Execuţia speculativă

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

august 2001

Subiect: diferite forme de execuţie speculativă în procesoarele contemporane

Cunoştinţe necesare: cunoştinţe despre arhitectura microprocesoarelor

Cuvinte cheie: execuţie speculativă, thread, microprocesor

Cuprins

1 Performanţa procesoarelor

Iată un citat din ştirile de astăzi (16 august 2001): “Pe data de 27 august, la Intel Developer Forum, compania Intel va lansa noul procesor Pentium 4 la 2 Ghz”.

Deja nu mai surprinde pe nimeni apariţia unei noi generaţii de procesoare: timp de treizeci de ani noile generaţii s-au succedat celor vechi, crescînd performanţa de fiecare dată cu un factor constant. Chiar dacă constanta în sine este mică, rezultatele acumulate de-a lungul timpului sunt copleşitoare. Pentru a măsura distanţa parcursă, este suficient să ne uităm la Intel 4004, primul microprocesor, care avea o viteză de ceas de 0,7 Mhz. Creşterea înregistrată în exact 30 de ani este de 2 Ghz/0.7 Mhz = 2875 de ori!

Performanţa computaţională a calculatoarelor a crescut însă şi mai mult în această perioadă. Acest lucru este posibil pentru că performanţa unui microprocesor nu depinde doar de frecvenţa ceasului, ci şi de numărul de instrucţiuni care pot fi executate într-un ciclu de ceas. Procesoarele au evoluat enorm şi în această privinţă: Intel 4004 lucra la o singură instrucţiune la un moment dat, şi îi trebuiau mai mulţi cicli de ceas pentru a o executa complet, Pentium 4 poate avea simultan în execuţie pînă la 126 de instrucţiuni diferite, şi poate termina execuţia mai multor instrucţiuni în fiecare ciclu.

În acest articol voi discuta despre contribuţia la performanţă a micro-arhitecturii şi compilatoarelor şi voi ignora contribuţia ceasului. Voi discuta atît procesoare contemporane, dar şi unele aflate încă pe "planşetele" designer-ilor, care încă nu au făcut saltul din laboratoarele de cercetare în fabrici.

Am mai scris articole în PC Report despre arhitectura procesoarelor moderne (unele sunt menţionate în finalul acestui articol); în articolul de faţă mă voi concentra asupra unei singure tehnologii, şi anume *execuţia speculativă* a codului.

2 Paralelismul din programe

Pentru a motiva folosirea execuţiei speculative, trebuie să subliniem încă odată rolul paralelismului în performanţa sistemelor de calcul. Dacă fixăm frecvenţa ceasului, singura metodă prin care putem creşte performanţa este să executăm mai multe instrucţiuni în aceeaşi perioadă de timp. (Putem face ca fiecare instrucţiune să dureze mai puţini cicli de ceas, dar beneficiile pe care le putem extrage cu această metodă sunt limitate.) O sursă de performanţă este deci execuţia mai multor instrucţiuni *simultan*. Toate procesoarele moderne folosite în calculatoare (nu neapărat şi cele din sisteme de control) sunt *superscalare*, putînd executa mai multe instrucţiuni în paralel. De exemplu, Pentium 4 are 8 unităţi funcţionale care pot opera în paralel.

Aparent punem deci mai multe unităţi computaţionale în paralel şi performanţa creşte. În realitate lucrurile sunt mult mai complicate, pentru că nu oricare două instrucţiuni dintr-un program se pot executa simultan. De exemplu, dacă o instrucţiune foloseşte rezultatul alteia, atunci prima trebuie să-şi termine execuţia înainte ca a două să înceapă. Acest fenomen se numeşte *dependenţă* între cele două instrucţiuni.

Există două feluri de dependenţe între instrucţiuni: dependenţe de date şi dependenţe de control. Acestea sunt ilustrate în figura 1.

Putem să ne punem întrebarea: cîte instrucţiuni independente există într-un program? Cît de multe instrucţiuni putem executa potenţial în paralel, presupunînd că avem un procesor ideal, cu o infinitate de resurse? Care tipuri de dependenţe impun mai multe constrîngeri?

Pentru a răspunde la astfel de întrebări, la începutul anilor '90 mai mulţi cercetători au făcut nişte studii "limită", care încercau să estimeze paralelismul total disponibil într-un program (cu alte cuvinte, care este limita superioară pentru paralelismul care poate fi exploatat de hardware).

Un astfel de studiu, a fost publicat de doi cercetători de la universitatea Stanford în 1992; titlul său este "Limite ale dependenţelor de control asupra paralelismului". Voi rezuma aici doar

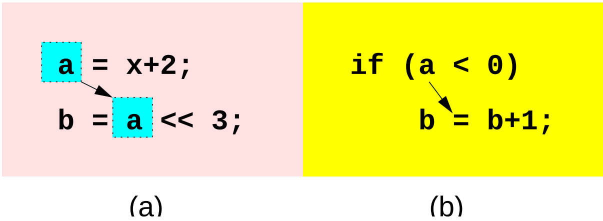

Figura 1: *Dependenţe (a)* **dependenţă de date:** *instrucţiunea a doua are nevoie de rezultatul primeia (b)* **dependenţă de control:** *instrucţiunea a doua trebuie să aştepte evaluarea condiţiei pentru a şti dacă trebuie să fie executată.*

concluziile principale ale studiului, care sunt foarte interesante.

Putem clasifica programele analizate în acest studiu în două mari categorii: programe “numerice”, scrise în FORTRAN, care manipulează matrici mari şi au structuri de date foarte simple, şi programe ne-numerice, care sunt scrise în C, au structuri de date complicate alocate dinamic şi folosesc pointeri. Această categorisire este relativ standard; suita de programe SPEC, cea mai folosită pentru a evalua performanţa sistemelor de calcul, conţine o mixtură de astfel de programe.

Natura paralelismului este diferită pentru cele două categorii: programele în FORTRAN exhibă mai mult paralelism şi mai multă regularitate în calcul (de exemplu, instrucţiunile de salt sunt mai predictibile). Dacă socotim numai dependenţele de date, paralelismul variază între 45 şi 3200 de instrucţiuni simultan pentru programele ne-numerice şi între 800 şi 300000 (sic!) pentru programele FORTRAN.

Dacă însă ne uităm şi la dependenţele de control, situaţia se schimbă în mod dramatic: paralelismul disponibil coboară la o valoare sub 10 pentru programele ne-numerice, şi la valori între 2 şi 60000 pentru cele numerice. Acest lucru se întîmplă pentru că instrucţiunile de salt sunt foarte frecvente în programe: în medie, una din 7 instrucţiuni este un salt. Dacă în plus presupunem că calculatorul nu poate executa două salturi simultan, paralelismul coboară la o valoare sub 3 pentru programele ne-numerice şi la un maxim de 400 pentru cele numerice.

Din acest studiu putem extrage următoarea concluzie:: *dacă executăm instrucţiunile programului în ordine, respectînd dependenţele de control, nu putem creşte performanţa programului prea mult:* pur şi simplu, nu există suficient paralelism în program. Acest lucru este foarte pregnant pentru programele ne-numerice, care constituie majoritatea covîrşitoare a programelor care se execută pe un desktop contemporan.

3 Execuţia speculativă

Dacă vrem să exploatăm mai mult paralelism, trebuie să facem ceva deosebit; nu putem executa instrucţiunile din programe în ordine. Soluţia este să executăm cod *înainte* de a fi siguri că trebuie executat; în felul acesta, dacă mai tîrziu aflăm că am anticipat corect, vom avea rezultatele precalculate. Aceasta este execuţia speculativă.

Există două feluri de execuţie speculativă:

- Care prezice instrucţiunile care se vor executa în viitor şi le execută mai devreme;

- Care execută instrucţiuni în toate direcţiile posibile simultan şi păstrează numai rezultatele corecte.

În general microprocesoarele execută programele după prima strategie. A doua strategie este însă adesea folosită în circuitele hardware. Figura 2 ilustrează beneficiile execuţiei speculative.

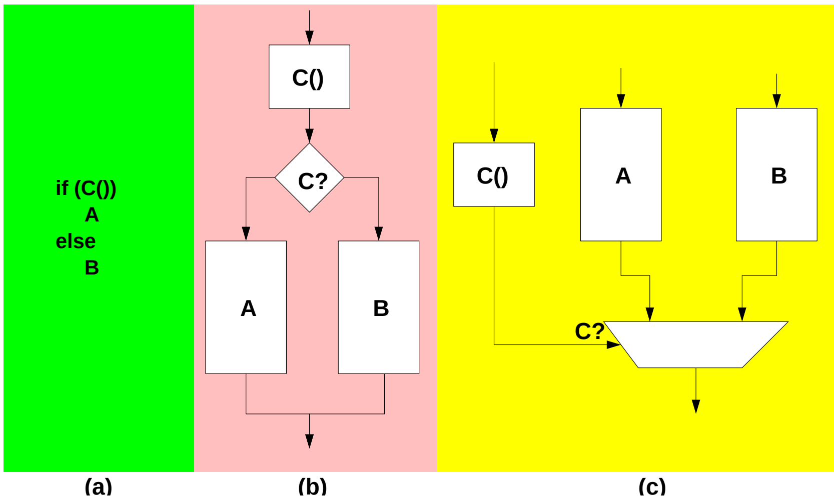

Figura 2: *(a) Un fragment de program în C (b) Execuţie a programului care respectă dependenţele de control: întîi trebuie evaluată condiţia C, şi abia după aceea se poate executa blocul A sau blocul B, depinzînd de rezultat. (c) Execuţia speculativă execută simultan A, B şi C. Apoi rezultatul evaluării condiţiei este folosit pentru a alege rezultatele corecte.*

Lucrurile nu sunt însă chiar aşa de simple: ce se întîmplă cu rezultatele speculaţiei dacă am executat ramura greşită? În acest caz va trebui să distrugem rezultatele parţiale calculate şi să recalculăm pe ramura corectă. Execuţia speculativă elimină dependenţele de tip control din program, dar nu poate elimina dependenţele de date.

Execuţia speculativă poate fi incorectă din două motive:

- Bucata de cod pe care am ales-o nu trebuia executată de loc: de exemplu, alegem să executăm A, dar trebuia să executăm B;

- Bucata de cod pe care o executăm speculativ are nevoie de informaţii care încă nu au fost calculate: de exemplu, o valoare calculată de C este necesară în calculul lui A.

În concluzie, pentru a implementa execuţia speculativă avem nevoie de următoarele ingrediente:

alegere: Un mecanism care alege codul care va fi executat în viitor;

detecţie: Un mecanism care depistează cînd execuţia speculativă este eronată, fie pentru că am ales o ramură greşită, fie pentru că am ignorat o dependenţă de date;

reparaţie: Un mecanism care permite ca execuţia speculativă greşită să fie des-făcută (rezultatele ei să fie "şterse");

reluare: Un mecanism care ne permite după o eroare să reluăm execuţia pe calea corectă.

În cele ce urmează voi discuta pe scurt în ce fel este implementată execuţia speculativă în procesoarele de astăzi, şi apoi cum arată propunerile pentru procesoarele viitorului. Procesoarele contemporane execută speculativ cod la nivel de instrucţiune, pe cînd cele viitoare vor suporta probabil execuţia speculativă la nivel de fir de execuţie (thread).

Execuţia speculativă poate fi implementată în software, în hardware, sau folosind o mixtură a ambelor tehnici.

4 Execuţia speculativă în software

Execuţia speculativă implementată în software este folosită în cazul procesoarelor paralele. Avem atunci de a face cu programe pe care vrem să le executăm pe mai multe procesoare simultan, dar care nu au fost scrise în mod paralel. Cînd compilatorul nu poate paraleliza automat codul, poate recurge la execuţia paralelă speculativă. Iată un astfel de exemplu:

```
for (i=0; i < N; i++)
        a[b[i]] = f(i);
```

Să presupunem că `N=2`, şi că evaluarea funcţiei `f` nu are "efecte laterale".

- Dacă matricea `b` conţine valorile `[1,1]`, atunci cele două atribuiri din buclă trebuie executate una după alta, pentru că amîndouă modifică variabila `a[1]`.
- Dacă matricea `b` însă conţine valorile `[1,2]`, cele două atribuiri se pot efectua în paralel, pentru că folosesc indici diferiţi în `a`.

Un compilator nu poate în general şti care sunt valorile din vectorul `b`, şi ca atare nu va paraleliza acest cod. Dacă `N` este mare şi ştim că în vectorul `b` nu există două valori identice, atunci putem distribui toate aceste operaţii pe mai multe procesoare, fiecare procesor efectuînd unele dintre ele.

Lawrence Rauchwerger a fost primul care a studiat în detaliu execuţia speculativă complet în software; în schema pe care o propune, compilatorul generează un program cu următoarea structură:

- Execută codul speculativ în paralel pe mai multe procesoare, colectînd rezultatul într-un vector auxiliar `aux`;
- Monitorizeaza valorile din vectorul `b`;
  - Dacă în final nu există două valori identice în `b`, copiază rezultatul din variabila `aux` în variabila `a`;
  - Dacă se depistează două valori identice în `b`, execută versiunea originală secvenţială a codului şi ignoră variabila `aux`.

Frumuseţea acestei scheme este că, atunci cînd speculaţia este corectă, programul se execută mult mai repede, iar cînd speculaţia este incorectă, nu se pierde foarte mult timp (se iroseşte doar timpul pentru o execuţie paralelă, care e mai mic decît cel pentru execuţia secvenţială). Preţul suplimentar pe care îl plătim este execuţia codului care verifică dacă valorile din vectorul `b` se suprapun.

Observaţi toate ingredientele pe care le-am descris mai sus:

alegere: în cazul nostru, se vor executa viitoarele iteraţii ale buclei `for`;

detecţie: cod suplimentar care monitorizează accesele la variabila `b`;

reparaţie: variabila auxiliară `aux` stochează rezultatele intermediare, fără a "polua" variabila reală `a`;

reluare: codul generat de compilator, care include atît versiunea paralelă cît şi pe cea secvenţială.

5 Execuţie speculativă în microprocesoare

5.1 Soluţia hardware

O schemă de execuţie speculativă implementată complet în hardware este folosită la ora actuală de toate microprocesoarele moderne: PowerPC 620, MIPS R10000, arhitectura P6 de la Intel, AMD K5 şi succesorii acestora. Multe din ingredientele necesare le-am discutat în alte articole din PC Report:

alegerea: este făcută circuitele de predicţia salturilor, care "ghicesc" direcţiile salturilor condiţionale înainte ca acestea să fie executate;

detecţie: cînd rezultatul saltului este cunoscut, acesta este comparat cu valoarea prezisă;

reparaţie: procesorul foloseşte o structură numită *reorder buffer*, în care face toate modificările (fără a modifica regiştrii adevăraţi);

reluare: în cazul unei erori, registrul PC, care indică instrucţiunea curentă, este pus să indice spre instrucţiunea care trebuia să fie executată după salt.

Deşi implementarea în hardware este foarte sofisticată, ideea este relativ simplă:

- Procesorul lansează în execuţie instrucţiuni înainte de a şti dacă salturile spre acele instrucţiuni vor fi sau nu executate;
- Aceste instrucţiuni se execută, dar rezultatele lor sunt scrise fie în nişte regiştri "ascunşi" (shadow registers), fie într-un buffer (dacă instrucţiunile modifică memoria);
- Cînd instrucţiunea de salt este în fine evaluată, se decide dacă speculaţia a fost corectă:
  - Dacă saltul a fost corect prezis, regiştrii ascunşi sunt copiaţi în cei corecţi, şi informaţiile din buffer sunt trimise spre memorie.
  - Dacă saltul a fost incorect prezis, regiştrii ascunşi sunt ignoraţi, buffer-ul este golit şi execuţia este repornită de la instrucţiunea corectă.

5.2 Soluţia mixtă

Arhitectura IA-64 de la Intel oferă o altă soluţie pentru execuţia speculativă, în care compilatorul colaborează cu procesorul. Fiecare instrucţiune este etichetată cu o valoare de 1 bit, numită *predicat*. Dacă predicatul unei instrucţiuni este 1, atunci instrucţiunea se execută, altfel instrucţiunea este ignorată. Dacă notăm cu

```
p# a = b + c
```

faptul că instrucţiunea `a = b + c` se execută numai dacă `p=1`, atunci un programul:

```
if (a < 0)
        b = b + 1;
else
        d = b * 2;
```

se va traduce în ceva de genul:

```
1# p = a < 0   /* predicatul este 1: instructiunea se executa neconditionat */
1# q = not p
p# b = b + 1
q# d = b * 2
```

Dacă are destule resurse, microprocesorul va executa toate aceste instrucţiuni simultan.

6 Execuţie speculativă la nivel de thread

Execuţia speculativă la nivel de fir de execuţie (thread) este un subiect de cercetare foarte fierbinte; se publică în continuare foarte multe articole pe această temă, dar nici un procesor încă nu implementează astfel de scheme. În această secţiune voi ilustra pe scurt una dintre propuneri; legături web spre alte proiecte puteţi găsi în finalul acestui text.

Motivaţia pentru acest gen de cercetare vine din faptul că, în viitorul foarte apropiat, din cauza miniaturizării, arhitecţii vor avea atît de multe resurse încît vor putea implementa mai multe procesoare pe aceeaşi pilulă de siliciu. Desigur, aceste procesoare pot executa fiecare alt program, dar cercetarea pe care o voi descrie aici discută despre cum mai multe procesoare pot colabora la execuţia *unui singur program secvenţial.*

Cheia este, desigur, execuţia speculativă: programul este împărţit în mai multe fragmente de cod, în general de zeci pînă la mii de instrucţiuni fiecare. Fiecare fragment este un *thread*, care este executat în paralel pe un alt procesor. Unul dintre thread-uri este cel "corect"; celelalte execută părţi din "viitorul" probabil al programului.

Să ne uităm la un simplu exemplu:

```
while (! gata) {
        ...
        x = hash[index1];
        ...
        hash[index2] = y;
}
```

Dacă valorile `index1` şi `index2` sunt diferite pentru toate iteraţiile, toate accesele făcute de buclă în tabela `hash` se pot efectua în paralel. Dacă însă valoarea lui `index2` dintr-o iteraţie este aceeaşi cu valoarea lui `index1` dintr-o iteraţie ulterioară, între valorile corespunzătoare din `hash` se stabileşte o dependenţă de date, care împiedică execuţia paralelă. În figura 3 (b) ilustrăm o posibilă execuţie secvenţială a acestui program; în dreptunghiuri am ilustrat fiecare iteraţie. În figura 3 (c) ilustrăm cum s-ar putea desfăşura execuţia acestui program pe un multiprocesor cu 3 procesoare, cînd fiecare iteraţie a buclei este un thread separat.

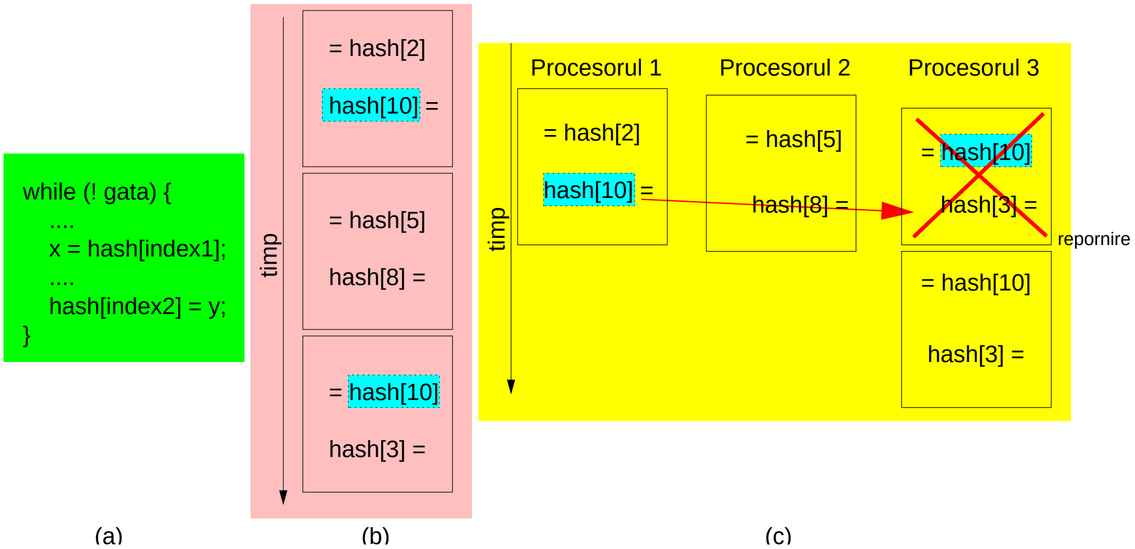

Figura 3: *Execuţia unui program (a) pe un uniprocesor (b) sau pe un multiprocesor cu execuţie speculativă (c). În (c) procesorul al treilea descoperă că a violat o dependenţă datorită execuţiei speculative, şi ca atare rezultatele pe care le-a calculat sunt distruse şi execuţia iteraţiei sale este reluată.*

Observaţi că, spre deosebire de schema software-pur pe care am descris-o mai devreme, în caz de speculaţie eronată aici nu reluăm întreg procesul de calcul, ci doar partea care a fost eronată.

Ingredientele acestei scheme sunt următoarele:

alegere: au fost propuse foarte multe scheme diferite de separare a programului în thread-uri; cele mai multe soluţii implică compilatorul, care inserează instrucţiuni suplimentare pentru pornirea şi oprirea thread-urilor. Schemele cele mai populare construiesc un thread din fiecare iteraţie a unei bucle (ca în exemplul nostru), sau execută apelul unei proceduri în paralel cu codul care urmează procedurii.

detecţie: una dintre cele mai elegante scheme foloseşte cache-urile procesoarelor şi protocolul de coerenţă al acestora (mai multe detalii urmează mai jos);

reparaţie: cache-ul este folosit şi pentru reparaţie după o speculaţie greşită: în figura 3, cache-ul procesorului 3 este golit după detecţia dependenţei eronate;

reluare: reluarea este obţinută re-setînd registrul PC la începutul thread-ului.

Dintre toate thread-urile, unul singur este “cel mai bătrîn”: acesta se execută ne-speculativ, şi poate face orice modificări. Celelalte thread-uri sunt ordonate după vîrstă: în figura 3 thread-ul de pe procesorul 1 este cel mai vechi, cel de pe procesorul 2 este următorul şi cel de pe procesorul 3 este cel mai nou. Thread-urile 2 şi 3 se execută speculativ, pînă cînd thread-ul 1 se termină. Apoi 2 devine cel mai bătrîn, şi procesorul 1 poate porni un thread-ul 4 (care nu e ilustrat în figură).

Fiecare thread face toate citirile şi scrierile din memorie folosind cache-ul propriu. Cînd un thread citeşte un cuvînt în cache, îl marchează ca accesat (de exemplu, thread-ul 3 marchează cuvîntul de la adresa `hash[10]`). Cînd un thread modifică un cuvînt, trimite această informaţie tuturor celorlalte thread-uri mai tinere decît el. Cînd un thread tînăr a folosit o valoare care apoi este modificată de un thread mai bătrîn (`hash[10]` în figură este citit de thread-ul tînăr 3 înainte de modificarea thread-ului bătrîn 1), mesajul de modificare care vine de la thread-ul 1 (indicat de săgeata roşie din figură) îi indică thread-ului 3 faptul ca a speculat incorect. Cînd un thread detectează o speculaţie eronată, se sinucide şi reporneşte.

Thread-urile se pot termina numai în ordinea în care au fost create: chiar dacă thread-ul 3 se termină înainte de 1, trebuie să aştepte pînă cînd 1 s-a isprăvit şi a anunţat toate modificările sale, ca să vadă dacă nu cumva unele din valorile pe care le-a folosit el însuşi au fost ilegale.

7 Rezumat

În acest articol am discutat despre paralelismul prezent în programe, care poate fi exploatat pentru a executa programele mai rapid. Am văzut că există două feluri de dependenţe care fac ca instrucţiunile să *nu* fie paralele: dependenţe de date şi dependenţe de control. Ambele tipuri de dependenţe limitează paralelismul, dar dependenţele de control au impact foarte important, limitînd paralelismul în programe ne-numerice la valori între 2 şi 8.

Am văzut că execuţia speculativă încearcă să elimine impactul dependenţelor de control, executînd cod înainte de a fi certă necesitatea lui. Am văzut de asemenea că execuţia speculativă poate fi implementată în multe feluri: în software, în hardware, sau cu suportul amîndurora; de asemenea, am văzut că putem specula la nivel de instrucţiuni sau de thread.

Problema selecţiei thread-urilor şi a suportului care trebuie să fie oferit de hardware este încă un subiect foarte activ de cercetare; cu certitudine însă speculaţia la nivel de thread va fi un ingredient al procesoarelor viitoarului.

8 Alte surse de informaţie

- Articolele mele anterioare despre arhitectură din PC Report:
  - Despre dependenţe şi metoda redenumirii regiştrilor, PC Report din iulie 1999: `http://www.cs.cmu.edu/~mihaib/articles/articles.html#redenumire`.
  - Despre predicţia salturilor pentru suportul execuţiei speculative, PC Report din august 1999: `http://www.cs.cmu.edu/~mihaib/articles/articles.html#predictie`.
  - Despre evoluţia arhitecturii microprocesoarelor şi implementarea de thread-uri direct în hardware, PC Report din martie 2000: `http://www.cs.cmu.edu/~mihaib/articles/articles.html#micro`.

– Despre coerenţa cache-urilor în multiprocesoare, PC Report din noiembrie 1998: `http://www.cs.cmu.edu/~mihaib/articles/articles.html#smp`.

- Studiul menţionat în acest articol, despre limitele paralelismului existent în programe şi depăşirea lui prin execuţia speculativă: "Limits of Control Flow on Parallelism", de Monica Lam şi Robert Wilson de la universitatea Stanford, publicat în 1992 în conferinţa International Symposium on Computer Architecture `http://suif.stanford.edu/papers/lam92.ps`.

- Proiecte universitare de cercetare despre execuţia speculativă la nivel de thread pentru multiprocesoare:

  – Proiectul *Stampede* la universitatea Carnegie Mellon este cel pe care l-am descris în acest articol: `http://www.cs.cmu.edu/~stampede`
  – Proiectul *Multiscalar* la universitatea Wisconsin-Maddison: `http://www.cs.wisc.edu/~mscalar`
  – Proiectul *Hydra* la universitatea Stanford: `http://ogun.stanford.edu`
  – Cercetare la universitatea Politehnică din Catalunya: `http://www.ac.upc.es/homes/antonio/specula.html`
  – Proiectul *IACOMA* la universitatea Illinois-Urbana: `http://iacoma.cs.uiuc.edu/iacoma`
  – Cercetarea lui Lawrence Rauchwerger la universitatea Texas A&M: `http://www.cs.tamu.edu/faculty/rwerger`
  – Proiectul *M-Machine* la universitatea Stanford (migrat de la universitatea MIT): `http://cva.stanford.edu/m-machine/cva_m_machine.html`
  – Proiectul *TimeWarp* la universitatea din Waiklao, Noua Zeelandă: `http://www.cs.waiklao.ac.nz/timewarp/wengine/index.html`
  – Proiectul *Multiplex* la universitatea Purdue: `http://dynamo.ecn.purdue.edu/~mux`