Prüfung Rechnerarchitektur 30. März 2021

Prüfer: Prof. Dr. -Ing. Dietmar Fey

Beisitzer: Christian Widerspick

RISC & CISC:

Wir haben uns ja mit RISC und CISC beschäftigt, worum geht es denn bei CISC?

> CISC hat zeitlich und in der Wortbreite unterschiedlich lange Befehle, die meist historisch gewachsen sind. Die Adressierung ist komplex und die Makrobefehle werden in Mikroprogramme umgesetzt.

Und wozu haben wir so Mikroprogramme?

> Dadurch erreichen wir höhere Flexibilität, außerdem kann man Befehle emulieren, für die keine Hardware vorliegt (z.B. MUL mit Addieren), und für Abwärtskompatibilität sorgen.

Sie haben ja die Abwärtskompatibilität bei CISC erwähnt, die ist ja nicht unproblematisch , weshalb man sich RISC ausgedacht hat. Was hat man denn da herausgefunden und was macht man bei RISC jetzt anders?

> Es gab eine Patterson Studie, dass nur XYZ % der CISC Befehle in Compilern verwendet werden, aber es muss wegen der Kompatibilität trotzdem noch alle geben. Bei RISC hat man nur einen kleinen Befehlssatz mit einfachen Ausdrücken und Adressierungen, die man gut Pipelinen kann und die kein Mikroprogramm haben sondern fest verdrahtet sind.

Pipelining:

Wie sie richtig gesagt haben kann man ja RISC gut Pipelinen, wie sieht denn so ein Pipeline aus? Vielleicht können Sie ja mal eine solche Pipeline zeichnen und dann erklären was so für zeitliche Verhältnisse da am besten vorliegen.

> (Beispielhafte 5 stufige Pipeline hingemalt, Befehle eingeordnet) Die Befehle werden wie an einem Fließband verarbeitet, wenn ein Befehl eine Ressource für einen Teil des Befehls, eine Stufe der Pipeline dann, freigibt, kann der nächste Sie nutzen. Am besten sind alle Pipelinestufen dann gleich lang, da die längste Stufe den maximalen Durchsatz bestimmt.

Und was ist der maximale Durchsatz bestenfalls?

> Bestenfalls ist der Speedup gleich der Anzahl der Stufen, wenn man nicht durch Hazards behindert würde.

Diese Hazards legen einem einen ordentlichen Stein in den Weg. Was sind denn Hazards eigentlich und was passiert da?

> Es gibt Strukturhazards durch belegte Ressourcen, Datenhazards wegen Abhängigkeiten der Operanden von Befehlen und Steuerungshazards bei Verzweigungen, Sprüngen oder Funktionsaufrufen.

Bleiben wir mal bei den Steuerungshazards. Warum ist das denn ein Problem für die Pipeline und was kann man da machen?

> Die Pipeline kann nicht zwangsläufig mit dem richtigen nächsten Befehl befüllt werden, weil noch gar nicht bekannt ist, was der nächste Befehl ist. Abhilfe schaffen da Sprungvorhersagetechniken. Da gibt es statische wie Branch Taken / Not Taken, Hinting oder auch am Opcode erkennbare, wie bei BranchNotEqual zum Beispiel. Aber auch dynamische Methoden wie der Branch Loop Buffer, obwohl das eher eine Art Level 0 Cache für Instruktionen ist. Dann gibt es noch Branch History Tables, Brach Target Buffer und Branch Prediction Buffer, obwohl da die Namen leider nicht ganz eindeutig sind.

Wie würde denn so ein Branch History Table funktionieren?

> Da speichere ich ab ob ich bei einem Sprung zuletzt gesprungen bin oder nicht und nehme das letzte Mal als Vorhersage für den nächsten Sprung (1 Bit Automatengraph gezeichnet), das hilft dann schon bei einfachen Schleifen.

Aber die gibt es ja auch mit zwei Bit.

> Ja genau, da gibt es dann Zustände strongly / weakly taken / not taken, dadurch kann man einmal falschliegen bevor sich die Vorhersage ändert.

Und wo benutze ich die?

> In verschachtelten Schleifen

Und was mache ich dann für falsche Vorhersagen?

> Angenommen man ist am Anfang in strongly not taken, dann macht man ganz am Anfang zwei falsche Vorhersagen, in jeder inneren Schleife dann nur eine falsche Vorhersage und ganz am Ende noch eine in der äußeren Schleife.

Multicore:

Was anderes, warum macht man denn Multicore?

> Da gibt es mehrere Gründe, einer war, dass es in der Entwicklung der Halbleitertechnik immer größere Probleme mit der Leistungsaufnahme der Chips gab.

Sagt Ihnen die Dennertsche Skalierung etwas?

> Ja das war eine von Dennert aufgestellte Übersicht über die Skalierung der Leistungsaufnahme von Prozessoren, wenn man deren Parameter verändert (Formel, ein bisschen erklärt wie sich die Werte bei der Skalierung verändern).

Und was hat das dann letztlich bedeutet in Summe?

> Bei der Multiplikation aller Effekte kommt man auf 1, man kann also die Leistung ohne Einbußen immer weiter skalieren.

Aber das ging ja nicht ewig so, warum?

Man kommt bei der Spannung an Grenzen.

Dann gibt es ja bei Multicore noch andere Argumente.

Ja, die Leistung steigt nämlich nach Pollack nur mit der Wurzel der zusätzlichen Komplexität.

Was heißt hier Komplexität?

Das ist die Anzahl der Transistoren. Und weil man auch bei den architektonischen Maßnahmen an seine Grenzen stößt, man hat ja 95% Richtigkeit bei der Sprungvorhersage, macht man lieber mehrere einzelne Prozessoren.

Da muss man die Programme ja dann parallel ausführen, wie gut skaliert das denn?

Dazu gibt es die Ahmdahlsche Regel (Formel), bei der ein serieller Anteil und ein parallelisierbarer Anteil berücksichtigt wird.

Und was heißt das im Idealfall?

Ich bekomme mit dem Limes Anzahl Kerne gegen Unendlich einen Speedup vom Kehrwert des seriellen Anteils.

Hier auf dieser Grafik wird ja sowohl Pollak als auch Ahmdahl berücksichtigt. Können Sie mir das Mal erklären?

(Kapitel 2 Folie 18 erklären)

Und warum macht man die Prozessoren dann nicht einfach noch kleiner, also statt 144 Kernen einfach 100 Tausend Kerne auf der gleichen Fläche?

Einem Kern stehen dann kaum Transistoren mehr zur Verfügung, dadurch kann er nicht mehr sinnvoll arbeiten, wenn selbst an den wichtigsten Funktionalen Einheiten gespart werden muss. Ein Addierer braucht schließlich auch Transistoren.

Nichts zu GPUs oder Spezialprozessoren, Antworten mit Randnotizen ausschmücken, Note: Sehr Gut