Predicţia salturilor

Mihai Budiu — mailto:mihaib+@cs.cmu.edu
http://www.cs.cmu.edu/~mihaib/

16 iulie 1999

Subiect: arhitectura modernă a calculatoarelor: predicţia salturilor

Cunoştinţe necesare: cunoştinţe de arhitectura procesoarelor moderne

Cuvinte cheie: salt, dependenţă, predicţie

Contents

Textul de faţă se constituie într-un al patrulea articol din seria "Arhitectura modernă a calculatoarelor". Deşi fac tot ce îmi stă în putinţă în a face fiecare articol independent de celelalte, ar fi o risipă dacă nu aş refolosi unele dintre informaţii; din cauza asta trimit cititorul doritor de a lămuri mai multe aspecte la articole precedente; cel mai folositor este cel intitulat "Despre conducte", publicat în PC Report din decembrie 1998. Pentru cei care nu au revista, dar au acces la Internet, textul este disponibil din pagina mea de web.

1 Maşini pipeline, superscalare şi dependenţe

Procesoarele moderne sporesc performanţa printr-o multitudine de mijloace; de căpetenie este exploatarea *paralelismului* din programe. Spunem că un program posedă paralelism dacă părţi diferite din program pot fi executate simultan fără a schimba rezultatele finale.

Am ilustrat în texte anterioare cum procesoarele moderne exploatează paralelismul prezent între instrucţiuni consecutive dintr-un program, paralelism numit "la nivel de instrucţiune" (Instruction Level Parallelism, ILP). Două sunt mijloacele de căpetenie folosite pentru acest scop: tehnologia de pipeline, şi tehnologia superscalară. Voi reaminti pe scurt în ce constau acestea, şi care sunt piedicile care stau în calea exploatării lor.

Restul articolului va arăta cum piedicile mai sus-numite includ o piedică foarte importantă cantitativ: chiar instrucţiunile de salt dintr-un program. Restul articolului va arăta care sunt metodele tehnologice folosite pentru a reduce impactul negativ al salturilor.

Paralelismul superscalar este relativ uşor de înţeles: dacă un procesor obişnuit are cîte o bucată din fiecare unitate funcţională, un procesor superscalar are mai multe astfel de copii. Astfel, un procesor superscalar poate efectua simultan două adunări, pentru ca posedă două unităţi aritmetice. De asemenea, el poate de obicei lansa în execuţie mai multe instrucţiuni, pentru că are mai multe unităţi care decodifică instrucţiunile, etc. Paralelismul superscalar este întîlnit peste tot: atunci cînd o pompă de benzină angajează mai mulţi lucrători pentru a manipula pompele, exploatează paralelismul în acelaşi fel ca un procesor superscalar.

Pe de altă parte, paralelismul pipeline este inspirat după banda de asamblare de la fabricile de maşini (şi nu numai): un lucrător face şasiul, altul pune uşile, un al treilea parbrizul; toţi aceşti lucrători manipulează simultan maşini diferite; o maşină trece succesiv pe la fiecare din ei, şi în timp ce prima maşină, cu uşile deja puse, are parbrizul în curs de montare, o a doua tocmai îşi primeşte portierele.

Toate procesoarele moderne de performanţă folosesc ambele tehnici simultan.

Din păcate, în lumea calculatoarelor, lucrurile nu sunt aşa de simple ca într-o fabrică. Asta se întîmplă pentru că paralelismul se poate aplica numai dacă obiectele asupra cărora se operează sunt independente. Dar instrucţiunile unui program depind adesea una de alta; în definitiv un program nu este decît un şir de prelucrări asupra aceloraşi date. De exemplu, dacă avem două instrucţiuni consecutive: `a=b+c; d=d-a`, a doua evident nu poate fi executată pînă cînd prima nu s-a terminat, pentru că are nevoie de valoarea lui `a` calculată de prima. Spunem că între aceste instrucţiuni avem o *dependenţă*.

Cînd un procesor întîlneşte cod ca cel de mai sus, nu are mare lucru de făcut: va trebui ca unii dintre lucrători să stea nefolosiţi o vreme, pentru că nu au nimic de făcut.

2 Măsurătorile SPEC (benchmarks)

Trebuie să realizăm că, decizia dacă astfel de situaţii sunt sau nu importante, depinde enorm de *frecvenţa* lor. Dacă astfel de cazuri apar extrem de rar, atunci optimizările făcute pentru a le preveni nu vor avea un impact prea mare, pentru ca nu e mare lucru de cîştigat. Dimpotrivă, dacă astfel dependenţele sunt dese, impactul lor asupra performanţei este major.

Bine, dar dese unde? În care program? Nici măcar în interiorul aceluiaşi program nu avem întotdeauna o densitate constantă de dependenţe. Ce trebuie să măsurăm? Atunci cînd un proiectant construieşte un nou procesor, trebuie să aibă oarecare metode pentru a-l evalua.

Ei bine, experţi din industrie au pus la punct nişte suite de măsurători (*benchmarks*) pentru a evalua performanţa sistemelor de calcul. Cele mai faimoase, şi probabil şi cele mai criticate, sunt cele numite SPEC, de la Standard Performance Evaluation Corporation. Puteţi găsi amănunte despre acestea la `http://www.specbench.org/`. Scopul acestui articol este însă altul, aşa că nu voi divaga prea mult despre acest interesant subiect. Important este de reţinut că afirmaţiile cu caracter cantitativ din acest articol, care nu este clar cui se aplică, sunt făcute în contextul acestor suite de măsurători.

De exemplu, principalul personaj al acestui articol, salturile, apar în medie la fiecare şapte instrucţiuni într-un program din suita SPEC. Dacă numărul era 1 la 100, situaţia era complet diferită. Proporţia este însă mai mult decît semnificativă, şi de aceea articolul de faţă există (are un subiect).

3 Costul unui salt

Înainte de a vedea de ce sunt salturile o problemă, vom nota că problema se manifestă în faptul că execuţia unei instrucţiuni de salt durează mai mult decît a unei instrucţiuni obişnuite, cel puţin în cazul în care nu aplicăm nici una dintre tehnologiile descrise mai jos. Un exemplu tipic este ca o instrucţiune aritmetică să se execute într-un ciclu de ceas, iar un salt să dureze 3. Cît de importantă este contribuţia salturilor în performanţa programului? Cu alte cuvinte, cu cît mai încet merge programul decît în cazul în care fiecare salt ar dura şi el exact un ciclu?

Un pic de aritmetică răspunde la această întrebare: raportul este $7/(6 + 1*3)$, din cauză că 7 instrucţiuni ne costa $6+3$ cicli în loc de 7; asta înseamnă $7/9$, sau o performanţă de 77%. Am pierdut deci un sfert din performanţă.

Am mai menţionat şi cu alte ocazii că arhitecţii calculatoarelor din ziua de azi sunt gata de orice pentru o creştere de 10% a performanţei, iar multe articole publicate se mulţumesc cu chiar mai puţin de atît. În acest context 23 de procente este o cantitate uriaşă, care categoric merită diminuată.

Chiar dacă ne-am mulţumi cu aceste pierderi, nu am fi liniştiţi pentru multă vreme: pe măsură ce conductele procesoarelor devin din ce în ce mai lungi, şi superscalarele pun la bătaie din ce în ce mai multe unităţi funcţionale, penalizarea unui salt creşte din ce în ce mai mult. Asta nu pentru că durata unui salt creşte neapărat, ci pentru că mai mulţi lucrători stau degeaba; un superscalar cu două unităţi funcţionale, în mod ideal ar putea termina *două* instrucţiuni în fiecare ceas. Atunci prezentă salturilor ar duce la o performanţă de 3.5/(3+1*3) = 3.5/6 (3 cicli pentru şase instrucţiuni şi 3 pentru salt, în loc de 3.5 în total), sau 58%. Ori procesoarele moderne au uneori chiar mai multe resurse paralele decît atît; nu e păcat ca jumătate din timp să fie irosit?

4 Dependenţe ale controlului

De ce sunt salturile scumpe? Nu e prea greu de înţeles: instrucţiunile unui program se execută în mod normal în ordine crescătoare a adreselor: 0, 1, 2, 3, etc. Salturile însă perturbă această ordine, indicînd noi adrese de unde programul trebuie executat.

Dar cum funcţionează un procesor care exploatează paralelismul? Extrage mai multe instrucţiuni consecutive şi încearcă să le execute în paralel, în cazul că nu au dependenţe. Ori saltul indică faptul că acele instrucţiuni nici nu trebuie executate!

Salturile sunt categorisite drept tot dependenţe de un tip special: *dependenţe ale controlului* (control dependencies). Se numesc astfel pentru că în terminologia limbajelor de programare tot ce nu calculează se numeşte control; instrucţiunile care indică direcţia de execuţie sunt instrucţiuni de control.

4.1 Dependenţe ale controlului într-un pipeline

Voi ilustra importanţa dependenţelor controlului pentru un procesor pipeline; pentru o explicaţie mai amplă despre cum se poate citi o schemă ca cea de mai jos vedeţi articolul recomandat în introducere.

Voi presupune ca procesorul nostru are o conductă formată din următoarele stagii: un stagiu de citire a instrucţiunii, care o aduce din cache, unul de decodificare, care determină ce face instrucţiunea şi aduce operanzii de care are nevoie, stagiul principal, de execuţie, în care operaţia indicată este efectuată asupra datelor, urmat de stagiul de scriere, în care datele calculate sunt puse la locul dorit. Ţeava poate conţine şi alte stagii, dar deocamdată doar acestea ne interesează.

Problema saltului (figura 1) este următoarea: adresa unde se face saltul şi condiţia de care saltul depinde sunt calculate doar în stagiul de execuţie, şi pot fi folosite doar în stagiul de scriere. Dar în momentul cînd instrucţiunea de salt a ajuns în acest stagiu, în mod normal instrucţiunile de după ea au intrat deja în ţeavă.

Problema este că salturile condiţionale pot avea două destinaţii posibile; oricum am ghici în absenţa rezultatului final, în orice parte am lua-o, putem greşi. Ori nu vrem în nici un caz ca procesorul să execute instrucţiuni pe care programul nu le indică!

Procesoarele oferă de obicei două tipuri de salturi: condiţionale şi necondiţionale. Marea majoritate a salturilor sunt condiţionale. Putem apoi categorisi salturile în salturi la adrese fixă (dominante ca număr), salturi la o adresă calculată (relativ puţine, sintetizate pentru instrucţiunile de tip `switch-case` din C/Pascal) şi instrucţiunile de întoarcere de la un apel de subrutină. Fiecare dintre aceste tipuri de instrucţiune de salt necesită alte tehnici pentru a fi optimizată; în acest text vom insista asupra instrucţiunilor de salt condiţionat.

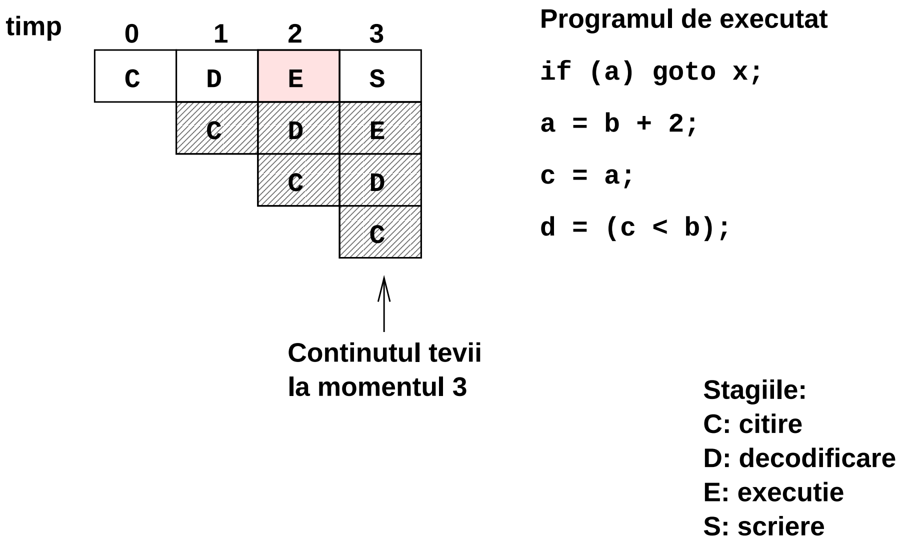

Figura 1: Evoluţia instrucţiunilor într-un pipeline în cazul apariţiei unui salt. Abia cînd saltul a ajuns în stagiul de execuţie (la momentul 2 de timp) ştim care ar trebui să fie instrucţiunea următoare. Dacă saltul trebuie luat, atunci instrucţiunile din stagiile celelalte nu trebuiau să fie executate (marcate haşurat).

5 Soluţia blocantă

Cum facem pentru a trata instrucţiunile de după un salt? Cea mai simplă şi mai costisitoare soluţie constă în a detecta instrucţiunile de salt cît se poate de devreme (în stagiul de citire, dacă se poate) şi de a bloca restul ţevii din execuţie pînă cînd adresa şi condiţia saltului sunt cunoscute. Terminologia tehnică pentru blocare este *stall*. Cel mai simplu mod de a face stall este de a injecta în mod artificial în ţeavă o instrucţiune care nu face nimic, numită “noop” (“no operation”) şi de a continua execuţia cu aceasta. Această instrucţiune fictivă se mai numeşte şi “bulă” (bubble). Figura 2 ilustrează acest procedeu.

Din păcate această soluţie asigură doar corectitudinea programului, şi nu şi eficienţa sa. Dacă procedăm astfel plătim toate costurile descrise mai sus. Ar fi grozav dacă am avea o soluţie mai eficientă. Aparent nu avem nimic de făcut, dacă nu facem o simplă observaţie.

6 Execuţia speculativă

Observaţia de care avem nevoie este că o instrucţiune, pentru a avea efecte permanente şi vizibile, trebuie neapărat să-şi scrie rezultatele undeva: fie într-un registru, fie în memorie. Asta înseamnă că atîta timp cît instrucţiunea nu a ajuns în stagiul de scriere ea nu este nicidecum vizibilă dinafară. Perfect: atunci putem face următorul lucru: putem încărca în ţeavă şi alte instrucţiuni de după cea de salt şi le lăsăm să se execute. Cînd ajungem cu saltul în stagiul de execuţie ştim deja dacă celelalte instrucţiuni trebuiau sau nu să fie executate; dacă observăm că am ales bine, suntem oameni făcuţi, pentru că saltul nu ne costă decît un ciclu, succesorii lui fiind deja gata de execuţie. Altfel recurgem la o soluţie asemănătoare cu cea de mai sus: ştergem conţinutul din începutul

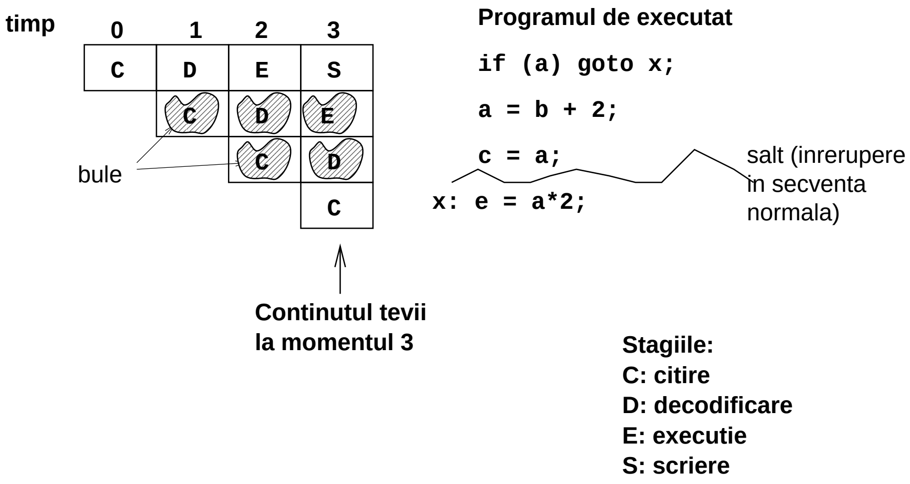

Figura 2: Metoda cea mai simplă este de a nu încărca nimic în ţeavă pînă cînd nu ştim exact care este instrucţiunea următoare; ţeava o să prelucreze atunci numai "bule".

ţevii, introducînd mai multe bule.

Acest tip de execuţie în care mizăm pe anumite instrucţiuni de a fi utile, şi în care dacă ne dăm seama că am greşit renunţăm la ele, se numeşte *execuţie speculativă* (speculation). Această tehnică este din ce în ce mai folosită în procesoarele moderne.

Observaţi că în acest caz nu avem nimic de pierdut: performanţa poate doar creşte, pentru că instrucţiunile pentru care am ghicit bine vor fi mai scurte, iar celelalte vor dura tot atît.

Un procesor superscalar poate face şi mai multă speculaţie: poate porni în execuţie ambele destinaţii ale ramurii (presupunînd că adresa destinaţie poate fi calculată rapid, înainte de a ajunge în stagiul de execuţie), după care poate alege să păstreze numai destinaţia reală.

7 Diferite scheme statice de predicţie

Întrebarea la care nu am răspuns este: "cum ghicim dacă saltul este luat sau nu?" Ei bine, există o sumedenie de scheme pentru a face acest lucru, din ce în ce mai complicate, dintre care unele le vom explora în continuarea acestui text. Să începem însă cu cele mai evidente, care iau mereu aceeaşi decizie (şi de aceea le voi numi "statice").

7.1 Salt niciodată luat

Cea mai simplă schemă este să presupunem că saltul nu este niciodată luat, şi să continuăm să executăm instrucţiunile în continuare. Schema aceasta are meritul de a fi extrem de simplu de implementat. Performanţa ei este de cam 40% (adică pentru 40% din salturi presupunerea se dovedeşte corectă). (Exerciţiu: calculaţi care este performanţa procesorului în acest caz; presupuneţi că toate instrucţiunile înafară de salt durează un ciclu, 40% din salturi durează tot un ciclu, iar restul salturilor durează 3.)

Un scurt raţionament ne va şi explica de ce schema aceasta nu are nici un fel de şanse să fie mai eficace de atît. Cea mai mare parte a timpului unui program se petrece în cicluri (dacă un program nu ar avea cicluri, execuţia lui s-ar termina imediat, pentru că viteza procesorului este de ordinul a sute de milioane de instrucţiuni pe secundă). Ori un ciclu *trebuie* să conţină cel puţin o instrucţiune de salt care este luat în mod frecvent. Pentru acest salt, metoda de mai sus va ghici mereu eronat.

7.2 Salt mereu luat

Cînd designerii au realizat acest lucru, s-au gîndit să schimbe ghicitul în sens exact opus: vor prezice că instrucţiunile de salt sunt toate luate. Calculul adresei destinaţie în general nu este o problemă, pentru că, aşa cum am văzut mai sus, majoritatea salturilor se efectuează la adrese constante, care fac parte din chiar codul instrucţiunii. În mod natural, corectitudinea unei astfel de scheme este de cam 60%.

7.3 Salt înapoi luat

Un alt rafinament al schemei este a observa că orice buclă trebuie să aibă cel puţin un salt înapoi care este luat. Măsurători pe SPEC au arătat că o schemă care prezice că orice salt înainte nu e luat şi orice salt înapoi este are performanţe mai bune decît schema precedentă.

Figura 6 arată acurateţea ghicirii pentru diferite scheme de salt. Deocamdată puteţi citi primele trei coloane; despre următoarele vom discuta în continuare.

Trebuie să observăm că, orice schemă de predicţie vom implementa, ea trebuie să fie foarte simplă şi rapidă. Nu ne putem permite să executăm un algoritm complicat de zeci de instrucţiuni pentru a economisi doi cicli de ceas! Mai mult, soluţiile trebuie să fie toate implementabile în hardware, ceea ce este o constrîngere destul de severă. Vom vedea însă că imaginaţia cercetătorilor depăşeşte toate aceste obstacole, creind scheme foarte ingenioase.

8 Predicţie dinamică cu cache-ul de instrucţiuni

O metodă foarte interesantă de a face predicţia este de a memora în cache-ul de instrucţiuni vechea comportare a unui salt: atît condiţia sa cît şi adresa de destinaţie. S-a observat, tot prin măsurători, că foarte adesea salturile tind să se facă în aceeaşi direcţie şi în acelaşi loc de mai multe ori consecutiv. Această schemă este folosită de microprocesorul PPC604.

Rafinamente ale acestei scheme au dus la crearea unui mijloc modern de anticipare a destinaţiei, numit "trace cache" (cache-urmă). Aceasta este o invenţie relativ recentă (la ora actuală nu ştiu de nici un procesor care să o folosească, dar este sigur că în curînd va fi o prezenţă comună în fiecare calculator), care practic rescrie programul în mod dinamic în cache, punînd instrucţiunile care tind să fie executate în succesiune una după alta. De pildă, dacă un salt este mereu luat, trace-cache-ul va pune instrucţiunile de dinaintea saltului şi cele de după una după alta, şi va schimba apoi condiţia saltului în cea opusă; schema aceasta permite de asemenea procesoarelor un acces mult mai eficace la cache. Sper să revin cu mai multe amănunte asupra acestei interesante tehnologii în alte articole.

9 Predicţie dinamică cu predictori locali

Soluţia cu cache-ul (nu cache-ul urmă, ci cea care menţine informaţiile despre salt) este destul de ingenioasă, dar este cam costisitoare şi destul de complicată din punct de vedere hardware; cere extragerea mai multor date din cache decît normal şi aparatură mai complicată de decodificare.

Putem simplifica schema menţinînd o tabelă separată în interiorul procesorului, şi nu în cache.

9.1 Predictorul cu un bit

Schema aceasta este extrem de interesantă, pentru că aparent face un compromis destul de mare: amestecă laolaltă informaţia despre toate salturile din program într-o tabelă unică. Algoritmul este următorul:

1. Cînd întîlnim o instrucţiune de salt notăm adresa sa proprie;
2. Apoi aplicăm asupra acestei adrese o funcţie de hash simplă care transformă adresa într-un domeniu limitat (de exemplu între 0 şi 1023). Un exemplu foarte simplu de funcţie de hash este de a păstra numai ultimii biţi din adresă (de pildă ultimii 10)[1].
3. Cu aceşti biţi indexăm într-o tabelă de 1024 de linii. La linia respectivă găsim un bit. Folosim acest bit pentru a prezice condiţia saltului. Speculăm executînd instrucţiunile de la destinaţia aparentă.
4. După ce condiţia reală este cunoscută, marcăm în tabelă informaţia despre valoarea reală a condiţiei.

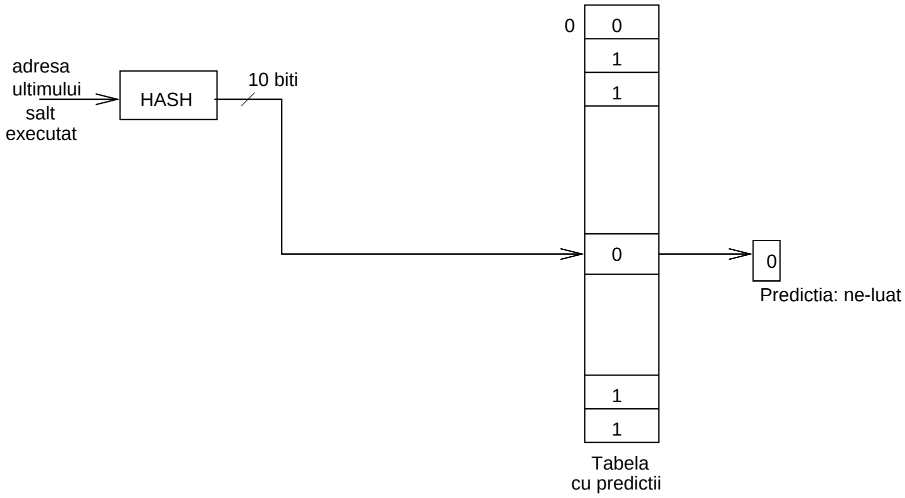

Figura 3: Predictorul cu un bit conţine o tabelă de predictori cu contoare saturate în care se indexează cu o valoare care depinde de adresa instrucţiunii de salt.

[1] În procesoarele moderne cu instrucţiuni de 32 de biţi toate adresele la care se poate sări se termină în "00" în baza doi, deci ar fi mai înţelept să păstrăm biţii 2–11 în loc de 0–10.

Metoda este extrem de ieftină de implementat şi foarte rapidă. Figura 3 arată cum este construită în hardware. Ne poate face însă să ne simţim nesiguri: funcţia de hash poate amesteca salturi independente. Este adevărat, pentru exemplul de mai sus, dacă avem două salturi în program ale căror adrese se termină în aceiaşi 10 biţi, atunci informaţia despre condiţia ambelor va fi stocată în acelaşi loc în tabel.

Şi atunci cum putem avea încredere? Ei bine, dacă puneţi această întrebare înseamnă că aţi uitat de fapt că tot ceea ce facem este doar o ghiceală; nimic nu este sigur aici. Execuţia speculativă ne asigură că nu putem strica nimic ghicind greşit. Singura care poate avea de suferit este performanţa. Dar performanţa depinde de procentul de erori. Ori avem două fenomene care ne vin în ajutor pentru a face schema de mai sus foarte rezonabilă:

- Probabilitatea ca două salturi care se execută frecvent şi întreţesut să aibă adresele în acest fel este minimă;
- În general programele manifestă "localitate spaţială"; asta înseamnă că la momente de timp apropiate se vor executa instrucţiuni de la adrese apropiate (aceasta este observaţia crucială care permite procesoarelor să folosească cu atît de mult succes cache-uri). Ei bine, asta înseamnă că este foarte puţin probabil ca două salturi aflate la distanţă atît de mare unul de altul să se execute succesiv.

9.2 Predictori cu contoare saturate

Predicţia cu 1 bit de "istorie" este bunicică, dar suferă de un simptom: este prea sensibilă la mici perturbaţii. Să presupunem că avem un salt care este mai întotdeauna luat, şi numai în mod excepţional nu este. Ei bine, iată cum se va comporta predictorul:

Salt	D	D	D	D	N	D	D	D	N	D
Predicţie	-	D	D	D	D	N	D	D	D	N
Corect	-	D	D	D	N	N	D	D	N	N

Aţi văzut slăbiciunea? Ei bine, la fiecare "Nu" predictorul va ghici prost, după care va schimba pe "Nu", deci va ghici din nou prost şi data viitoare. La fiecare schimbare facem două erori. În plus, pentru un salt care alternează la fiecare execuţie luat/ne-luat, acest predictor va greşi tot timpul (mai rău chiar decît schemele de predicţie statică). Aparent acest gen de salturi este relativ frecvent, aşa că merită să facem un efort să îmbunătăţim cumva metoda.

Soluţia este din nou la îndemînă: în loc de un bit vom folosi mai mulţi! Vom implementa pentru fiecare rînd din tabelă un mic automat finit, care va avea patru stări, ca în figura 4. Stările sunt: "Sigur Nu", "Poate Nu", "Poate Da", "Sigur Da". Automatul va face tranziţii spre dreapta la fiecare salt luat, şi spre stînga la fiecare salt ne-luat.

Este un exerciţiu simplu de a vedea că acest automat se va comporta mult mai bine pentru exemplul de mai sus (va face, desigur, erori, dar mult mai puţine). Implementarea în hardware este de asemenea banală: foloseşte ceea ce se numeşte un *contor saturat* (saturated counter). Acest contor este ca unul obişnuit, care numără în sus la fiecare salt luat şi în jos la fiecare ne-luat, dar care nu trece niciodată mai jos de 0 sau mai sus de 3. "Ghiceala" va corespunde celui mai semnificativ bit: dacă e 0, atunci nu sărim, dacă e 1 sărim.

Nu e deloc clar că putem creşte calitatea predictorului folosind mai mulţi biţi pentru contoarele saturate, pentru că atunci predictorii atunci vor fi prea insensibili la schimbări.

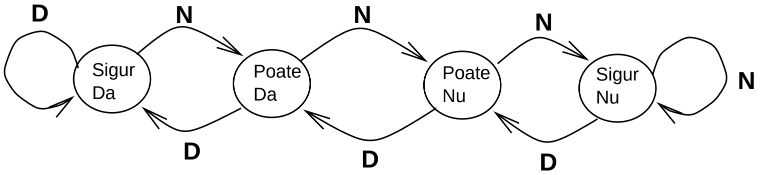

Figura 4: Automatul finit cu 4 stări pentru predicţia salturilor. Cele două stări din stînga vor prezice "Luat", iar cele două din dreapta "Ne-luat". Fiecare arc este etichetat cu acţiunea curenta; de exemplu, dacă automatul este în starea "Poate Da" şi acţiunea programului este de a nu sări (N), automatul tranziţionează în starea "Poate Nu".

9.3 Performanţă şi limitări

Sigur, nici schema asta nu e impecabilă: există pentru fiecare schemă o succesiune de salturi care o poate facă să greşească la fiecare pas. Cu toate acestea, majoritatea procesoarelor din generaţiile 1997–1998 foloseau această schemă.

Este un exerciţiu interesant de a face ceea ce se numeşte "reverse engineering": putem scrie un program simplu care să testeze comportarea predictorilor la salturi. Iată un exemplu prezentat la cursul doctoral de arhitectura calculatoarelor, ţinut de domnul profesor Randal Bryant în toamna trecută:

```
#define MARIME 1024
#define ABS(x) (((x) < 0) ? (-x) : (x))

int vector[MARIME];
int raspuns;

static void
bucla()
{
        int i;
        unsigned suma = 0;
        int prod = 1;
        for (i=0; i < MARIME; i++) {
                x = vector[i];
                unsigned ax = (unsigned)(ABS(x));
                suma += ax;
                prod *= x;
        }
        raspuns = suma + prod;
}
```

Instrucţiunea ABS se va traduce în ceva de genul:

```
  ax = x;
  if (x > 0) goto corect;
```

```
  ax = -x;
corect:
```

care conţine un salt.

Pentru a studia comportarea fiecărui predictor iniţializaţi vectorul `vector` cu valori potrivite (pozitive sau negative, după cum doriţi să fie executat sau nu saltul), după care executaţi în mod repetat bucla şi măsuraţi timpul de execuţie. Am scris în trecut un articol lung în două episoade despre cum se pot face astfel de măsurători, care include şi codul necesar; îl puteţi obţine din pagina mea de web.

Vă recomand să iniţializaţi vectorul cu trei feluri de valori: pozitive (pentru salt ne-luat), negative (pentru salt luat) şi aleatoare, în care valorile sunt generate la întîmplare. Încercaţi tot felul de formule: toate pozitive, alternant ($+ - +-$ etc.), alternant dupa o secvenţă de iniţializare ($+ + + + - + -+$ etc.), aleator, etc.

10 Predicţie dinamică cu predictori globali

Schemele de mai sus sunt simpatice, dar suferă de o boală comună: fiecare foloseşte numai informaţie *locală*, despre un singur salt. Dar adesea salturile din program sunt corelate între ele: dacă unul se execută, atunci şi un altul se execută, etc.

O schemă deosebit de ingenioasă şi eficace a fost propusă în 1993 de Yeh şi Yale Patt (Yale Patt, profesor la Universitatea din Michigan, este şi unul dintre inventatorii cache-ului cu urmă menţionat mai sus, şi este una dintre figurile cele mai proeminente din cercetarea contemporană în arhitectura procesoarelor).

Schema aceasta ţine minte rezultatele ultimelor X salturi (de exemplu X=6) şi în funcţie de acestea prezice rezultatul următorului. Schema este imediat implementabilă în hardware: ne trebuie doar un "shift register" în care introducem cîte un bit la fiecare nou salt, şi o tabelă de 2^X înregistrări în care ţinem minte rezultatul saltului următor, sau un contor saturat (figura 5). Arhitectural este destul de asemănătoare cu un predictor local; consumă cam aceeaşi cantitate de resurse hardware.

Schema aceasta pare complet paradoxală, pentru că nici măcar nu se uită *care* salturi sunt cele X, ci doar la rezultatele lor. Ca orice schemă de acest gen, valabilitatea ei poate fi doar verificată empiric; măsurători pe testele SPEC arată că schema are o comportare excelentă.

11 Predictori micşti

În fine, cel mai complicat predictor implementat în procesoarele contemporane este cel din procesorul Alpha 20264 al firmei Compaq (fost al lui Digital, acum achiziţionată de Compaq). Acest predictor este de fapt o combinaţie a trei predictori:

- Un predictor local pe 3 biţi ceva mai complicat decît cel descris aici;
- Un predictor global cu 12 biţi de istoric;
- Un al treilea predictor, care alege unul dintre ceilalţi doi predictori (din păcate nu există prea multă informaţie publicată despre cum acest al treilea predictor este implementat; am

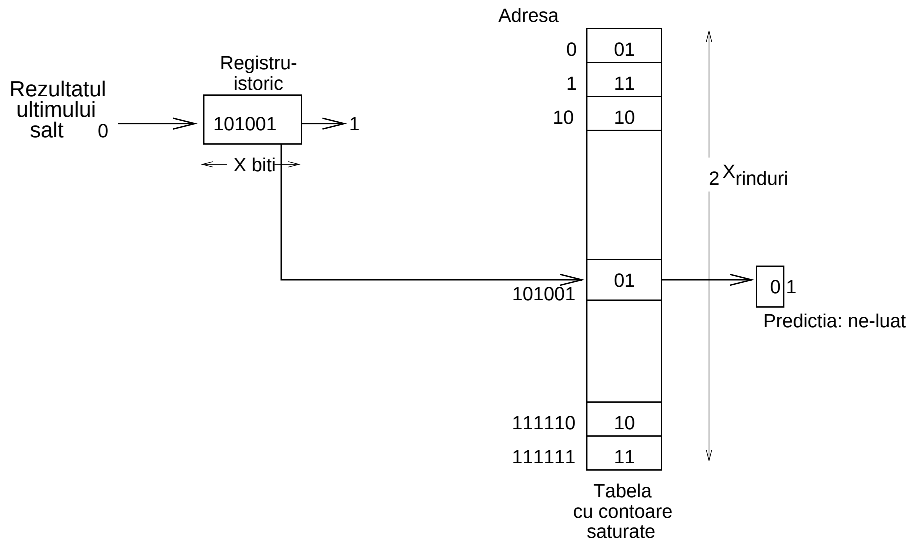

Figura 5: Predictorul global are un "shift-register" care menţine istoricul ultimelor salturi, şi o tabelă de contoare saturate care este indexată cu valoarea registrului. Performanţa acestui predictor este excelentă pentru orice model determinist de salturi, pentru că învaţă corelaţiile între salturi vecine.

avut ocazia să aflu cu ocazia unui seminar ţinut de un cercetător din grupul de hardware de la Digital, dar nu mi-am notat; din cîte îmi amintesc acest al treilea predictor măsura performanţa celorlalte două pentru fiecare salt şi-l alegea pe cel mai bun).

Performanţele tuturor predictorilor prezentaţi sunt condensate în figura 6. După cum vedeţi, predictorul mixt este excelent.

12 O clasă generală de predictori: "value prediction"

Am văzut că execuţia speculativă deschide poarta unei serii întregi de metode empirice de predicţie, unele foarte neverosimile. Cercetătorii contemporani însă împing şi mai departe aceste tehnici, pentru aplicarea lor şi înafara domeniului salturilor. De exemplu, la ultima conferinţa internaţională de arhitectura calculatoarelor ISCA (International Symposium on Computer Architecture) nu mai puţin de 5 articole din 26 discutau forme felurite de predicţie a valorii (*value prediction*).

Predicţia valorii este o generalizare a predicţiei salturilor. Ea poate de pildă fi aplicată şi pentru cazul instrucţiunilor de întoarcere de la apelul unei proceduri, a căror adresă de întoarcere este extrasă de pe stiva din memorie.

În general, predicţia valorii se face de fiecare dată cînd ceva ia mult timp pentru a fi obţinut: de pildă accesul la memorie durează foarte mult (în cazul în care datele nu sunt cache), sau execuţia anumitor operaţii aritmetice este foarte costisitoare, sau calculul destinaţiei unui salt, etc. Pentru astfel de cazuri este mai bine ca procesorul să ghicească rezultatul final decît să stea degeaba să-l aştepte pe cel corect; dacă a greşit nu-i bai: calculele pot fi reluate de la punctul de unde a început speculaţia.

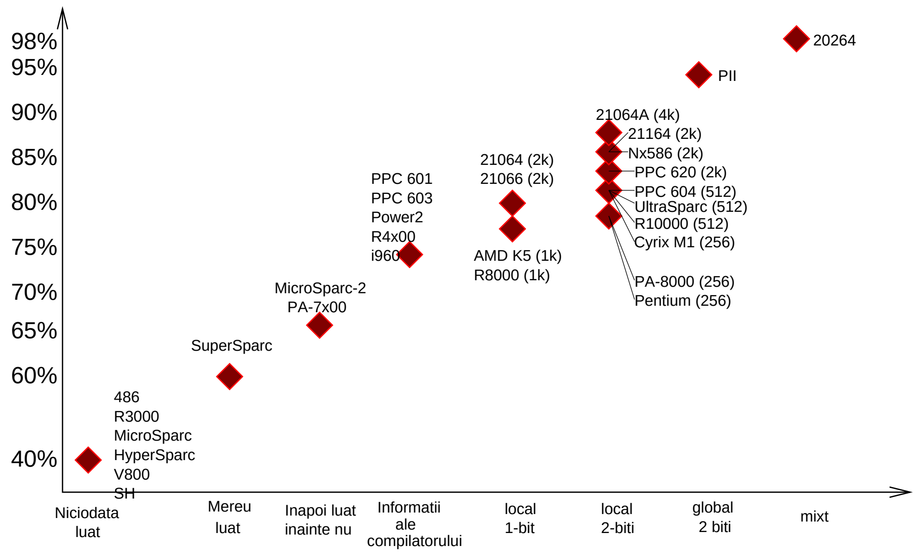

Figura 6: Performanţe ale feluritelor scheme de predicţie. Pe axa y este procentajul de ghiciri corecte. Valorile sunt măsurate pentru SPECint92. Etichetele sunt feluritele procesoare. Aceste date sunt extrase parţial din revista Microprocessor Report, 17 Martie 1995.

Măsurătorile experimentale au arătat că şi scheme banale de predicţie (de pildă: “ultima valoare a acestui obiect”, menţinută într-un cache) oferă îmbunătăţiri substanţiale.

Sper să pot dedica un articol separat tehnicii de predicţie a valorii, deşi strict vorbind, predicţia salturilor este doar un caz special. În procesoarele disponibile la momentul de faţă pe piaţă însă, putem găsi din plin implementate metode de predicţie a salturilor, dar nu prea multe de predicţia valorilor. Deci subiectul acestui articol merita o oarecare atenţie individuală.

13 Soluţii ale compilatorului

Toate metodele prezentate se bazează pe soluţii hardware: procesorul menţine informaţii suplimentare şi are circuite în plus pe care le foloseşte pentru a anticipa rezultatele unor acţiuni din viitor.

Compilatorul, ca întotdeauna în epoca modernă a sistemelor de calcul (cu precădere în ultimii 20 de ani) are însă un cuvînt important de spus pentru a ajuta procesorul să-şi sporească performanţa. Vom indica pe scurt cîteva lucruri pe care compilatorul, cîteodată în conjuncţie cu un suport specializat din partea procesorului însuşi, le poate face pentru a scădea impactul costului salturilor:

- Compilatorul poate rearanja codul în aşa fel încît salturile să fie mai uşor şi mai precis de prezis de către hardware; compilatorul poate oricînd schimba două blocuri de cod între ele şi comuta sensul testului din instrucţiunea de salt;
- Compilatorul poate desfăşura buclele (*unroll loops*). Astfel frecvenţa salturilor se reduce. De exemplu bucla:

```
for (i=0; i <10; i++)
```

```
   a[i] = b[i] * c[i];
```

poate fi transformată în:

```
for (i=0; i < 10; i+=2) {
   a[i] = b[i] * c[i];
   a[i+1] = b[i+1] * c[i+1];
}
```

Avantajul acesteia este că face de două ori mai puţine comparaţii cu 10, deci execută mai puţine salturi (metoda se poate generaliza şi pentru limite necunoscute sau impare);

- În anumite arhitecturi (ex. PPC 603) compilatorul poate insera biţi în instrucţiuni prin care îi indică procesorului cum ar trebui să se aştepte ca un anumit salt să fie executat;
- Arhitecturile moderne (de exemplu PowerPC604) oferă instrucţiuni de *încărcare condiţională*, care pot fi folosite pentru a elimina anumite salturi. De exemplu, secvenţa de cod care calculează minimumul dintre două numere:

  ```
  c = (a < b) ? a : b
  ```

  era implementată în procesoarele convenţionale în acest fel (dau echivalentul în C al codului din limbajul maşină):

  ```
    d = a - b;
    c = a;
    if (d > 0) goto corect;
    c = b;
  corect:
  ```

  Instrucţiunea de încărcare condiţională zice ceva de genul "încarcă o anumită valoare numai dacă o condiţie este adevărată". Atunci traducerea ar arăta cam aşa:

  ```
    d = a - b;
    c = a;
    c = (d > 0) ? b : c;   /* incarcare conditionala */
  ```

  Observaţi cum saltul a dispărut cu totul.
- Noua arhitectură anunţată de Intel, numită IA64, proiectată împreună cu Hewlett-Packard, şi îmbrăţişată de toţi marii fabricanţi de staţii de lucru (în afară de Sun), al cărei prim reprezentant, Merced, este deja cu doi ani în întîrziere, va conţine o generalizare a instrucţiunilor de încărcare condiţională. Aceste instrucţiuni se vor numi "instrucţiuni cu execuţie predicată" (*predicated execution*).

Aceste instrucţiuni vor avea asociat un registru de un bit. Dacă bitul este 0 instrucţiunile sunt executate. Dacă bitul este 1, atunci instrucţiunile vor fi pur şi simplu ignorate. Tehnica seamănă cu execuţia speculativă, numai că controlul este dat de compilator, iar nu de procesor.

Aparent e o mare risipă: de ce să execuţi instrucţiuni care nu le vrei, nu ar fi mai ieftin să sari peste bucata asta de cod? Aşa cum am văzut, cu cît procesorul este mai paralel (are mai multe stagii în ţeavă sau mai multe unităţi funcţionale), cu atît costul unui salt (sau a oricărei instrucţiuni care se lasă aşteptată de cele care depind de ea) este mai mare. Dacă un salt te costă cît 20 de instrucţiuni, atunci e clar mai bine să execuţi 10 instrucţiuni inutile decît să sari peste ele. De exemplu, dacă notăm cu

```
c# instructiune
```

pentru a indica faptul că `instructiune` este predicată cu bitul `c` (adică se execută numai dacă bitul respectiv este 1), atunci următorul fragment de cod:

```
if (ceva) {
   bla;
   bla;
   bla;
} else {
   tranca;
   tranca;
   tranca;
}
```

se va traduce ca:

```
   c = (ceva == 0);
   d = !c;
c# bla
c# bla
c# bla
d# tranca
d# tranca
d# tranca
```

14 Concluzii

Ştim din articole anterioare că designerii sporesc performanţa microprocesoarelor exploatînd paralelismul instrucţiunilor. Dar mai ştim şi că unele instrucţiuni nu se pot executa în paralel, pentru că sunt dependente una de alta. Am văzut în acest articol că instrucţiunile de salt implică astfel de dependenţe, pentru că ele indică de fapt *care* este instrucţiunea următoare de executat.

Faptul că instrucţiunile de salt sunt extrem de frecvente pe arhitecturile contemporane (un rezultat al operaţiilor primitive oferite de procesoare şi al modului în care compilatoarele generează cod) este extrem de neplăcut, pentru că împiedică exploatarea tuturor resurselor de calcul.

Arhitecţii au găsit o multitudine de soluţii: unele dintre ele schimbă setul de instrucîuni al procesorului, permiţînd scrierea de cod cu mai puţine salturi. Altele se bazează pe soluţii exclusiv hardware, în care procesorul încearcă să anticipeze direcţia şi destinaţia salturilor, şi să execute în mod speculativ de acolo programul, în speranţa că, dacă a ghicit corect, va cîştiga ceva timp.

Am văzut tot felul de scheme de predicţie, unele foarte stranii şi neintuitive, dar am mai văzut că ultimul cuvînt în estimarea calităţii unei scheme îl are performanţa ei pe programe reale (de obicei însă acestea sunt substituite cu teste speciale gen SPEC).

Înainte de a încheia trebuie să vă spun că de fapt bogăţia de scheme de predicţie este mult mai mare, şi acest text nu se ocupă decît de cele mai tradiţionale. Dacă subiectul vă interesează, faceţi un salt spre web şi căutaţi mai multă informaţie. Eu deja pot să prezic unde veţi "ateriza": la altavista sau o altă maşină de căutare înrudită. Se vede că am învăţat ceva de la hardware...