aber bei neuen Prozessoren wird es nicht unbedingt besser, weil die Pipelines noch tiefer sind. gebraucht um so eine falsche sprungvorhersage zu korrigieren und das ist natürlich richtig teuer also keine sprungvorhersage schlecht falsche sprungvorhersage ganz schlecht muss man aufpassen so und das nächste problem ist dass sprünge gar nicht so selten sind das heißt einer aus fünf bis sieben je nach architektur von befehlen sind sprungbefehle also wir haben hier eine prozentzahl zwischen 14 und mehr als 20 gut, das heißt aber auch und das macht die Sache noch schlimmer dass wenn ich eine spekulative Befehlsausführung habe dann kann es sein, dass ich in der Pipeline mehrere Sprungbefehle behandeln muss nicht nur einen Sprungbefehl behandeln muss und ja, dementsprechend aufwendig wird die ganze Geschichte ok, schauen wir uns mal an wie oft sind wir denn in der Situation wie oft passiert das Ganze Wir machen wieder Benchmarking, wir haben die Spec, wo ich sehr viele Benchmarks finde, unterschiedlicher Natur, also von Integer über Floating Point Benchmarks über String Processing Benchmarks, finde ich alles mögliche. bei dieser Anwendung hier, bei diesem Benchmark hier, LibQuantum Benchmark, dass ich hier sogar 27 Prozent aller Programmbefehle sind, sprünge.

Das heißt, ich habe wirklich ein großes Problem zu lösen. So, dementsprechend muss ich das Ganze in meinem Datenpfad, in meiner Pipeline sehr frühzeitig lösen und wir haben jetzt in der E-Fetch Phase, Instruction Fetch Phase, da und PowerPC gesehen, wo da Branch Unit gestanden ist. Da haben wir also mehrere neue Einheiten. Wir haben einmal unseren Branch Predictor. Das ist also eine Logik für die spekulative Ausführung von Branches. Und die ist natürlich verbunden mit dem Instruction Cache, der eben mir ausspuckt, die Befehle oder den Block von Befehlen, der in der nächsten Phase Instruction Decode weiterzuverarbeiten ist. wird das ganze ein bisschen renamed, register renaming, das heißt einfach wenn ich feststelle dass mehrere befehle mit den gleichen register namen arbeiten dass ich die dann umbename wenn ich es kann wenn da keine datenabhängigkeiten bestehen um eben diese register ungestört in der execute phase nutzen zu können dann packe ich das ganze in eine issue unit das heißt okay jetzt ist mein befehl zur ausführung bereit auf diese die habe ich einmal für integers und einmal und auf diese issue unit, da greift dann der Hardware Scheduler zu und der weist dann diese Befehle eben den Execution Units zu. Er muss natürlich erstmal die entsprechenden Register, die vorgeschaltet sind in den Execution Units, mit den entsprechenden Daten füllen. Dann findet hier die Execute Phase statt und dann geht es entweder wieder zurück ins Register des Ergebnisses oder wenn ich Speicherzugriffsbefehle habe, dann geht es halt in den Daten Cache, also wenn ich Load Store Befehle habe. Befehle, die ausgeführt werden. Gut, also es geht jetzt ein bisschen in den nächsten Folien darum, diesen Branch Predictor erstmal genauer zu verstehen. So, wenn ich keine Vorhersagen habe oder Fehlprognosen habe, dann ist das sehr teuer. Haben wir jetzt glaube ich genügend begründet. Hier ist einfach nochmal das Beispiel. Ich habe den Befehl A, so nach dem Befehl A habe ich jetzt einen Sprung und ein Sprung entweder die Möglichkeit B1 oder die Möglichkeit B2 und möglicherweise, wenn das Ganze auf dynamischen Daten basiert, weiß ich einfach nicht, welcher Befehl kommt als nächstes. Weil standardmäßig führe ich aus dem Fetch PC plus 4, also der wird immer ausgeführt, dass ich den nächsten Befehlsblock hole. eine Fehlprognose gemacht habe, dann muss ich natürlich diese ganze Pipeline wieder leeren und die Strafe, die ich erleide, das ist praktisch die Tiefe der Pipeline. Also ich muss das Ganze hier zurück abwickeln, so ein Rollback, um diesen Fehler, den ich gemacht habe, zu korrigieren. Also deshalb ist es eben sehr wichtig, diesen Fehler nicht zu machen. Das heißt, ich muss versuchen natürlich, den Befehl von der korrekten Zieladresse zu holen, ist klar. Okay, was kostet mich denn tatsächlich? Machen wir einfach mal ein Zahlenbeispiel, die Performance-Einbuße. Wir gehen von ein paar Größen aus in unserer Formel. B soll es sein, die Sprunghäufigkeit in Prozent, also wie oft springe ich. M, die Fehlprognoserate, wie oft habe ich falsch spekuliert. die ich erfahre, also die Branch-Panel, die intakten, wie viele Takte kostet mich eine falsche Spekulation und die Ausführungszeit T eines Programmes mit N Befehlen, wenn ich jetzt von einer einfach skalaren Architektur ausgehe, also von einer sequentiellen Architektur ausgehe, dann habe ich, sagen wir mal, pro Takt wird ein Befehl erledigt, dann ist meine Zeit, ich da brauche N für die N Befehle plus die Strafe, die ich erleide und die Strafe, die ich erleide, ist N mal B mal

M mal P. Anzahl der Befehle mal Sprunghäufigkeit mal Fehlerprognoserate mal die Penalty, die es mich kostet in Takten. Gut, gehen wir jetzt mal von einer 5-Skalaren-Pipeline aus, also nicht 1-Skalaren, sondern einer 5-Skalaren-Pipeline. Das heißt, ich will 5 Befehle pro Takt gleichzeitig bearbeiten. Wenn ich eine Fehlprognose mache, dann habe ich eine Strafe von 20 Takten. Also das Rollback kostet mich 20 Takte. Und die Frage ist jetzt, wie lange dauert es 500 Befehle zu holen, wenn ich 20% Branches habe. Also ein Fünftel sollen Sprungbefehle sein und wie lange dauert mich das. Wenn ich von einer Vorhersagekorrektheit von 100% ausgehe, also ich mache alles, alle Vorhersagen, die ich treffe sind richtig, was in der Realität nicht allzu oft auftreten wird, dann ist es natürlich einfach, ich hole 5 Befehle pro Takt. also 500 durch 5 ist 100, das heißt das Ganze dauert 100 Takte und ich habe keine Strafe erlitten, also keine zusätzliche Arbeit. Ich schaffe das also in 100 Takten. So, jetzt gehen wir mal davon aus, dass ab und zu so ein kleiner Fehler passiert, das heißt 99% Vorhersagekorrektheit, das ist ja schon ein ganz guter Wert, das heißt meine Fehlerprognoserate liegt bei 1% und jetzt können wir dann im Wesentlichen Seite vorher anwenden, also wenn wir alles richtig machen, dann habe ich 100 Takte und dann kommt noch die Strafe drauf für den falschen Pfad, wenn ich den bestreite. Und die Strafe für den falschen Pfad ist N mal B mal M mal P und dann setze ich das Ganze ein, dann habe ich 500 mal 20% mal 1% mal 20 Straftakte und dann komme ich also auf den Wert von 20, das heißt ich habe insgesamt 120 Takte. verschlechtert sich schon mal um 20%. Bei 1% Fehlerrate habe ich 20% schlechtere Performance und ich muss natürlich zusätzlich 20% Befehle ausführen. Ich habe 20 zusätzlich Takte, pro Takt hole ich 5 Befehle. Also habe ich 100 Befehle, die ich zusätzlich ausführe. Das sind 20% aller Befehle, die zusätzlich ausgeführt werden. Jetzt haben wir hier die 5% Fehlprognose und wir haben wieder am Anfang 100 richtige. und 5% sind falsch, das heißt die Strafe ist N mal W mal M mal P wie vorher, aber jetzt ist das natürlich ein bisschen mehr, weil wir hier 5 jetzt stehen haben, das heißt wir brauchen 200 Takte, also wir sind halb so schnell, bei einer Vorhersagekorrektheit von 95%, das heißt nur 5% Fehlerquote kostet mich die Hälfte der Performance. eine gute Vorhersage ist, welche Befehle bearbeitet werden müssen. Also eine gute Sprungvorhersage ist, wie wichtig das ist. Das heißt, ich mache natürlich doppelt so viele Befehle. Also 100% aller Befehle sind die zusätzlichen Befehle und dementsprechend bin ich natürlich nur halb so schnell. Gut, also ich glaube, das zeigt uns ganz eindrucksvoll, dass wir da einen guten Job machen müssen. Und wie schaffen wir das? Das schaffen wir nur dadurch, dass wir, auf den nächsten 20 Folien ist das Ergebnis von, keine Ahnung, 20 Jahren Forschungsarbeit von vielen Forschungsgruppen in vielen Labors. Das Ganze wird jetzt kondensiert und das ist jetzt nicht so, dass sich da jemand hingesetzt hat mit Papier und Bleistift und nach 10 Minuten eine gute Lösung erreicht hat, sondern dass man da wirklich jahrzehntelang geforscht hat und das sind die Ergebnisse, die wir uns jetzt anschauen. Also was müssen wir machen, um erstmal muss ich schauen, ist jetzt der geholte Befehl ein bedingter Sprung oder ist es einfach ein Jump dann wenn ich weiß, es handelt sich um einen Sprung, dann muss ich natürlich so früh wie möglich wissen springe ich nach vorne oder springe ich nicht, äh nicht nach vorne, Entschuldigung dann muss ich

wissen, ob ich springe oder nicht, wenn es ein bedingter Sprung ist das heißt, nehme ich den Sprung oder nehme ich ihn nicht für nicht taken. Und wenn ich jetzt den Sprung nehme, wenn ich weiß, der Sprung wird genommen, dann ist es natürlich wichtig zu wissen, wohin springe ich denn? Also die Sprungzieladresse, Branch Target Address, BTA abgekürzt, die muss ich natürlich auch möglichst frühzeitig herausfinden. So, welche Sprünge gibt es jetzt alles in Programmen? Normalerweise erstmal oder Branch equals Zero, irgend sowas. Ich habe also eine Bedingung, die wird ausgewertet, und dann springe ich oder nicht. Und wohin ich springe, das steht in einem Register drin. Wir gehen ja von der Risk-Architektur aus, und eine Register-Register-Maschine, und die Sprungadresse steht immer im Register drin. Wohin ich springe, das weiß ich nicht. Ob ich jetzt nach vorne oder nach hinten springe, das weiß ich nicht. Ob ich springe oder nicht, das weiß ich auch nicht, das hängt von der Bedingung ab. Ich habe zwei Möglichkeiten, entweder ich springe, dann ist es taken, oder ich springe nicht, dann ist es nicht taken. Und dementsprechend habe ich zwei unterschiedliche Adressen für die nächsten Befehle, die auszuführen sind. So, und wann weiß ich praktisch, wohin ich springen muss, wenn ich springe? Da muss ich ins Register schauen, also das steht eben im Register drin. es unbedingt die Sprünge, also Jumps, die werden natürlich immer genommen, das heißt ich weiß, dass die genommen werden. Ich weiß, ich habe nur eine Folgeadresse und diese Folgeadresse, die kann ich einfach ausrechnen, das habe ich schon zu dem Zeitpunkt, als ich mein Instruction Fetch mache, das steht ja im Maschinenbefehl drin, das heißt ich kann an der Stelle schon die Sprungadresse ausrechnen oder darauf zugreifen zumindest, weil es im Wesentlichen im Program Counter drinsteht, da kommt noch ein Offset drauf, so eine einfache Addition, dann weiß Beim Unterprogrammaufruf ist es genauso, das ist ein Sprung der immer genommen wird und auch da beim Unterprogrammaufruf steht im Maschinebefehl drin, wo ich springe, kann ich darauf zugreifen, ich brauche nicht auf Register zugreifen. ich weiß nicht wohin ich zurückspringe, ich muss da auch wieder ins Register schauen. Also da gibt es natürlich sehr viele Möglichkeiten, wo ich vom Unterprogramm aus zurückspringe und da muss ich wieder ins Register schauen, in dieses spezielle Stackregister, wo drin steht, wohin, was der nächste Befehl ist, der auszuführen ist. Also das sind so die vier grundsätzlich unterschiedlichen Möglichkeiten, die ich bei Sprüngen vorfinde. Und jetzt kann ich im einfachsten Fall mal versuchen, eine statische Sprungvorhersage zu machen. Das heißt schon zur Compile-Zeit. Und eine Möglichkeit wäre, okay, ich sage vorher, Sprünge werden nie genommen. Ich habe Branches und weiß, okay, die meisten Branches ist so eine If-Abfrage, ob irgendwas vorhanden ist. Und die allermeisten Bedingungen sind nicht erfüllt. Das heißt, ich springe in der Regel nicht. Und deshalb sage ich einfach vorher, okay, Sprünge werden nie genommen. einfach zu realisieren, allerdings wenn ich dann eine Laufschleife habe, dann habe ich ein Problem, weil hier symbolisiert, am Ende ihrer Laufschleife springe ich natürlich. Und wenn so eine Laufschleife bis 10.000 geht, bis 100.000 geht, springe ich natürlich sehr, sehr häufig und das würde dann natürlich bedeuten, dass meine Vorhersage nie genommen ganz schön auf die Nase fällt. dann doch sehr teuer, weil es einfach Performance kostet. Die andere Vorhersage, die dem entgegenwirkt, ist natürlich

die gegenteilige, indem ich sage, okay, Sprünge werden immer genommen. Und das hat den Vorteil, wenn ich jetzt in so einer Laufschleife mich befinde, dann liege ich immer richtig. Allerdings, wenn ich dann mein Programm untersuche, dann ist es natürlich klar, dass für normale Infabfragen ich da oft auch falsch liegen kann. Um das Ganze, wenn ich solche Laufschleifen habe, wirklich zu beschleunigen, ist es am einfachsten, die Sprungzieladresse ist ja immer wieder die gleiche, das heißt ich baue mir da so einen kleinen Puffer, den sogenannten Sprungziel-Adress-Cache, Branch-Target-Adress-Cache mit B-Tag abgezeichnet, abgekürzt und da drin speichere ich eben die nachfolgenden Befehle und dann kann ich auf diesen B-Tag sehr schnell zugreifen und kann also mir den nächstfolgenden Befehl sehr einfach vom B-Tag holen. So, es haben eben Untersuchungen gezeigt, dass Rückwärtssprünge öfter genommen werden und Vorwärtssprünge öfter nicht genommen werden. Daraus baue ich mir auch wieder eine statische Strategie. Die statische Strategie heißt dann eben Rückwärtssprünge genommen, Vorwärtssprünge nicht genommen. Also Backward Taken, Forward Not Taken, BTFN. Und das ist einfach so eine Untersuchung, die man über Programmbeobachtung, die das gezeigt hat. da sehen wir also der Rückwärtssprung wird immer genommen und so ein If-Else, der wird eben der Vorwärtssprung oft nicht genommen. Ja, wie komme ich zu diesen Ergebnissen? Ganz klar, ich mache eine Programmanalyse und baue mir Profile und schaue mir diese Profile genau an und treffe dann diese Strategie. gerade in dem Beispiel schon gesehen, dass ich da natürlich auch sehr oft sehr leicht falsch liegen kann und deshalb hat sich gezeigt, dass man dynamische Methoden braucht. Also statisch, wir haben ja, denken Sie vorhin an die 5% Fehlerquote, dass die schon ziemlich weh getan hat und ich muss also besser werden und das schaffe ich mit statisch nicht, ich muss also das Ganze dynamisch machen. Das heißt, ich muss ein bisschen während der Laufzeit den Sprungverlauf beobachten daraus eine sprung richtung vorher sagen also ich beobachte einfach mein programm in hardware und macht dann eine geeignete logik die mir dann hilft die richtige entscheidung zu treffen ich muss natürlich immer dann wenn ich weiß welche befehl geholt wird den irgendwo schnell verfügbar haben das heißt meine sprung ziel adresse die muss im beta sein also im branch address cash Sprung wird genommen, dass ich dann schnell auf diese Adresse zugreifen kann. Das heißt, er ist dann delay-free dieser Branch, weil eben der Befehl da schon abgespeichert ist in dieser Tabelle. Dann muss natürlich die Hardware in der Lage sein, bei einer falschen Vorhersage den Rollback so schnell wie möglich zu machen. Das heißt, dass das Penalty für eine falsche Spekulation so gering wie möglich ausfällt. mehrere Branches in einem Block drin habe und die spekulativ ausführe, dann arbeite ich gern mit Predication. Jetzt müssen wir an der Stelle ein bisschen aufpassen, dass Sie das nicht verwechseln. Also es gibt Prediction und es gibt Predication. Prediction heißt Vorhersage. Das ist also Sprungvorhersage. Predication, das heißt Prädikation, also es gibt auch im Deutschen das Wort mit E das ist praktisch, dass ich Befehle mit Prädikaten versehe. Also dass ich da zum Beispiel reinschreibe, das letzte Mal wurde dieser Sprung genommen, dass ich da so einen Flagbit setze oder irgendwas anderes. Also ich versehe meine Befehle mit irgendwelchen Prädikaten, die mir Rückschluss erlauben auf das Sprungverhalten. Während

Prediction ist einfach Vorhersage eines Ereignisses. Also nicht verwechseln, wenn wir von Predication sprechen, dann sprechen wir von Prädikaten, wenn wir von Vorhersage. So wie sieht jetzt dieser magische Branch Target Address Cache aus? Es ist ein voll assoziativer Cache, das heißt ich kann an beliebiger Stelle meine Befehle einfügen und die komplette Sprungbefehlsadresse die ist praktisch das Tag. Sie erinnern sich an Caches, an Assoziativität, an die Tags, dass also Hier ist praktisch die komplette Sprungbefehlsadresse ist das Tag. Also wir haben hier die Sprungbefehlsadresse und in dieser Sprungbefehlsadresse ist hinterlegt die Sprungzieladresse. Gut. Das Problem ist, dass ich diesen Branch Target Address Cache praktisch schon in der Instruction Fetch Stufe auswerten muss dekodierte Befehle, weil wir sind ja beim Instruction Fetch, ob das ein Sprung oder ein bedingter Sprung ist. Und dann kann ich dann, wenn ich weiß, okay, das ist ein Sprung oder ein bedingter Sprung, dass ich dann gleich auf diesen Betag auch zugreifen kann. Das heißt, das passiert alles in der Instruction Fetch Phase. Und wir sind ja momentan auch bei der Instruction Fetch Phase. So, und jetzt kommen die zusätzlichen Speicher und die zusätzliche Kombinatorik, die ich brauche, um eine halbwegs einmal eine Branch Direction History, das heißt wenn Sprünge ausgeführt werden, speichere ich einfach, ok Sprung wird genommen oder nicht genommen und eine Branch, also Direction History, das heißt Vorwärtssprung, Rückwärtssprung und Sprung Verlaufstabelle, Sprung genommen oder nicht genommen in der Branch History Table. Und das trenne ich voneinander, also das Branch History Table ist ein Extraspeicher, der getrennt ist So, die Branch History Table, damit ich weiß, okay, ist das jetzt für meinen Sprung zuständig oder nicht, die wird indiziert mit den niedrigen Bits der Sprungadresse. Das heißt, ich nehme nicht die ganze Sprungadresse, sondern nur die untersten Bits und hoffe, dass die Sprünge sich in den untersten Bits unterscheiden. Im schlimmsten Fall unterscheiden die Sprünge sich nicht im untersten Bit, dann habe ich halt mal so eine Korrelation, die zu einem Problem führen kann. History Table, da stehen wesentlich die Vorhersage-Bits drin, ob ich jetzt taken oder non-taken fallen habe, also genommen oder nicht genommen. Also die Idee, die man dabei hat, ist einfach, okay, wenn so ein Sprung nochmal aufgerufen wird, dann verhält er sich wie das letzte Mal. Also der nächste Branch verhält sich wie der vorherige Branch. Das ist das Prinzip. Das stimmt ja zum Beispiel bei Laufschleifen. Jetzt kommt noch eine Frage. Ja, gerne. Ja, und zwar wurde gefragt, ob der Compiler das BTFN zum Beispiel jetzt F-Anweisungen auch optimieren würde für die Hardware. Okay, BTFN ist die Spekulation hier, diese da. Vorhersage rückwärts genommen, vorwärts nicht genommen. Ob der das für die Hardware optimiert? Also sagen wir mal so, diese statische Sprungvorhersage, das ist eine Idee, die aber in der Regel nicht so oft zum Tragen kommt. Compiler natürlich wahnsinnig schwierig so eine statische Sprungvorhersage zu treffen, weil die ja ja in der Regel ist es eine reine Spekulation die der Compiler da trifft und der der wird es wahrscheinlich nicht nochmal zusätzlich für die Hardware optimieren sondern wenn der Fall nicht eindeutig ist dann, also ein optimierender Compiler wenn ich so eine Abfrage habe, if a gleich 0 und ich habe vorher a auf 0 gesetzt aufzulösen und das ist ja dann schon eine statische Sprungvorhersage, die richtig ist. Aber wenn der Com-

piler die Bedingungen nicht auflösen kann, weil die Bedingung eher dynamisch ist, dann würde ich sagen, er gibt das Problem an die Hardware weiter und versucht da jetzt nicht für die Hardware zu optimieren, weil die Hardware dann da eh nochmal drauf schaut. Also wird da jetzt nicht speziell für die Hardware da irgendwas reingeschrieben, sondern der Befehl wird praktisch dann kodiert und das wird der Hardware überlassen, was er mit diesem Befehl hinsichtlich Sprungvorhersage zu tun hat. Okay, gut, also wir sind bei der dynamischen Sprungvorhersage beim Brainstarget Address Cache und hier ist die Erklärung und hier ist nochmal ein Bild dazu. der befehls zähler besteht aus tag und index und wir haben hier eine sprung verlaufs tabelle für die sprung erlaubt verlaufs tabelle da werden praktisch die unteren bits genommen um nachzuschauen okay wie hat sich der sprung vorher verhalten mit der gleichen adresse ja wobei hier eben über lagerungen stattfinden können weil ich nur die unteren bits nehmen also je nachdem 2 und dann schaue ich nach, ok, finde ich unter diesen Bits was und wenn ich was finde, dann finde ich hier die Information, wurde das letzte Mal der Sprung genommen oder nicht genommen. Und darüber hinaus haben wir den Sprung Ziel Address Cache, also hier steht jetzt tatsächlich drin unter dem Tag, wobei normalerweise für das Tag wirklich der komplette Program Counter Adresse genommen wird, da steht jetzt unter dieser Adresse in der Zeile, da steht jetzt drin, was mein Sprungziel ist. Sprungziel, also den nächsten Befehl, den ich brauche und hier bekomme ich Informationen, wurde der Sprung genommen oder nicht genommen und dementsprechend habe ich zwei Möglichkeiten, wenn ich jetzt spekuliere, dass der Sprung nicht genommen wird, dann erhöhe ich meinen Befehlsteller ganz normal um 4, also bei einer 32-Bit-Architektur und ansonsten, wenn ich der Meinung bin, es wird gesprungen, dann habe ich hier die Zieladresse und gebe praktisch wenn ich mit einem Branch Target Address Cache arbeite und mit einer einfachen Sprungverlaufstabelle, Branch History Table. So, gerade am Anfang, ich bin dynamisch und ich muss ja erstmal mein Programm laufen lassen und ich kann erst dann beobachten, wenn schon ein paar Befehle ausgeführt wurden. gefüllt während der Befehlsausführung und der Branch Target Address wird auch dynamisch gefüllt während der Befehlsausführung. Das heißt also, ein Warm-up am Anfang, da liege ich natürlich, da habe ich natürlich nichts in diesen beiden Speichern und ich beobachte erstmal ein bisschen und dann versuche ich Vorhersagen zu treffen. So, was ich natürlich nicht kann, wenn ich jetzt zum Beispiel immer taken habe bei einer Laufschleife und die Laufschleife endet ja irgendwann. dann werde ich natürlich zwangsläufig diesen letzten, den Ausgang der Laufschleife, den werde ich natürlich zwangsläufig falsch vorhersagen. Ja, ist klar. Das springt jetzt 10 Mal und dann auf einmal nicht mehr. Also wird der letzte Vorhersage immer falsch sein, wenn ich eine Laufschleife habe. Oder da ich ja mein Branch History Table, diese Tabelle da oben, die will ich ja möglichst klein haben, dass ich die auch schnell durchsuchen kann, kann es natürlich sein, dass sich unterschiedliche Sprungbefehle nicht unterscheiden in den unteren Bits. bei der Befehlsadresse und dadurch kriege ich eine Interferenz. Und das führt natürlich auch zu falschen Vorhersagen, weil die Sprünge, auf die ich schaue, eben nicht gleiche Sprünge sind, aber ich gleiches Verhalten vorhersage. Das kann ich natürlich einfach lösen, indem ich die Registry Table größer mache

und mehr Adressbits für die Indizierung verwende. Und eine Faustregel ist klar, je mehr Adressbits ich verwende und je größer ich meine Registry Table mache, desto kleiner ist diese Interferenz hier. So, jetzt kommen wir zu den Prädiktoren, also zu der Anzeige der Vorhersage und eine einfache Strategie ist, dass ich ihm sage, okay die letzte Ausführung des Branch Befehlssatzes bestimmt das Vorhersagebild. Also ich habe eine ganz einfache Möglichkeit in meiner Branch History Table, ich sage vorher Sprung wird genommen oder nicht genommen, taken oder non taken. Wir haben einmal den Zustand, der Sprung wurde genommen und einmal den Zustand, der Sprung wurde nicht genommen beim letzten Mal. Was passiert, wenn das nächste Mal hier Taken auftaucht und ich habe vorher schon den Sprung genommen, dann bleibe ich natürlich hier. Dann bleibe ich in Taken drin und erst wenn der Sprung nicht genommen wurde, mache ich die Vorhersage, kippe ich die Vorhersage und sage, okay, ich sage jetzt nicht Taken voraus und solange eben der Sprung nicht genommen wird, bleibe ich hier. Erst wenn der Sprung genommen wird, gehe ich wieder in diesen Zustand hier. ich kann das Ganze natürlich platzsparend machen, indem ich einfach die non-taken Teil weglasse, das heißt nur taken Branches, weil dann, ich brauche ja nur bei den taken Branches praktisch die Folgeadresse und deshalb schreibe ich auch nur taken Branches in die Branch History Table. So, das Resultat von so einem Einbitt-Prediktor ist gar nicht so schlecht, Laufschleifen werden im Wesentlichen korrekt vorhergesagt, bis eben auf den letzten Durchlauf der Laufschleife, ich falsch je nachdem wie oft diese laufschleifen durchlaufen werden bin ich sehr gut oder auch nicht so ganz gut es gab noch eine frage zur jahre ist wie table ja es wurde gefragt wann diese halt so initialisiert wird oder halt rückgesetzt wird das jetzt bei system start oder bei verschiedenen programmen bei verschiedenen programmen also das programm oder der prozess startet ja dann habe ich ja meinen adressraum in dem der prozess läuft also jeweils immer wenn dynamisch ständig praktisch gefüllt an. Solange der Prozess läuft, sagen wir mal so, solange der Prozess eine Zeitscheibe hat, in der Zeit wird das Ganze natürlich dynamisch gefüllt, weil wenn der nächste Prozess läuft, also diese Hardware habe ich ja bloß einmal pro Prozess, wenn der nächste Prozess läuft, dann muss das natürlich alles wieder geräumt werden und der nächste Prozess arbeitet dann mit dieser Hardware. Also das ist prozessspezifisch, immer solange ein Prozess läuft. praktisch dann wieder neu gefüllt. Die große Frage ist natürlich, die kann ich jetzt ehrlich gesagt nicht genau beantworten, was passiert beim Prozesswechsel? Ich habe ja zwei Möglichkeiten. Entweder speichere ich beim Prozesswechsel als Umgebung mir diese Inhalte der Branch History Table und des Branch Targeted Interest Caches mit und wenn das nächste Mal der Prozess eingelagert wird, lage ich das wieder mit ein. Das glaube ich aber eher nicht, weil das Ganze wahrscheinlich zu aufwendig ist. Das heißt, es bleibt da wahrscheinlich alles drin stehen und wenn das nächste Mal der Prozess startet und ich Glück habe, stehen vielleicht meine Daten da sogar noch drin, wenn sie nicht von anderen Prozessen verdrängt wurden. Das ist für mich jetzt die einfachste Möglichkeit. Das heißt, das Ganze wird bei Systemstart initialisiert und dann wird es dynamisch ständig von den Prozessen genutzt, die Hardware. dann wird er halt neu eingelagert und irgendeiner wird verdrängt dabei. Okay, immer

falls so eine Frage nicht ganz beantwortet ist, bitte nochmal nachfragen, kein Problem. Gut, der Ein-Bit-Prediktor, der hat natürlich ein großes Problem bei geschachtelten Schleifen. Wir haben jetzt zwei Schleifen ineinander geschachtelt und dann habe ich natürlich immer gleich zwei Fehlprognosen ineinander. Wenn eben so eine innere Schleife schleife dann wieder zum tragen kommt ja dann durch zweimal falsch vorhersagen not taken und der äußere ist dann taken und das also die äußere schleife ist auf jeden fall immer falsch die ich da vorhersage die innere bloß beim letzten durchlauf und die äußere ist dann immer falsch ich habe also hier zwei probleme beim letzten durchlauf da wird taken vorhergesagt und es ist Durchlaufe dann not taken vorausgesetzt von der äußeren Schleife, das ist aber taken. Also zwei falsche Vorhersagen und sowas kann man zum Beispiel lösen mit einem 2-Bit-Prediktor. Also wir spendieren nicht nur ein Bit für die Vorhersage, sondern wir spendieren 2 Bits für die Vorhersage, aber also vier Zustände und diese vier Zustände, die befinden sich jetzt in diesem Diagramm hier, das heißt ich habe einmal ein Predict strongly taken, Strongly Not Taken, das sind die vier Zustände, eben entsprechend abgekürzt mit ST für Strongly Taken, WT für Weekly Taken, WNT für Weekly Not Taken und SNT für Strong Not Taken. So, und wenn ich das jetzt durchlaufe, dann sehe ich, dass ich am Ende der geschachtelten Schleife nur noch eine Vorhersage habe. Und in Real habe ich hier Not Taken. Und das Not Taken führt dann dazu, dass ich den Übergang mache. Also von Strongly Taken mit Not Taken komme ich nach Weekly Taken. Und das Weekly Taken führt eben dazu, dass dieses Taken tatsächlich korrekt vorhergesagt wird. Und dann, wenn da ein Taken kommt, dann wechsle ich wieder nach Strong Taken, also nach ST. Das heißt, ich habe hier nur noch einen Fehler statt vorher zwei Fehler. Aber das hier ist ja richtig. Weekly taken führt ja zu einem taken und das habe ich hier. Das heißt, mit so einem einfachen Bit spendieren, spare ich mir eine Menge falsche Vorhersagen. Gut, also haben wir das Laufschleifenproblem schon ein bisschen besser gemacht. Man kann das Ganze jetzt auch noch anders lösen mit einem 2-Bit-Sättigungszähler. und verringere um 1, wenn er nicht genommen würde. Ich mache einfach so einen 2-Bit-Sättigungszähler, das heißt der kann maximal von 0,0 bis 1,1,1 laufen. Das sind auch wieder vier Zustände. Diese vier Zustände weise ich einfach zu. 1,1 soll strongly taken sein, 1,0 weakly taken, 0,1 weakly non taken und 0,0 entspricht dann strongly non taken. Also so ein 2-Bit-Sättigungszähler Zustandsdiagramm hier, was wir gerade für den 2-Bit-Prediktor eingeführt haben. Ja, jetzt kann ich natürlich mal untersuchen wieder das Ganze, wie gut bin ich denn eigentlich mit meiner Lösung und wenn man jetzt so eine Branch History Table mit 4096 Einträgen hat, da ist in Hennessy Patterson in der Literatur, gibt es da eben so einen Versuch dazu und da hat man festgestellt, dass die falschen Spekulationsraten zwischen 1 und 18% sind. Und zwischen 1 und 18 Prozent, das heißt, es tut immer noch richtig weh, wir müssen uns einfach noch weiter verbessern. Und dann könnte man natürlich auf die Idee kommen, naja, wenn ein 2-Bit-Prädiktor so viele Vorteile im Vergleich zum 1-Bit-Prädiktor gibt, warum spendieren wir nicht mehr als 2 Bits für die Prädiktion und für die Vorhersage? Und dann hat man wieder ewig viele Untersuchungen gemacht und festgestellt, naja, wenn man ehrlich ist, bringt nichts, also bringt keine wesentliche Verbesserung. Das

heißt, ich kann mir den Aufwand sparen. Also 2-Bit-Prediktor ist eigentlich gut genug. Ich muss andere Möglichkeiten suchen, um meine Sprungvorhersage zu verbessern. Einfach noch an der Stelle ein Bit zu spendieren, bringt mir keinen Vorteil. Was ist das nächste Problem von 2-Bit-Vorhersagen? Ja gut, wenn ich jetzt sehr viele If-then-Elses habe, also weniger Schleifen habe, ist aber nicht unbedingt gut für If-then-else-Konstrukte. Ganz im Gegenteil, wenn ich jetzt richtig unfair bin, und das machen wir jetzt mal, unfair zu sein, dann kann ich da Beispiele konstruieren, wo der 2-Bit-Prediktor richtig auf die Nase fällt. Und richtig auf die Nase fallen heißt in dem Zusammenhang, jede Vorhersage ist falsch. Ich baue mir also, ich konstruiere zugegebenermaßen jetzt ein Beispiel, ein einfaches Beispiel, ich habe If D gleich 0, dann setze D auf 1, also das ist mein Branch S1, f2, der arbeitet mit d gleich 1. So. Ich baue das Beispiel jetzt in Maschinensprache, dann habe ich also ein Branch, Entschuldigung, ein Branch not equals 0, R1 L1 ist meine Sprungmarke, dann habe ich eine Addition, ja, ich addiere hier nicht, sondern ich setze R1 zu 1, also d gleich 1, R1 beinhaltet ja d und R0 steht immer die 0 drin und wenn ich auf die 0,1 drauf addiere, dann habe ich da halt 1 drin, und hier habe ich eine Subtraktion, ich subtrahiere von D1 und dann springe ich nach L2. Wenn dieses Stück Code jetzt öfter ausgeführt wird und wenn ich mit D gleich 0 und D gleich 2 starte, dann habe ich also alternierende Sequenzen von Taken und Non-Taken. 0, das heißt Abfrage D ungleich 0 ist False, also ist die Sprungrichtung non-taken, das D vor S2 ist 0, damit ist D ungleich 1, also Abfrage ist False und die Sprungrichtung S2 ist auch non-taken. Dann hat jetzt D den Wert 2, das heißt diese Abfrage hier ist True, dann springe ich also und wenn ich springe, dann hat D den Wert vor S2 auch 1, das heißt ist auch true, dann wird beim nächsten Mal auch wieder gesprungen. Ich habe also alternierende Sequenzen von Springen und Nichtspringen. Und wenn wir das jetzt einfach mal wiederholt ausführen und in so eine Tabelle schreiben, das sind also die Code-Stücke, die immer wieder ausgeführt werden. Und eben beim ersten Mal wird nicht gesprungen, beim zweiten Mal wird gesprungen, beim nächsten Mal wird nicht gesprungen, sowohl für das R1 und R3, da ist die Abfrage drauf, das haben wir hier nochmal farblich markiert. taken wird aber ausgeführt. Not taken vorhergesagt, taken wird ausgeführt. Und das praktisch in jedem einzelnen Sprung. Und wie kann ich sowas auffangen? Also wenn ich den einen Predictor mir anschaue, da kann ich es überhaupt nicht auffangen, der liegt immer falsch. Da habe ich ja noch einen Bit und ich nehme immer das letzte Sprungverhalten. Und da ist alternieren einfach tödlich, weil da immer genau das falsche vorhergesagt wird. Wenn ich jetzt aber so einen 2 Bit Predictor nehme, mache, ich springe immer genau zwischen diesen 2 Zuständen, zwischen weekly taken und weekly not taken hin und her und zwar springe ich immer, also mache ich Zustandswechsel immer zum falschen Zeitpunkt. Das heißt also, in der Situation nützt mir weder ein 1-Bit-Prediktor noch ein 2-Bit-Prediktor, weil meine Vorhersagen immer falsch sein werden. Was ich also machen kann, ich kann mir sagen, okay, das Ganze passiert, weil eine Korrelation in den Predictor fassen, also kann ich mir nicht einen Predictor mit Korrelation einfallen lassen. Den sogenannten Korrelationspredictor. Und für diesen Korrelationspredictor brauche ich jetzt wieder meinen

Branch History Register. Das heißt, ich versuche eben diesen Sprungverlauf, also in dem Beispiel vorher taken, non-taken, taken, non-taken so einen Sprungverlauf einfach abzuspeichern und danach zu schauen nicht nur auf den letzten Sprung, sondern auf die letzten Sprünge, gibt es denn da ein Muster für die Vorhersage des nächsten Sprungs. Ist ja eine gute, naheliegende Idee. Das heißt also, jedes Branch History Register, da stehen Muster drin. Diese Muster habe ich natürlich in Bits für Taken und Non-Taken. Also Non-Taken ist 0, Taken ist 1. Und die letzten Sprünge, die tragen jeweils ein ins Branch History Register, sind sie jetzt genommen worden oder nicht genommen worden. Ich spreche jetzt von einem Wobei K ist die Anzahl der unterschiedlichen Branches in der Vergangenheit, die ich beobachte. Also letztendlich ist es die Länge meines Bredgister-Registers, das ist das K. Und das N ist es jetzt ein 1-Bit oder ein 2-Bit-Prediktor. Also N klingt allgemein, könnte man auch ewig groß machen, aber in der Realität hat sich ja gezeigt, es bringt nichts, wenn man mehr als 2 Bits für einen Prediktor verwendet. Also ist das N hier 1 oder 2 in der Realität. Register ist einfach ein Schieberegister, das heißt der letzte Sprung trägt sich immer ein und das Ganze wird nach links draußen geschiftet. So, was ich jetzt mache ist praktisch ein zweistufiges Verfahren. Ich habe eine zweidimensionale Auswahl. Ich verwende dieses Muster, um eine von möglicherweise mehreren Pattern History Tables auszuwählen. Also ich verwende dieses Muster praktisch zur Adressierung der richtigen Table da verwende ich dann das Muster, das ich habe in meinem Branch History Register, um die richtige Table auszuwählen. Das heißt, die Pattern History Table, die enthält N Vorhersagebits pro Branch Muster, also gerade gesagt, normalerweise 2, mehr Bits nutzen nichts. Also einfache Vorgehensweise für einen Korrelationsprediktor ist die, ich habe hier meine Branch Address, also meine Sprungadresse, Sprungadresse oder dieses Branch History Register da steht ja das Muster drin so und aus diesem Muster was da drin steht hole ich mir die Zeile für die Pattern History Table ich habe also für jeden Sprung praktisch so eine Pattern History Table und das mache ich mit diesem Muster und dieses Muster, nachdem es auf die richtige Zeile gezeigt hat, das ergibt mir dann praktisch das letzte Bit, ergibt mir dann, wenn ich zwei Möglichkeiten habe, Pattern History Table 1 oder Pattern History Table 2. K gleich 1 und N gleich 1. K gleich 1 heißt, ich habe ein Branch History Register und N gleich 1 heißt, ich habe einen 1-Bit-Prediktor. Und 2 hoch K, ich habe zwei Pattern History Tables. Gut. Was ich kriege aus diesem Branch History Register, das ist das letzte Bit praktisch und dieses letzte Bit entscheidet zwischen Pattern History Table 0 und Pattern History Table 1 außerdem kriege ich aus der Adresse praktisch aus der Sprungadresse kriege ich die Zeile die ich hier habe also das ja ist hier jetzt nicht eingezeichnet es gibt ein globales Branch History Register also das hier ist nicht das Branch History Register sorry das habe ich vorhin falsch gesagt das ist natürlich die Branch Sprungadresse. Sprungadresse gibt mir die Zeile in der Pattern History Table und das Branch Register, da das hier so einfach ist, ist natürlich nur ein 1-Bit-Register, 1-Bit-Shift-Register, brauchen wir das nicht groß einzeichnen. Und dieses 1-Bit-Shift-Register, das kann 0 oder 1 haben, da steht eben drin, taken oder nicht taken. Und dieses eine Bit aus dem Branch History Register schaltet um zwischen diesen beiden Tabellen hier. wir die

richtige Tabelle aus. So ist das Ganze gemeint. Wie sieht es jetzt aus, wenn ich einen 2-2-Prediktor habe, dann ist es schon ein bisschen komplexer. Wir haben aber die gleiche Vorgehensweise. Wir haben hier wieder unsere Sprungadresse. Die Sprungadresse, da können wir eine bestimmte Anzahl von Bits nehmen, um die Pattern History Table zu adressieren. In dem Beispiel hier nehmen wir eben die unteren 10 Bits, um die Pattern History Table zu adressieren. sind bis zu 1024 Einträge, also 2 hoch 10 und über die untere Bits kriege ich die richtige Zeile raus. So jetzt habe ich 2 hoch K, nämlich 4 Pattern History Tables und jetzt ist natürlich entscheidend, mein Branch History Register, das ist nur 2 Bits lang, wir haben einen 2-2 Prädiktor, in 2 Bits kann ich 4 Zustände kodieren und diese diese Kodierung hier, die gibt mir jetzt wieder die richtige Hier steht ja das Muster drin, das Sprungmuster drin. Und dieses Sprungmuster entscheidet, welche dieser vier Tabellen ich jetzt nehmen soll. Die Sprungadresse entscheidet, welche Zeile aus der Tabelle ich nehmen soll. So, und da finde ich jetzt meine Prädiktoren. Und in dem Fall sind die zwei Prädiktoren 1,1, also stark genommen. Ja, ich werde auf jeden Fall vorher sagen, der Sprung soll genommen werden. So, und was jetzt passiert ist, wenn dieser Sprung tatsächlich genommen wird, dann bleibt dieser Eintrag unverändert. Vorhersage falsch war, dann wird natürlich dieser Eintrag hier korrigiert, dass ich beim nächsten Mal dann die aktualisierte Vorhersage in der Pattern History Table finde. So funktioniert das Ganze. Und Sie sehen schon, je nachdem wie groß ich eben mein K mache, desto mehr Aufwand muss ich treiben. Prediction. Ich verwende einmal meine Sprungadresse und dann verwende ich aber auch das Muster der vorhergehenden Sprünge, um hier zu selektieren. Also das wird natürlich dynamisch hier immer aktualisiert und dann wird natürlich die Vorhersage überprüft und gegebenenfalls korrigiert. Diese Idee, die kann man natürlich auch anders realisieren, so wie das in dem Beispiel ist, also mit sogenannten zweistufigen adaptiven Prädiktoren. Also adaptiv, das dynamisch ist, das heißt während der Laufzeit eben ein Profil erstellt wird und dann die Information genutzt wird. Und auch hier werden 2-Bit-Prädiktoren verwendet, weil das einfach das Optimum ist. Man hat so eine Art Pattern History Table und auch hier ist es wieder so, dass man die Sprungadresse als Information verwendet für die Adressierung und die Muster, die in der als zweite Information, um die richtige Pattern History Table auszuwählen. Und ja, das Ganze heißt ein bisschen anders, aber wenn man es sich jetzt genauer anschaut, dann kann man so zweistufige adaptive Prädiktoren einfach als Verallgemeinerung der Idee der Korrelationsprädiktoren ansehen. Also es ist im Prinzip das Gleiche. Ich habe das einfach nur zweistufig, ich habe einen Kontext und einmal aus der Sprungbefehlsadresse. So, jetzt kann ich diese zweistufigen adaktiven Prädiktoren, jetzt kann ich natürlich in den Kontexten, kann ich da jetzt beliebig kompliziert werden. Das heißt, jetzt geht es darum, wie aufwendig will ich meine Sprungvorhersage machen, wie viel Hardware will ich dafür spendieren und wie groß ist dann der Erfolg. sondern ich habe immer so eine Aufwands-Ertrags-Betrachtung. Wenn ich so und so viel Aufwand da jetzt reinstecke, wie viel Ertrag bringt mir das? Und deshalb wird im Folgenden ein bisschen genauer unterschiedliche Strategien miteinander verglichen, aber erstmal muss ich mir diese Strategien zusammenbauen und dann schaue ich

mir die zwei Kontexte an, einmal das Branch History Register und einmal das Pattern History Table und beim Branch History Register kann ich sagen, okay, ich nehme einfach ein globales Branch History Register für alle Branches, nicht groß, weil alle Branches ins gleiche Register schreiben. Oder ich spendiere viel, viel mehr Hardware und sage, okay, jede Sprungadresse bekommt ihr separates Branch History Register. Es gibt dann eine Branch History Table, also nicht mehr ein einzelnes Register, sondern eine ganze Tabelle. Erfordert natürlich wahnsinnigen Hardwareaufwand, weil jeder Sprung baut sich so einen Eintrag in die Tabelle rein. Oder ich versuche Befehlsgruppen zu bilden und mache so ein Per Set, für eine bestimmte Gruppe von Befehlen. Und auch die fasse ich dann wieder in eine Branch History Table zusammen. Und das gleiche habe ich natürlich für meine Pattern History Tables. Ich kann global arbeiten, das heißt ich habe nur eine Pattern History Table. Ich kann pro Sprungbefehl eine Per Address Scheme machen, also separate Pattern History Table pro Sprungbefehl. Oder ich mache wieder Gruppen und habe dann eins pro Gruppe. insgesamt neun Möglichkeiten, wie ich das Ganze kombinieren kann. Je nachdem, ob ich große oder kleine Buchstaben nehme. Und da gibt es eben globale Schemata, da gibt es per Adress Schemata und per Set Schemata. Und global für die Branch History und global bis zur Adress für Repair and History Tables. So, jetzt wollen wir uns mal ein paar von diesen Möglichkeiten anschauen. So ein globales Schema ist am einfachsten zu verstehen. Ich habe also ein Bright History Register, also keine Tabelle, sondern ein einfaches Register. Und dieses Register verwende ich als Index für meine Pattern History Table. Ein ganz einfacher degenerierter Fall, weil ich verwende nur das Branch History Register für die Adressierung hier in meiner PID & History Table. Und hier finde ich dann praktisch meine zwei Bits für die Vorhersage. Also eigentlich ist das ja keine Korrelation, weil die Sprungadresse die erste Stufe überhaupt nicht beeinflusst. zu viel. Der einzige Vorteil ist, ich kann meine Vorhersage beginnen, bevor auf dem Sprungbefehl eigentlich zugegriffen wird, weil ich ja einfach nur das Register verwende. Dafür brauche ich ja den Sprungbefehl nicht. Das heißt, die Adresse vom Sprungbefehl geht in dieses Schema überhaupt nicht ein, in das GAG-Schema. Einfach, aber schlecht. globale Branch History Register. Ich habe per Address PHT, das heißt ich habe für jeden Sprungbefehl eine eigene Pattern History Table. Das heißt also der Sprungbefehl adressiert mir hier welche Pattern History Table zu nehmen ist und das Muster im Branch History Register adressiert mir wieder die richtige Zelle, also Zeile und in der Zeile finde ich dann meine 2 Bits für die Prädiktion. So jetzt kann ich berechnen wenn ich eine 24 bit sprungadresse habe ja dann brauche ich 4 bits für das branch history register plus 2 hoch 24 für die 24 bit sprungadresse mal 2 hoch 4 mal 2 sind also 0,5 gigabyte die ich hier brauche ja weil ich habe pro history table 2 hoch 4 einträge weil ja 4 bits von der branch history Register geben ja 16 Muster, das heißt ich habe hier 16 Zeilen zu adressieren, also 2 hoch 4 Einträge und ich habe 2 hoch 24 Pattern History Tables. Also das würde 0,5 Gigabit plus die 4 Bit fürs Branchister Register und das würde natürlich bedeuten, ein bisschen zu groß. Also ich muss das Ganze ja in Hardware Perln History Table, das wird nicht funktionieren, weil es einfach zu aufwendig ist. Sowas ist leicht abzuschätzen. Aber die neuen

Möglichkeiten, die ich hier habe, die spannen natürlich einen riesengroßen Raum auf. Und dieser riesengroße Raum, den muss ich natürlich erkunden. Also wir haben gerade gesagt, okay, dieses GAP, das hier wird zu aufwendig. Das hier ist zu einfach. sind. Das werden wir heute nicht mehr schaffen, deshalb, ja, falls keine Fragen mehr sind, möchte ich das für heute ganz gerne beenden. Wir sehen uns dann kommenden Dienstag und dann untersuchen wir die anderen Möglichkeiten der Sprungvorhersage mit Korrelationsprediktor. Ich bedanke mich für Ihre Aufmerksamkeit und freue mich auf die nächste Vorlesung am Dienstag. Bitte, wenn Sie noch Fragen haben, stellen Sie sie gerne. .