L. N. VINŢAN - *Direcţii de cercetare în domeniul sistemelor multicore / Main Challenges in Multicore Architecture Research*, Revista Romana de Informatica si Automatica, ISSN: 1220-1758, ICI Bucuresti, vol. 19, nr. 3, 2009, http://www.ici.ro/RRIA/ria2009_3/index.html

Direcţii de cercetare în domeniul sistemelor *multicore*

Main Challenges in Multicore Architecture Research

Lucian N. VINŢAN[1,2]

[1]Universitatea "Lucian Blaga", Str.E. Cioran, Nr. 4, Sibiu - 550025, ROMÂNIA, Tel./Fax:++40-269-212716, E-mail: lucian.vintan@ulbsibiu.ro, http://webspace.ulbsibiu.ro/lucian.vintan
[2]Academia de Ştiinţe Tehnice din România, www.astr.ro

Rezumat: S-a realizat un studiu asupra impactului arhitecturilor *multicore* şi *manycore* în cadrul ingineriei calculatoarelor. S-a pornit de la geneza acestor sisteme, determinată de limitele paradigmei monoprocesor. S-au identificat şi s-au analizat în mod sistematic, pe baza unei literaturi de specialitate recente, următoarele provocări importante pentru cercetarea şi dezvoltarea acestor sisteme: arhitecturi *multicore* omogene vs. eterogene, exploatarea sinergică a tipurilor de paralelism, ierarhia de memorii *cache*, coerenţa şi consistenţa variabilelor partajate, reţele de interconectare, modele de programare paralelă, paralelizarea aplicaţiilor, simularea ca instrument de cercetare, *benchmarking*, explorarea automată a spaţiului parametrilor, arhitecturi *multicore* cu procesări anticipative, putere consumată, disipaţie termică. Concluzia de bază este că sunt necesare progrese importante în toate aceste domenii, pentru ca proiectarea şi utilizarea sistemelor *multicore* şi *manycore* să fie adecvate. Cu sau fără voia noastră, aceste sisteme vor constitui dispozitivele de calcul universale. Abordarea oricăreia dintre aceste provocări trebuie să fie una de tip holistic. S-a arătat cum vor schimba aceste noi arhitecturi, paradigma ştiinţei ingineriei calculatoarelor. În particular, programarea acestor sisteme constituie o provocare majoră, pentru care nu suntem încă pregătiţi suficient.
Cuvinte cheie: arhitectura calculatoarelor, sisteme *multicore* şi *manycore*, sisteme paralele

Abstract: It is presented the impact brought by the new multicore and manycore systems in the computer engineering field. There were explained the limits of the actual mono-processor paradigm that involved the multicore shift. A state of the art in multicore architecture and compiler research was presented. We identified and analysed some important research challenges in multicores' research and development, like the followings: homogenous vs. heterogeneous multicores, synergistic exploitation of parallelism grains, cache hierarchy, cache coherence and consistency protocols, interconnection networks, parallel programming models, applications' parallelisation methods, simulation, benchmarking, automatic design space exploration, speculative multicore architectures, power consumption and thermal effects. There are necessary important progresses related to all these scientific challenges, in order to adequately develop and program multicore and manycore systems. The research approach must be a holistic one for each

of the above topics. It is shown how these systems would change the computer science and computer engineering paradigm.
Keywords: advanced computer architecture, parallel computing, multicore and manycore

Geneza arhitecturilor tip *multicore*

Arhitecturile de procesoare de tip *single-core*, care să exploateze în mod agresiv paralelismul la nivel de instrucţiuni prin execuţii speculative de tip *out of order*, trebuie cercetate şi dezvoltate în continuare, deşi frecvenţa tactului nu mai poate creşte prea mult, din cauza consumului de putere dinamică ($Pd=kCV^2f$) şi a disipaţiei termice. La actualele frecvenţe de tact de câţiva GHz, densităţile de putere din *chip*-uri sunt enorme, de câteva sute de W/cm^2. Supercalculatoarele actuale, cu mii de procesoare, consumă puteri de ordinul MWatt. Totodată, la o frecvenţă de tact de 14 GHz (70 ps), estimabilă în anul 2014, întârzierea semnalului pe 5 mm de conductor devine enormă (390 ps). Evident că, în aceste condiţii, creşterea frecvenţei de tact practic nu mai este posibilă, performanţa putând creşte doar prin inovaţii arhitecturale. Provocarea esenţială aici constă în determinarea compromisului optimal între performanţa procesării (*Instructions Per Cycle* – IPC) şi complexitatea arhitecturală (puterea consumată, disipaţia termică, aria de integrare şi bugetul de tranzistori). Astfel, metrici de evaluare de tip MIPS *(Million Instructions per Second) per watt, MIPS per area of silicon* etc. sunt tot mai frecvent utilizate. Aceste microarhitecturi monoprocesor vor exploata paralelismul la nivel fin din cadrul aplicaţiilor cu secvenţialitate intrinsecă (scrise în limbaje secvenţiale). În acest sens pot fi avute în vedere inclusiv metode şi tehnici de compilare adaptivă a codului obiect, bazate pe informaţii de tip *profilings* (tipul algoritmilor, gradul de utilizare al resurselor *hardware*, nivelul şi granularitatea paralelismelor la nivelul *thread*-urilor etc.). Aceste metode pot determina adaptarea sau reconfigurarea microarhitecturii la cerinţele aplicaţiei, în vederea maximizării IPC –ului şi minimizării puterii consumate. Şi totuşi, metodele de exploatare a ILP-ului (*Instruction Level Parallelism*) au ajuns la o oarecare saturaţie – *ILP Wall* (*superpipeline* cu frecvenţe mari de tact, procesarea *out of order*, *branch prediction*, *trace-cache*, procesări speculative, metode de *scheduling* static, metode de eliminare a *memory-wall* etc.)

Sisteme *multicore* şi *manycore*

Soluţia cea mai frecventă la ora actuală pentru creşterea performanţei şi evitarea limitărilor anterior schiţate, constă însă în dezvoltarea de arhitecturi *multicore* şi *manycore*. Deşi vor face mai dificilă activitatea calculatoriştilor, acestea oferă o rată de performanţă/Watt mai bună decât sistemele monoprocesor, la o performanţă similară. În plus, aceste sisteme care exploatează paralelismul *thread*-urilor, au şanse mai mari decât sistemele monoprocesor ca să mai micşoreze din prăpastia de comunicare între microprocesor şi memorie (DRAM). Cercetarea în acest domeniu este extrem de necesară având în vedere faptul că în anul 2015 se aşteaptă microprocesoare comerciale de uz general de 256 de nuclee. Se consideră că sistemele *multicore* (eterogene) vor deveni dispozitivul universal de calcul. Se speră că aceste sisteme vor putea corela eficienţa procesării cu creşterea densităţii de integrare, care evoluează conform legii lui *Moore*. Compania Intel a fabricat deja un *chip* cu 80 de *core*-uri integrate, v. http://techfreep.com/intel-80-cores-by-2011.htm, pe care îl va lansa în producţia de masă în curând. Procesoarele grafice Nvidia Tesla C1060, cu 240 de nuclee integrate în *chip*, oferă

performanţe de până la un Teraflop/s. Procesorul *multicore* Sony/Toshiba/IBM Cell, cu 9 nuclee neomogene integrate per *chip*, atinge rate de procesare de 200 Gflop/s. Programarea şi utilizarea eficientă în asemenea cazuri, nu vor fi posibile până când nu vor avea loc schimbări radicale în modelele de programare şi în instrumentele *software* disponibile.

Metricile succesului în cadrul dezvoltării sistemelor *multicore* se referă în principal la productivitatea programării şi la performanţa aplicaţiei. Performanţa trebuie să aibă în vedere minimizarea acceselor la datele aflate în afara memoriilor locale (prin managementul "localităţii" datelor), optimizarea balansării încărcării procesoarelor şi respectiv optimizarea comunicaţiilor şi sincronizărilor. În acest scop, se impune o nouă proiectare a algoritmilor, în vederea mapării lor optimale pe sistemele de tip *multicore*.

Arhitecturi *multicore* omogene vs. eterogene

Nu putem fi de acord în totalitate cu opinia specialiştilor de la *Berkeley* care afirmă că nucleele viitoarelor multiprocesoare vor consta în procesoare simple [Asa06, pg. 22]. În acest caz, paralelismul la nivel de instrucţiuni, (singurul) exploatabil la nivelul programelor secvenţiale, ar fi mult diminuat. Acest dezavantaj este inacceptabil având în vedere că peste 99% din programele scrise până acum au fost scrise în limbaje secvenţiale. Din acest motiv credem că viitorul, în calculul de uz general, este cel al unor sisteme *multicore* eterogene (câteva nuclee superscalare *out of order* cu execuţii speculative + multe nuclee superscalare mai simple, de tip in *order*, cu structuri *pipeline* scurte, de 5-9 stagii). Eterogenitatea permite adaptarea dinamică la diferitele caracteristici ale programelor rulate. Acest fapt este agreat de multe cercetări recente şi este justificabil în baza unui exemplu simplu preluat din [Hen07]. Se presupune, spre ex., că f=10% din timp, un program nu poate fi accelerat prin paralelizare pe un sistem cu N=100 de procesoare. Se consideră că pentru a rula această parte secvenţială de doua ori mai rapid decât pe un procesor simplu, este nevoie de un procesor complex, care are de 10 ori mai multe resurse (complexitate) decât procesorul simplu. Aplicând legea lui *Amdahl*, accelerarea obţinută pe un sistem omogen cu 100 de nuclee simple este:

$$S = \frac{1}{f + \frac{(1-f)}{N}} = 1 / (0.1 + 0.9/100) = 9.2 \qquad (1)$$

Accelerarea obţinută pe un sistem neomogen de complexitate echivalentă, având 90 de nuclee simple şi un nucleu complex, este superioară:

$$S' = 1 / (0.1/2 + 0.9/90) = 16.7 \qquad (2)$$

Totuşi, superioritatea sistemelor neomogene faţă de sistemele omogene, va trebui dovedită mult mai convingător prin simulări complexe, atât pe *benchmark*-uri secvenţiale cât şi paralelizate (IPC, energie consumată, buget tranzistori, arie integrare etc.). De altfel, chiar şi actualele multiprocesoare comerciale sunt neomogene, având un nucleu superscalar *out of order* foarte puternic (IBM Cell BE, Intel IXP – procesoare de reţea etc.) şi totuşi, la nivelul unor *manycore* – uri eterogene, modelele de programare ar putea deveni foarte complicate. Virtualizarea va juca un rol important atât în vederea portabilităţii aplicaţiilor cât şi în vederea evidenţierii concurenţelor, mai facilă la nivelul maşinii virtuale, care poate beneficia de meta-informaţii derivate din codul HLL (*High Level Language*). În schimb, în multe aplicaţii din calculul dedicat (aplicaţii grafice, aplicaţii numerice etc.) este foarte probabil ca sistemele *multicore* omogene să aibă succes.

Exploatarea sinergică a tipurilor de paralelism

Cercetările viitoare trebuie să vizeze şi exploatarea sinergică a diverselor tipuri de paralelism (de tip *pipeline* prin suprapunerea execuţiei fazelor instrucţiunilor, ILP, TLP – *Thread Level Parallelism*, ori chiar la nivelul unor *task*-uri independente) Extensiile de paralelizare ale limbajelor de programare (ex. OpenCL, Grandcentral) vor ajuta aplicaţiile să beneficieze de toate resursele disponibile (CPU, *multi-cores*, GPU).

Nici paralelismul la nivelul datelor (bit, cuvânt – arhitecturi vectoriale) nu trebuie neglijat. Spre exemplu modelul OpenMP permite exploatarea paralelismelor la nivel de *task*-uri, bucle şi date, deopotrivă. Aplicaţiile de tip web, mobile (*cloud computing*) sau bazele de date sunt caracterizate de paralelism la nivel de fire sau la nivelul acceselor la memorie şi mai puţin la nivel ILP. În asemenea cazuri, fiecare client poate fi servit de către un nucleu separat. Aceste aplicaţii necesită multe procesoare simple, cu memorii performante de mare capacitate.

Ierarhia de memorie

Se arată în literatură că ierarhia de cache-uri este departe de a fi optimal proiectată. Circa 50% din blocurile din *cache* sunt "moarte" (analog, cca. 50 % din datele dintr-o pagină sunt nefolosite). Sistemul de *cache*-uri nu mai trebuie văzut ca un modul fix, pe care soft-ul trebuie să se plieze. Acest sistem trebuie să devină mai maleabil, adaptat la cerinţele rulării dinamice a aplicaţiilor. Sunt necesare cercetări în vederea optimizării ierarhiei de memorii cache în sistemele multiprocesor, prin exploatarea agresivă a "localităţilor" (vecinătăţilor) spaţiale şi temporale. Astfel, spre exemplu, în [Wat09] autorii se focalizează pe mecanisme *hardware* care adaptează dimensiunea logică a blocului din L2 *cache* la caracteristicile aplicaţiei. Accesarea L2 *cache* se face în paralel cu accesarea unor aşa numite *cache*-uri de monitorizare (*Observation Cache* - OC), fiecare având lungimi diferite ale blocului. *Cache*-ul OC care contorizează pe un anumit interval de timp un număr minim de *miss*-uri, determină blocul optimal din punct de vedere al dimensiunii, necesar a fi adoptat în L2 *cache*. Un bloc de o asemenea dimensiune exploatează cel mai bine vecinătatea spaţială a datelor şi a instrucţiunilor. Un *controller* special de întreruperi monitorizează periodic starea *cache*-urilor OC si, în consecinţă, modifică în mod corespunzător lungimea blocului logic din L2 *cache*. Mecanismul este unul simplu, eficient şi robust, care nu implică intervenţia compilatorului ori a aplicaţiei *software*. Sunt necesare cercetări care să dezvolte noi scheme de protocoale de coerenţă a *cache*-urilor în sistemele MIMD, scalabile, flexibile, reconfigurabile chiar, care să suporte sute şi chiar mii de nuclee de procesare integrate pe un singur *chip* de uz general. Există studii empirice care arată că la o tehnologie de integrare de 30 nm se pot integra 1000-1500 de nuclee simple pe o singura pastilă de siliciu (actualmente Intel utilizează tehnologii la 45 nm). Tehnicile de coerenţă actuale de tip *snooping* (MSI, MESI, MOESI etc.) precum şi cele de tip *directory-based* sunt depăşite, fiind nescalabile la nivelul sutelor şi miilor de nuclee integrabile pe un *chip*. Totuşi, cercetări incrementale utile, bazate în principal pe predicţie şi speculaţie, ar putea perfecţiona chiar şi aceste protocoale. Tehnicile de coerenţă vor trebui să aibă în vedere noile caracteristici specifice sistemelor *multicore* (latenţele *miss*-urilor *cache to cache*, lărgimea de bandă a *bus*-urilor de interconectare etc., care sunt mult diferite decât cele aferente sistemelor multiprocesor clasice.) În consecinţă, viitorul aparţine sistemelor *multicore* cu memorie partajată distribuită (*Distributed Shared Memory* sau *Non*

Uniform Memory Architectures) cu protocoale de coerenţă *hardware-software* sau de tip *Message Passing* (NoC – *Networks on a Chip*) şi în tehnologii de integrare de tip 3 D [Mog09]. În optimizarea mecanismelor de coerenţă trebuie ţinut cont în mod deosebit de minimizarea puterii consumate.

Coerenţa şi consistenţa

Un program paralel trebuie să specifice, printre altele, ordinea operaţiilor de scriere-citire. Cel mai simplu, ar fi să se păstreze ordinea impusă de dependenţele de date din program. Când ordinea implicită a acestor operaţii nu este determinată în mod unic, sunt necesare operaţii explicite de sincronizare între fire (creare/unificare fire, excludere mutuală, alocare memorie partajată, bariere etc.). Coerenţa *cache*-urilor asigură o viziune consistentă a memoriei pentru diversele procesoare din sistem. Nu se răspunde însă la întrebarea "cât de consistentă?", adică în ce moment trebuie să vadă un procesor că o anumită variabilă a fost modificată de un altul? Cea mai simplă soluţie constă în forţarea fiecărui procesor care scrie o variabilă partajată, de a-şi întârzia această scriere până în momentul în care toate invalidările cauzate de către procesul de scriere, se vor fi terminat. Această strategie simplă se numeşte consistenţă secvenţială. Ea impune ca orice simulare a procesării unui anumit program, prin respectarea secvenţialităţii interne din cadrul unui fir (*data-flow order*) şi, respectiv, prin ordinea aleatoare de execuţie a firelor între ele, atunci când nu este specificată una anumită, să conducă la aceleaşi rezultate finale. Altfel spus, un program scris corect va duce la aceleaşi rezultate, indiferent de întreţeserea firelor. Deşi consistenţa secvenţială prezintă o paradigmă simplă, totuşi ea reduce performanţa, în special pentru sistemele cu un număr mare de procesoare sau cu reţele de interconectare de latenţe ridicate. Aşadar, este necesară dezvoltarea unor noi modele, mai relaxate decât cel al consistenţei secvenţiale (*release consistency*) de asigurare a consistenţei memoriei partajate. Unele cercetări alternative se focalizează pe anumite mecanisme de *prefetch* a datelor, implementate în cadrul mecanismului simplu al consistenţei secvenţiale, încercând astfel să reducă degradarea de performanţă pe care acest mecanism o implică prin secvenţialitatea acceselor la memoria de date.

Reţele de interconectare

Sunt necesare reţele de interconectare de lărgime de bandă ridicată, în cadrul unor sisteme cu memorie partajată *on-chip,* scalabilă. Actualmente, reţelele de interconectare (*on chip*) au lărgimi de bandă relativ modeste, fiind limitate de capacităţile parazite, datorate modulelor interconectate, întârzierilor de arbitrare etc. În plus, ele sunt nescalabile la nivelul sutelor ori miilor de nuclee integrate. O excepţie pozitivă o constituie *HyperTransport,* o reţea de interconectare de tip *point to point* (pachete), bidirecţională, serial/paralelă, scalabilă, de latenţă mică şi de lărgime de bandă ridicată, v. http://www.hypertransport.org. Ea permite interconectarea procesoarelor, a acestora cu interfeţele de I/O şi cu diferitele acceleratoare. AMD utilizează această reţea flexibilă în multe dintre *multicore*-urile sale (Athlon 64, Athlon 64 X2, Athlon 64 FX, Opteron, Sempron şi Phenom). *HyperTransport* oferă legături distincte între procesor – memorie şi respectiv între procesor - sub-sistemul de I/O. Mai mult, ea oferă legături fizice distincte pentru citiri şi scrieri din/în spaţiul de I/O, oferind deci paralelizări agresive ale acestora. La ora actuală reţeaua oferă rate de transfer de până la 10400 MB/s (versiunea a 3-a).

În domeniul reţelelor de interconectare implementate *on-chip*, cercetările vor avea în vedere, în special, arhitecturile *switch*-urilor, topologiile şi algoritmii de rutare. Tehnologia VLSI utilizată influenţează în mod direct soluţiile arhitecturale. Probabil că cercetările se vor inspira din reţelele de interconectare ale supercomputerelor actuale (spre exemplu supersistemul IBM *BlueGene* care conţine 65536 de noduri, clusterizate în cate 80 CPU 2-SMT PIM - *Processing în Memory*, conectate *Crossbar*/cluster. Aceste clustere sunt interconectate prin intermediul unei reţele tip *3 D Torus Network* care face ca fiecare nod să poată comunica direct cu alţi 6 vecini situaţi pe axele ortogonale XYZ. Aici şi subsistemul ierarhic de *cache*-uri va juca un rol extrem de important, fiind necesară îmbunătăţirea sa prin noi idei. De remarcat că eficientizarea acestui subsistem va conduce la scăderea presiunii asupra reţelelor de interconectare şi a memoriei principale partajate. Detecţia şi abortarea execuţiei instrucţiunilor *Store* care doresc să scrie o valoare deja existentă în memoria partajată, numite *Silent Stores*, ajută de asemenea în mod semnificativ la reducerea presiunii asupra reţelei de interconectare [Vin07]. Cercetări novatoare în arhitectura memoriilor DRAM sunt foarte necesare, având în vedere că un *chip* DRAM de 512 Mbit, spre exemplu, conţine sute de blocuri fizice independente, oferind deci un potenţial uriaş în creşterea lărgimii de bandă, prin accesări întreţesute. Integrarea *on-chip* a memoriilor DRAM nu mai presupune multiplexarea adreselor (necesară în memoriile *off-chip* datorită costurilor mari ale terminalelor). Acest fapt conduce la scăderea semnificativă a timpului de acces. De altfel, sistemele masiv paralele se bazează pe conceptul de PIM. În acest caz, memoria DRAM este integrată în cadru procesorului, cu mari beneficii asupra latenţei şi lărgimii de banda procesor-memorie. O reţea de procesoare PIM se numeşte arhitectură celulară. În arhitecturile celulare se pot conecta milioane de procesoare, fiecare procesor fiind conectat doar la câţiva vecini din cadrul topologiei de interconectare. Sistemele multicore au accentuat şi mai mult *gap*-ul între CPU şi sistemul secundar de memorie (discuri), conducând la scăderea lărgimii de bandă *per core*. Este deci imperios necesar un sistem de I/O mult mai rapid, bazat pe abilitatea de a mapa în mod eficient un număr mare de operaţii concurente de I/O, pe sutele de unităţi de stocare. În acest scop trebuie aduse îmbunătăţiri aplicaţiilor, maşinii virtuale, sistemului de operare şi controlerului dedicat al unităţilor de disc.

Modele de programare paralelă

Conform [HiPEAC], în următoarea decadă, în cadrul aplicaţiilor vor fi deosebit de pregnante următoarele tendinţe: accesul ubicuu, servicii personalizate şi delocalizate, sisteme masive de procesare a datelor, realitate virtuală de înaltă calitate, senzori inteligenţi. Aceste tendinţe se vor manifesta în aplicaţii concrete, precum cele legate de roboţi domestici, vehicule auto-pilotate, teleprezenţă, jocuri, implanturi şi extensii ale corpului uman (*human*++) etc. Pentru programarea acestor aplicaţii pe sistemele *multicore* şi *manycore* sunt necesare modele eficiente, performante dar şi simple (!) de programare paralelă. Cercetările în acest domeniu trebuie să investigheze metodele de evidenţiere a concurenţelor la nivelul limbajelor de programare. În general, este de dorit ca aceste modele să fie independente de numărul de procesoare din sistem. Paralelismul reprezintă rezultatul exploatării concurenţei pe o platformă paralelă. Rolul modelului de programare este acela de a exprima concurenţa, într-un mod independent de platformă. Este sarcina compilatorului şi a maşinii *hardware* să decidă cum să exploateze concurenţa, prin procesări paralele. Gradul de abstractizare al modelului de programare trebuie ales prin prisma compromisului optimal între productivitatea şi eficienţa programării paralele. Aceste modele trebuie să pună în evidenţa paralelismele inter-*thread*-uri, alocările de memorie, accesul la zonele

de date partajate şi modurile de sincronizare. Ingineria programării se focalizează în continuare pe corectitudinea şi pe reutilizarea codului, pe productivitatea dezvoltării dar nu şi pe exploatarea paralelismelor. Se estimează că în viitorul apropiat, circa 10% dintre programatori vor dezvolta programe paralele în mod explicit. Actualmente aceste modele de programare paralelă sunt nesatisfăcătoare, conducând la o programare, testare şi depanare extrem de dificile. Spre exemplu, standardul Posix este considerat ca fiind puţin flexibil şi de nivel jos. Standardul OpenMP nu exploatează "localitatea" datelor. De asemenea, modelele de programare actuale nu iau în considerare eterogenitatea arhitecturii. În plus, scalabilitatea acestor modele este una scăzută. Productivitatea actualelor paradigme de programare paralelă *(shared memory* - memorie partajată respectiv *message passing* - memorie distribuită logic) este una scăzută. Astfel, de exemplu, metodele actuale de sincronizare, bazate pe secţiuni critice atomice (excluziune mutuală prin *lock/unlock*), nu mai sunt fezabile, fiind nevoie de metode noi, mai productive.

In acest sens, conceptul de memorie tranzacţională (*Transactional Memory* - TM) pare a fi unul promiţător, deşi cercetările sunt încă într-un stadiu incipient. Tranzacţia constituie o secvenţă de cod care se execută atomic, în mod speculativ, prin mai multe citiri şi/sau scrieri la nivelul unei memorii partajate. Rularea programului nu ţine cont de secţiunile critice. Dacă apar conflicte la nivelul variabilelor partajate accesate de fire multiple, rezidente pe diferite procesoare, aceste conflicte se vor detecta şi firul violat îşi va relua execuţia tranzacţiei în mod corespunzător (*roll-backs*). Aşadar gestiunea coerenţei nu se mai face la nivelul fiecărei scrieri aferente unei variabile partajate, ci la nivelul unor pachete atomice, fiecare pachet conţinând mai multe astfel de scrieri. Tranzacţia este atomică (se execută dpdv logic în totalitate sau deloc), consistentă (dpdv al variabilelor partajate inter-tranzacţii) şi durabilă (odată începută, nu mai poate fi abortată). TM simplifică tehnicile de excluziune mutuală din programarea paralelă. Avantajul principal al conceptului de TM nu o constituie atât performanţa rulării, cât corectitudinea acesteia, chiar şi în condiţiile în care programatorul (compilatorul) efectuează în mod eronat paralelizarea aplicaţiei. Productivitatea şi facilizarea programării constituie alte obiective importante asociate acestui concept. Probabil că cercetările în domeniul TM trebuie să abordeze scheme hibride, de tip *hardware-software*. Aceste cercetări trebuie dezvoltate în paralel cu extensia şi optimizarea setului de instrucţiuni maşină (ISA – *Instruction Set Architecture*) şi a interfeţei *hardware-software*, în vederea facilizării programării paralele.

În lucrarea [Ham04], cercetători de la Stanford propun un nou model de memorie partajată, numit *Transactional memory Coherence and Consistency* (TCC), practic o memorie tranzacţională implementată în *hardware*. Aici, tranzacţiile atomice sunt întotdeauna unităţile de bază ale procesării paralele. TCC trebuie să grupeze, în *hardware*, toate scrierile dintr-o tranzacţie, într-un singur pachet. Acest pachet se trimite în mod atomic la memoria partajată, iar scrierile variabilelor partajate se efectuează la finele execuţiei tranzacţiei (*commit*). Se controlează prin *hardware roll-back*-urile tranzacţiilor procesate în mod speculativ. Aceste tranzacţii speculative necesită *roll-back* atunci când mai multe procesoare încearcă să citească şi să scrie, în mod simultan, aceeaşi dată. Protocoalele de coerenţă de tip *snoopy* se implementează la nivelul acestor tranzacţii atomice şi nu la nivelul scrierilor individuale. Ele permit detecţia faptului că tranzacţia curentă a utilizat date care au fost deja modificate de o altă tranzacţie (*dependence violation*), şi deci, este necesar *roll-back*-ul tranzacţiei în curs. În consecinţă, consistenţa secvenţială se implementează la nivelul tranzacţiilor, care se vor termina în ordinea secvenţială a programului original, şi nu la nivelul scrierilor individuale. Întreţeserea între

scrierile diferitelor procese este permisă numai la nivelul tranzacţiilor. Acest model TCC impune la nivelul programatorului să insereze explicit tranzacţiile în codul sursă, ca pe nişte regiuni paralelizabile. Aceste tranzacţii pot fi rafinate iterativ şi adaptiv, în urma diferitelor rulări ale programului. Evident că o anumita tranzacţie nu poate separa un *Load* de un *Store* succesiv, care accesează aceeaşi variabilă partajată ca şi *Load*-ul.

O altă provocare, extrem de importanta, o constituie dezvoltarea unui model de programare care să permită utilizarea transparentă şi simultană atât a modelului cu memorie partajată cât şi a celui cu memorie distribuită. Ceva încercări în acest sens se regăsesc în limbajele *Co-Array Fortran, UPC, X10, Fortress, Chapel* etc. [HiPEAC]. De asemenea, sunt necesare mecanisme novatoare care să permită compilatorului şi sistemului *run-time* să optimizeze structurile de date partajate, adaptându-le la condiţiile execuţiei. Ideea de esenţa este că o structura de date partajată să fie distribuită automat în sistem. Cercetările în domeniul arhitecturilor reconfigurabile, în special în lumea *embedded*, sunt şi ele extrem de necesare întrucât aceste arhitecturi se pot adapta în mod static sau/şi dinamic mai bine la cerinţele aplicaţiei specifice. Avantajul limbajului *Java* în arhitecturile multiprocesor constă în faptul că implementează în mod nativ conceptul de fir de execuţie. Din acest motiv, cercetările în domeniul sistemelor *multicore* care utilizează procesoare capabile să execute direct în hardware *bytecode*-urile *Java*, sunt deosebit de utile, în special în lumea sistemelor dedicate.

Este necesar suport *hardware* pentru programarea paralelă, inclusiv pentru depanarea programelor paralele prin monitorizarea execuţiilor. Depanarea unor sisteme *multi-core* cu sute de *thread*-uri procesate în limbaje native diferite, de către procesoare neomogene, este o problemă deschisă de mare actualitate şi interes. O singura sesiune a depanării trebuie să vizualizeze toate instrucţiunile maşină, informaţii legate de variabile şi funcţii, punerea în evidenta a erorilor de comunicare între module dar şi a erorilor locale etc. Evitarea violărilor de *timing* în cazul sistemelor în timp real având constrângeri tari (*Worst Case Execuţion Time*) şi a erorilor de execuţie având cauze incerte (*Heisenbugs*), constituie provocări majore. Fără ajutorul *hard*-ului, care să ofere o viziune globală a stării maşinii *multicore*, asemenea cerinţe par imposibil de îndeplinit. Proiectarea *hardware* trebuie să aibă în vedere deci şi observabilitatea rulărilor multiple, fără a genera însă prea mari cantităţi de date nerelevante.

Paralelizarea aplicaţiilor

Paralelizarea automată constituie un obiectiv maximal, vizat de vreo 40 de ani de cercetări asidue. În acest sens s-au obţinut realizări notabile, îndeosebi în paralelizarea automată a programelor ştiinţifice scrise în limbaje orientate pe vectori (*Fortran, Matlab* etc.) şi pentru arhitecturi omogene cu memorie partajată, în special din lumea *embedded*. Aceste realizări trebuie extinse la tipuri cât mai diverse de aplicaţii, la limbaje bazate pe pointeri şi la sisteme multiprocesor eterogene, cu diferite modele de memorie. În acest scop sunt necesare, în special, tehnici statice noi de analiză a programelor scrise în limbaje bazate pe pointeri. O altă direcţie insistă pe limbajele orientate pe domenii, unde punerea în evidenţă a concurenţelor este mai facilă decât în cele de uz general. Paralelizările speculative, incluzând aici tehnici de *multithreading* speculativ, vor juca un rol tot mai important. Planificarea dinamică a *thread*-urilor în vederea optimizării comunicaţiilor şi sincronizărilor este deosebit de importantă. Astfel, de exemplu, în [Gio09] autorii dezvoltă o aşa numită *Decoupled Threaded Architecture* (DTA) în

vederea exploatării paralelismelor fine şi medii la nivel TLP în cadrul nor sisteme *many-core* eterogene. Comunicaţiile şi sincronizările între fire sunt interesante, făcându-se pe un model de tip flux de date (*data-flow*) şi prin mecanisme de sincronizare de tip *non-blocking*. La crearea unui fir, acestuia i se asignează un numărător de sincronizare (SC), reprezentând numărul datelor de intrare pe care firul trebuie să-l primească de la alte fire. Acest numărător este decrementat de fiecare dată când firul primeşte o astfel de dată în memoria sa locală, destinată comunicaţiilor inter-fire. Când SC-ul firului a ajuns la zero, se va starta execuţia firului respectiv prin încărcarea datelor din memoria locală în registre. Din păcate, metodologia de simulare din această lucrare este una superficială. De asemenea, transmiterea de către compilator a unor informaţii de semantică a aplicaţiei către sistemul *multicore* pe care aceasta va rula, ar putea conduce la optimizări semnificative în procesarea aplicaţiei.

Simularea ca instrument de cercetare

Sunt necesare metode de simulare adecvate, stăpânirea complexităţii cercetării-dezvoltării sistemelor *multicore*, inclusiv prin simulare tranzacţională, *Transaction-Level Modeling* - TLM, precum permite mediul de dezvoltare pentru sisteme *multicore UniSim*, v. - www.unisim.org. Astfel, metoda TLM lucrează la un nivel de abstractizare superior celui utilizat la nivelul simulării ciclu cu ciclu. Aici, simulările se focalizează preponderent pe comunicaţiile între modulele componente. Este importantă simularea întregului sistem de calcul, inclusiv a sistemului de operare, cu toate nivelurile ierarhice funcţionale pe care acesta le deţine (*full-system simulation* – FSS, exemple: M5, *Simics*, GEMS etc.). Simulatoarele monolitice, gen *SimpleScalar*, M-*Sim*, etc., vor fi înlocuite cu simulatoare modulare, care să exploateze actualele tehnici ale ingineriei programării obiectuale, în vederea facilizării scrierii programelor şi a reutilizării codului (exemple *SystemC, Liberty, MicroLib, GEMS, SimFlex, M5, UniSim* sau *ASIM*). O idee importantă în acest sens constă în maparea facilă, intuitivă, a blocurilor *hardware* pe modulele (funcţiile, clasele) *software* ale simulatorului. Prin instanţierea acestor blocuri şi specificând conexiunile lor, se poate realiza o dezvoltare ierarhizată a sistemului *multicore*. Astfel, dezvoltarea proiectului va fi una mult mai facilă decât într-un mediu monolitic de simulare. Calitatea unui simulator de sisteme *multicore* este dată de caracteristici precum: posibilităţi de dezvoltare (modularitate), *benchmark*-uri & compilatoare, posibilităţi *full-system simulation*, tipuri de procesoare oferite, caracteristici multiprocesor oferite (UMA/NUMA, protocoale de coerenţă, reţele interconectare, modele de programare pe care le pune la dispoziţie, modele de consistenţă etc.), facilitate de calcul putere/temperatură, viteză de simulare, acurateţe de simulare, gradul de parametrizare (flexibilitatea arhitecturală). Un simulator pentru sistemele *multicore* care simulează întregul sistem de calcul este *Simics* (*Virtutech* - v. http://www.virtutech.com). Acesta oferă un set larg de procesoare (Alpha, ARM, MIPS, PowerPC, SPARC, x86-64) şi interfeţe. *Simics* poate încărca şi rula sisteme de operare precum *Linux, Solaris, Windows* XP. Poate virtualiza maşina ţintă în sisteme multiprocesor, clustere de procesoare sau reţele. Un simulator pentru sistemele *multicore*, dezvoltat în mediul academic american, este *RSim* - v. http://rsim.cs.uiuc.edu. Implementează o memorie partajată distribuită fizic (fiecare procesor având o memorie locală) şi protocoale de coerenţă de tip *directory* (MSI, MESI). Oferă procesoare superscalare puternice cu execuţii *out-of-order* şi arhitecturi complexe de memorie (CC-NUMA), inclusiv cu posibilităţi de adresare intreţesută. Comunicarea între procesoare se face prin intermediul unei reţele bidimensionale, de tip plasă. De asemenea, oferă suport pentru implementarea consistenţei secvenţiale sau chiar a unor modele mai relaxate. Un

alt simulator interesant, la nivel de ciclu maşină (emulator de arhitectură MIPS), destinat sistemelor *multicore* cu un număr configurabil de nuclee, este SESC, http://sesc.sourceforge.net. Acesta deţine un modul de calcul al *timing*-ului de mare acurateţe. M5 este un FSS-simulator dezvoltat în tehnologia programării pe obiecte (C++), permiţând deci instanţierea facilă a modulelor sistemului multiprocesor. Conţine modele pentru procesoarele Alpha, MIPS, ARM şi SPARC. Implementează ierarhii complexe de *cache* cu protocoale de coerenţă de tip *snoopy*. Încarcă sistemele de operare Linux şi Unix (Solaris). Permite 3 moduri de lucru în vederea compromisului între viteza de simulare şi acurateţea simulării. În domeniul sistemelor *multicore* dedicate aplicaţiilor multimedia, simulatorul *Sesame* dezvoltat la Universitatea din Amsterdam este unul cunoscut şi apreciat.

O altă problemă majoră constă în viteza simulărilor. La ora actuală, simularea cu acurateţe totală a unui sistem monoprocesor real, la nivel RTL – *Register Transfer Logic*, necesită în jur de o zi de simulare, pe sisteme performante! Pentru sisteme *multicore* acest timp de simulare va creşte cel puţin într-o maniera liniară. Având în vedere complexitatea enormă a simulărilor de tip *cycle by cycle*, care le face nefezabile pentru optimizarea unor sisteme cu sute de nuclee integrate, există un mare interes inclusiv pentru dezvoltarea unor metode de simulare la nivel de tranzacţii sau chiar a unor metode analitice de optimizare. Se au în vedere inclusiv metode de simulare statistică, bazate pe eşantioane reprezentative ale procesării (*sampling simulation*). Ideea de esenţa aici constă în selectarea unor subseturi ale *benchmark*-urilor având un comportament suficient de similar cu cel al întregului set de *benchmark*-uri. O bună sinteză a cercetărilor din acest domeniu se găseşte în [Yi06]. Se pune problema extragerii caracteristicilor *benchmark*-urilor în urma rulării şi construcţia unor benchmark-uri sintetice având aceleaşi caracteristici, reprezentative şi mai scurte (*statistical simulation*). O altă soluţie în vederea reducerii timpului de simulare constă în paralelizarea simulatorului şi rularea lui pe sisteme *multicore* performante precum IBM Cell BE de exemplu. O altă soluţie în vederea reducerii timpului de simulare o constituie clasificarea / clusterarea prin metode statistice a *benchmark*-urilor. În fine, o altă provocare importantă o constituie îmbunătăţirea acurateţii simulărilor prin metodologii specifice. Acurateţea relativă a simulărilor poate fi mai importantă decât acurateţea absolută a acestora, în fazele incipiente ale proiectului.

Benchmarking

Ca metodologie de cercetare-dezvoltare se impune tot mai mult proiectarea *hardware-software* integrată (*hardware-software co-design*). Tot aici se pune problema dezvoltării unor metode de *benchmarking* adecvate evaluării sistemelor de tip *multicore*. Problema este una extrem de delicată, întrucât companiile îşi protejează atent aplicaţiile comerciale. Astfel, în locul uzitatelor *benchmark*-uri (SPEC – pentru procesoare de uz general, EEMBC şi *MiBench* – pentru sisteme dedicate, *Mediabench, ALPBench* – pentru sisteme multimedia, TPC – pentru baze de date, *Livermoore, Parsec*, SPLASH 2 – pentru sisteme cu paralelism masiv etc.) s-au propus aşa numitele *dwarfs*, constând în metode algoritmice tipice calculului paralel care să conţină atât *pattern*-uri computaţionale cât şi, mai ales, *pattern*-uri de comunicaţie între nucleele componente [Asa06]. Şi totuşi, problema rămâne una deschisă, cu multe semne de întrebare (spre exemplu, *benchmarking*-ul trebuie să ţină cont şi de tendinţa accentuată a virtualizării, ca şi metodă esenţială în asigurarea compatibilităţii şi portabilităţii.)

Automatic Design Space Exploration

Se pune problema dezvoltării unor metode de căutare euristică optimizată în spaţiul enorm al parametrilor aplicaţiilor şi arhitecturii cercetate - *Automatic Design Space Exploration* (ADSE). Spre exemplu, proiectarea unui 4-*core* în care fiecare nucleu poate fi ales dintr-o bibliotecă conţinând 480 de modele, impune evaluarea a peste 2.2 miliarde de posibilităţi! Dacă evaluarea fiecărei posibilităţi ar necesita o zi, evaluarea tuturor posibilităţilor ar necesita aproape 1 milion de ani. Relativ la aplicaţii, se au în vedere transformările algoritmice şi de limbaj ce pot creşte gradul de paralelism TLP. Scopul este evident, anume determinarea acelor parametri care conduc la optimizarea raportului performanţă/cost (IPC, energie, arie de integrare etc.) Optimizările au deci obiective multiple. Evident că nu se pune problema căutării complete în spaţiul enorm al tuturor parametrilor arhitecturii sau/şi aplicaţiilor investigate. Tehnici din domeniile învăţării automate (*machine learning*), a cercetărilor operaţionale ori *data mining* ar putea fi utile, în vederea reducerii spaţiului de căutare şi deci, a reducerii numărului de simulări. Aceste metode de optimizare automată a parametrilor arhitecturii se vor integra sub forma unor API-uri în cadrul simulatoarelor dezvoltate. Tot aici, sunt necesare tehnici euristice, inspirate din domeniul *machine learning,* în vederea optimizării incrementale a compilării. Ideea este că programul să se adapteze la schimbările din *hardware* (spre ex. creşterea numărului de procesoare) dar şi la evenimentele dinamice care apar pe parcursul procesării (spre exemplu *miss*-uri în *cache*). La ora actuală, nu există un instrument ADSE matur, care să fie larg folosit în optimizarea sistemelor de calcul. Obiectivele constau în creşterea performanţei, dar şi în reducerea consumului de putere (atât dinamică cât şi statică). Evident că în acest sens sunt necesare informaţii de *feedback*, captate în urma rulărilor aplicaţiei. Este necesară deci o abordare mai strânsă între cercetările în domeniul microarhitecturilor şi cel al compilatoarelor (*compiler-architecture co-design*). Compilatorul trebuie să poată manipula infrastructura microarhitecturii, în timp ce aceasta trebuie să beneficieze de informaţiile transmise ei prin intermediul ISA, de către compilator. Din păcate, este foarte dificilă trecerea de la actualele compilatoare, orientate pe exploatarea ILP, la compilatoare noi, orientate pe exploatarea TLP şi a paralelismelor la nivelul *task*-urilor. Totodată, această abordare holistică este una mare consumatoare de timp. În scopul reducerii timpului de proiectare integrată a dualităţii arhitectură-compilator, se dezvoltă modele bazate pe învăţare automată, care pot predicţiona performanţa unui compilator optimizat pentru o anumită arhitectură, fără ca să îl construiască efectiv. Modelele folosesc ca intrare o parte infimă din spaţiul parametrilor compilator-arhitectură.

Avem aşadar nevoie de tehnici inteligente şi eficiente de căutare în spaţiul enorm de parametri ai sistemului. Adaptarea tehnicilor de căutare-optimizare din domeniul învăţării automate a fost investigată de către mulţi autori. În [Suk07], spre exemplu, autorii dezvoltă un instrument complex de accelerare a DSE pentru arhitecturi *multicore*, numit *Magellan*. Acesta determină parametrii cvasioptimali, cei care maximizează rata de procesare pentru un buget prestabilit al ariei de integrare şi puterii disipate. *Magellan* foloseşte algoritmi euristici de căutare de tip *hill climbing*, genetici, stigmergici etc. Clasificarea soluţiilor şi a *benchmark*-urilor funcţie de caracteristicile acestora, poate accelera căutarea.

Algoritmul *Steepest Ascent Hill Climbing* (SAHC) implică în acest caz, căutarea în vecinătatea celui mai bun k-procesor curent. Un procesor vecin este un procesor care diferă de cel curent, prin valoarea unui singur parametru. Următorul procesor optimal este ales ca fiind cel mai bun

vecin, dacă este superior procesorului curent. Algoritmul se opreşte în proximul punct de extrem. Avantajul principal al acestui algoritm simplu constă în rapiditatea convergenţei. Dezavantajele constau în complexitatea exponenţială cu numărul de *core*-uri (k) şi în posibilitatea eşuării în extreme locale. Evident, rafinamente ale acestui algoritm au fost propuse şi investigate (spre exemplu, algoritmi tip *annealing search* → SAHC + alegere *random*).

Algoritmul genetic (AG) utilizează printre alţi operatori şi operatorul de reproducere, care transferă în populaţia următoare vecinul cel mai bun. Se asigură astfel că algoritmul este cel puţin la fel de bun ca SAHC. Operatorul *crossover* generează noi procesoare, prin combinarea a două procesoare din populaţie. Se implementează şi mutaţia, care modifică aleator numărul de nuclee. În implementarea citată se păstrează în populaţia următoare doar cele mai bune 4 soluţii. Astfel, algoritmul scalează foarte bine cu creşterea numărului de nuclee (k), evaluând doar 4 k-procesoare într-o iteraţie. Nu există la acest moment o soluţie optimală de oprire a algoritmului.

Algoritmii stigmergici se bazează pe comportamentul furnicilor din lumea reală (*ant colony optimizations* - ACO). În căutarea hranei, furnicile marchează drumurile cu feromoni, astfel încât traseele să poată fi urmate şi de alte furnici. Feromonii se evaporează în timp, eficientizând astfel căutarea (spre exemplu se evită explorarea unor căi devenite, între timp, fără de succes.) Autorii forţează căutarea pe câte o cale diferită în fiecare iteraţie, prin pornirea căutării de la o soluţie situată în vecinătatea celei precedente. Se evită astfel eşuarea în extreme locale, explorându-se însă mai mult spaţiul soluţiilor.

Concluzia acestor cercetări a arătat că aceste tehnici euristice de explorare sunt cel puţin de 3800 de ori mai rapide decât căutarea exhaustivă, generând soluţii cu maximum 1% mai puţin performante decât aceasta, fapte remarcabile. Scalabilitatea este de asemenea asigurată.

Dacă în ştiinţele tari, mature din punct de vedere teoretic precum fizica sau biologia, posibilitatea reproducerii rezultatelor experimentale constituie o condiţie *sine qua non*, în arhitectura calculatoarelor acest lucru este de multe ori imposibil, datorită faptului că nu există o metodologie standardizată a cercetării. În cadrul cercetărilor dintr-o companie, simularea modulara ar putea constitui o soluţie, dacă toţi cercetătorii ar folosi acelaşi simulator. Din păcate, la nivelul cercetărilor academice ori la nivel global inter-companii, această abordare este, practic, imposibilă. În [Des08] se prezintă o metodologie automată de optimizare euristică a parametrilor unei microarhitecturi (ADSE), implementată sub forma unui *site web* numit *ArchExplorer.org*. Aceasta este oarecum independentă de simulatorul care implementează blocul *hardware* ce se doreşte a fi optimizat, chiar dacă, evident, îl utilizează pentru căutarea în spaţiul enorm al parametrilor aferenţi arhitecturii dar şi compilatorului. Desigur că integrarea simulatorului corespunzător unui anumit bloc *hardware* (de tip *cache, TLB,* predictoare de *branch*-uri, module DRAM, unităţi funcţionale, reţele de interconectare etc.) în *ArchExplorer.org* este mai facilă dacă acesta are deja o construcţie modulară, cu interfeţe bine definite între module (analoage API-urilor), compatibile cu cele definite în *ArchExplorer.org*. Un astfel de simulator este *UniSim*. Metodologia de explorare a spaţiului parametrilor arhitecturali utilizată în [Des08] se bazează pe algoritmi genetici. Fiecare modul are asociată o genă, iar aceasta la rândul ei, deţine mai multe sub-gene, codificând parametrii modulului. Din păcate, modelarea operatorilor genetici utilizaţi nu este clar explicitată în lucrare. În cadrul ADSE, relaţia arhitectura-compilator, deşi subtilă, este una puternică şi adesea subestimată de către cercetători.

Modificările arhitecturale pot produce modificări importante în strategiile de optimizare din compilator, dar şi reciproc. Dacă pentru un compilator neoptimizat performanţele procesorului sunt mai bune pentru setul de parametri P1 decât pentru P2 atunci este posibil ca folosind un compilator optimizat pentru P1 respectiv P2, performanţele în punctul P2 să fie mai bune decât în P1. Interesant este faptul că mediul *ArchExplorer.org* oferă o explorare continuă a spaţiului stărilor pe un web-server special dedicat.

O altă platformă web DSE este implementată prin proiectul colaborativ intitulat *cTuning.org* - http://ctuning.org/wiki/index.php/Main_Page, care oferă în mod gratuit utilizatorilor o tehnologie inteligentă în vederea optimizării ansamblului arhitectură-compilator-aplicaţie, pe baza unor metode din teoria învăţării statistice şi automate. Se poate accesa o bază de date (*Collective Optimization Database*) care conţine detalii asupra optimizărilor aplicate unor sisteme de calcul complexe. Accesarea acestei baze de date oferă utilizatorilor şi posibilitatea schimbului reciproc de experienţe, vizând optimizări interesante, pentru diferite aplicaţii şi respectiv platforme utilizate pentru rularea acestora. În principiu, oricine poate să-şi optimizeze în mod automat, cu instrumentele puse la dispoziţie, o anumită aplicaţie sau *benchmark*. Trebuie specificate arhitectura *hardware*, sistemul de operare şi compilatorul utilizate, urmând ca apoi serviciul web *cTuning.org* să încerce optimizarea parametrilor acestora, în vederea obţinerii unui timp minimal de execuţie dar şi a unei lungimi cât mai mici a codului sursă. În final, se generează automat parametrii optimali aferenţi procesorului, compilatorului (parametrii de optimizare a codului sursă), sistemului de operare şi aplicaţiei. De asemenea se determină şi câştigurile (performanţă, lungime de cod) obţinute faţă de un compilator GCC standard. Proiectul FP7 intitulat *MultiCube* se ocupă şi el de asemenea cercetări.

Arhitecturi *multicore* cu procesări anticipative

Puţine sunt cercetările care analizează tehnicile *Value Prediction* (VP) şi *Dynamic Instruction Reuse* (DIR) [Vin07], în cadrul procesării firelor concurente pe procesoare SMT (*Simultaneous Multithreading*) sau *multicore* (personal nu cunosc nici o cercetare pe problematica grefării DIR în *multicores)*. De aceea grupul nostru de cercetare doreşte să activeze în aceasta nişă, unde avem deja realizări apreciate (a se vedea http://acaps.ulbsibiu.ro/research.php). În asemenea arhitecturi, a prezice valoarea unei instrucţiuni şi apoi, a verifica predicţia, după ce valoarea a fost produsă, nu este întotdeauna suficient. Mai mult, acest proces poate chiar implica erori de consistenţă a variabilelor predicţionate. Astfel spre exemplu, în unele cazuri predicţia poate să fie corectă, dar execuţia incorectă datorită violării consistenţei unor variabile. În [Mar01] se exemplifică în mod convingător asemenea anomalii pe baza unui exemplu ce descrie un fir care inserează un element (nod) în capul (*head*) unei liste simplu înlănţuite. Problema este că, înainte de a insera noul nod în capul listei, firul îi modifică valoarea. Un alt fir, citeşte primul element al listei. Aceasta citire, bazată pe o VP corectă, poate fi însă eronată. Între predicţia valorilor şi problema consistenţei *cache*-urilor în sistemele multiprocesor există legaturi subtile, neexplorate încă în mod aprofundat.

Instrucţiunile cu latenţă ridicată reprezintă o sursă de limitare a paralelismului la nivelul instrucţiunilor. În [Gel09] am prezentat dezvoltarea un mecanism de anticipare selectivă a valorilor instrucţiunilor cu latenţă ridicată de execuţie, care include o schemă de reutilizare pentru instrucţiunile Mul şi Div, respectiv un predictor de valori pentru instrucţiunile Load

critice, adică cele cu *miss* în ierarhia de *cache*-uri. Rezultatele simulărilor efectuate, au arătat creşteri de performanţă (IPC) de 3,5% pe *benchmark*-urile SPEC 2000 întregi respectiv 23,6% pe cele flotante şi o scădere importantă a consumului relativ de energie (a EDP-ului), de 6,2% pe întregi respectiv 34,5% pe flotante. După ce am arătat utilitatea anticipării selective a instrucţiunilor cu latenţă ridicată într-o arhitectură superscalară, am analizat eficienţa acestor metode şi într-o arhitectură SMT, focalizându-ne pe aceleaşi instrucţiuni: Mul şi Div respectiv Load-uri critice. În acest context firele de execuţie conţinând Load-uri critice ori alte instrucţiuni de latenţa ridicată pot bloca resursele partajate ale procesorului şi, în consecinţă, pot bloca celelalte fire şi deci, pot reduce performanţa globală. Rezultatele au arătat îmbunătăţiri ale IPC pe toate configuraţiile SMT evaluate. Cu cât numărul de fire este mai mare, cu atât creşterea de performanţă devine însă tot mai puţin semnificativă, datorită exploatării tot mai eficiente a unităţilor de execuţie partajate de către procesorul SMT. Plastic spus, cu motorul SMT mergând în plin, sporul de performanţă aferent tehnicilor anticipative implementate adiţional devine mai mic. Cele mai bune performanţe medii, de 2,29 IPC pe benchmark-urile de numere întregi respectiv de 2,88 IPC pe cele flotante, s-au obţinut cu şase fire de execuţie.

In [Cra09] autorii se ocupă tot de problema instrucţiunilor *Load* critice în arhitecturile SMT, însă abordarea lor este una diferită de a noastră. Ideea de bază este ca la detecţia unui *Load* critic, în anumite condiţii, să se declanşeze execuţia speculativă a instrucţiunilor următoare în scopul exploatării paralelismelor la nivelul memoriei prin procese de *pre-fetch* (MLP – *Memory Level Parallelism*). Gradul de MLP este dat de numărul de accese independente la memorie. Evident că un astfel de *Load* de mare latenţă va genera automat un *checkpoint*, salvând deci starea curentă a procesorului (regiştrii logici, istoria *branch*-urilor etc.). Execuţia anticipată a instrucţiunilor care urmează unui *Load* critic, în cadrul unui anumit fir, se face numai dacă gradul de MLP estimat este unul suficient de ridicat. În plus, ele nu vor afecta starea arhitecturală (logică) a procesorului. Există implementat câte un predictor dinamic al gradului de MLP per *thread*. Acesta estimează gradul de MLP, după detectarea *run-time* a fiecărei instrucţiuni *Load* critice. În caz contrar (grad MLP redus), firul respectiv este rejectat din procesor, evitând astfel blocarea resurselor partajate ale acestuia. În cadrul procesării speculative, unele instrucţiuni independente de *Load*-ul critic pot cauza *miss*-uri în ierarhia de *cache*-uri. Latenţa acestora se suprapune peste cea a *Load*-ului critic aflat în curs de execuţie. Astfel se exploatează gradul de MLP existent în programe. Când un *Load* critic se încheie, procesorul iese din modul speculativ de execuţie, goleşte *pipe*-urile, restaurează *checkpoint*-ul şi reia procesarea normală, cu instrucţiunea următoare *Load*-ului. Aceasta execuţie normală se va face mai rapid având în vedere aducerile anticipate în *cache*-uri, efectuate pe parcursul execuţiilor speculative. Aşadar performanţa *per thread* creşte. Foarte important, aceasta execuţie speculativă a instrucţiunilor evită blocajul procesării în momentul în care un *Load* critic a atins vârful *Reorder Buffer*-ului (reducând deci *Memory Wall*-ul). Se arată că această arhitectură duce la beneficii considerabile, în special în cazul unor programe care lucrează intensiv cu memoria.

Asemenea idei merită aprofundate, îmbunătăţite şi investigate în continuare. Un studiu comparativ între aceste două abordări, care urmăresc reducerea influenţei instrucţiunilor *Load* critice într-o arhitectură SMT, ar fi unul interesant şi util. Desigur, comparaţia trebuie efectuată multicriterial (IPC, energie consumată, disipaţie termică, arie de integrare etc.) Mediul de simulare utilizat ar putea fi M-Sim întrucât permite şi calculul puterii (prin simulatorul *Wattch* pe care îl conţine) şi, totodată, l-am folosit deja în dezvoltarea arhitecturii SMT îmbunătăţite prin

anticiparea valorilor instrucţiunilor critice. Astfel, sperăm să obţinem rezultate interesante într-un timp rezonabil.

Grefarea unor asemenea tehnici anticipative precum cele de reutilizare dinamică a instrucţiunilor sau de predicţie dinamică a instrucţiunilor în cadrul arhitecturilor *multicore* şi *manycore* reprezintă o problemă de mare interes în opinia noastră. Implementarea unor scheme de VP şi DIR care să nu violeze consistenţa variabilelor partajate în sisteme SMT şi *multicore* este o problemă de mare interes. După cum se arată în literatura de specialitate, chiar *benchmark*-urile paralele (SPLASH-2, PARSEC etc.) nu conţin suficient paralelism pentru sistemele multiprocesor moderne [Liu08]. În aceste condiţii, creşterea performanţei porţiunilor secvenţiale de cod peste bariera impusă de celebra lege a lui *Gene Amdahl* (practic, bariera generată de dependenţele de date între instrucţiuni), se poate realiza prin anticiparea rezultatelor instrucţiunilor (înainte ca acestea să fie produse, încă din faza de decodificare). Aceste tehnici anticipative vor reduce semnificativ accesele la reţeaua de interconectare în vederea comunicării prin variabile partajate.

La ora actuală, influenţa predicţiei dinamice a valorilor şi, cu atât mai puţin a reutilizarii dinamice a instrucţiunilor, sunt departe de a fi înţelese în cadrul sistemelor SMT sau multiprocesor. Acest fapt este valabil chiar şi la nivel teoretic-principial, după cum vom arăta în continuare printr-un exemplu simplu. Este binecunoscuta legea lui *Amdahl*'s pentru un sistem MIMD cu N procesoare:

$$S(N) = \frac{1}{f + \frac{(1-f)}{N}} \qquad (3)$$

In [Liu08] autorii încearcă să generalizeze aceasta formulă, la un sistem paralel cu predicţia valorilor instrucţiunilor. Notând cu *p* acurateţea medie a predicţiei valorilor instrucţiunilor, autorii obţin relaţia de mai jos:

$$Svp(N) = \frac{1}{f(1-p) + \frac{fp + (1-f)}{N}} \qquad (4)$$

Aceasta relaţie, în opinia noastră, nu este corectă. Pentru N=1 (monoprocesor) → Svp(1)=1, ceea ce ar însemna că tehnica de predicţie a valorilor în sistemele monoprocesor nu ar aduce nici un avantaj. În mod evident, această afirmaţie este falsă. Aşa cum s-a arătat în [Gab98, Vin07], considerând un program pur secvenţial (f=1), câştigul de performanţă este în acest caz $\frac{1}{1-p}$.

O formula corecta pentru Svp(N) trebuie să ţină cont că:

- f(1-p) [%] din program este procesat în acelaşi timp în care ar face o maşină secvenţială (SISD) echivalentă.
- fp [%] din program s-ar procesa teoretic instantaneu din cauza predicţiei perfecte a valorilor instrucţiunilor.
- (1-f) [%] din program s-ar procesa teoretic de N ori mai rapid decât pe maşina secvenţială.

In aceste condiţii formula corectă este:

$$Svp(N) = \frac{1}{f(1-p) + \frac{(1-f)}{N}} \qquad (5)$$

Pentru N=1 rezultă:

$$Svp(1) = \frac{1}{1-fp} \geq 1. \qquad (6)$$

Particularizând f=1 în formula precedentă, în perfect acord cu [Gab98], rezultă:

$$S = \frac{1}{1-p} \qquad (7)$$

Implementarea unor mecanisme DIR în arhitecturile *multicore* impune ca invalidările din cadrul *buffer*-ului de reutilizare (*Reuse Buffer* – RB) aferent procesorului considerat, să fie efectuate în mod global, deci inclusiv de către instrucţiuni *Store* executate de către un alt procesor, care scrie la o adresă existentă în RB-ul procesorului considerat. Mecanismele globale de asigurare a coerenţei *cache*-urilor bazate pe invalidări la scrieri, ar putea ajuta la menţinerea consistenţei datelor din *buffer*-ul de reordonare. Invalidarea unei date din *cache*-ul de date ar trebui să determine automat invalidarea datei din RB, dacă aceasta este stocată şi acolo.

Paradigma de procesare pe baza fluxurilor de date (*Data-Flow*, DF) găseşte adepţi şi în domeniul multiprocesoarelor. Modelul DF, dezbătut puternic în anii ’80, reprezintă un model formal distribuit, inerent paralel, funcţional şi asincron. Execuţia într-un nod startează de îndată ce toate datele de intrare sunt disponibile. Programul se reprezintă sub forma unui graf în care nodurile reprezintă operaţii iar arcele reprezintă sursele (căile) de date. Modelul *Data Driven Multithreading* (DDM), relativ nou, exploatează modelul DF la nivelul firelor de execuţie, în cadrul unor arhitecturi *multicore*. Modelul DDM se bazează pe conceptul paralelizării automate la nivelul *hardware – software*, incluzând tehnici de reprezentare poliedrală a programelor. Actualmente se cercetează ansambluri *multicore* tip DDM – compilator, atât la nivel de virtualizare cât şi la nivelul unor implementări FPGA.

Putere consumată, disipaţie termică

Un obiectiv important constă şi în grefarea unor module care să calculeze automat puterea disipată şi disipaţia termică pe procesorul nostru SMT, îmbunătăţit cu predicţia valorilor instrucţiunilor Load critice (cu *miss* în L2 *cache*) respectiv cu reutilizarea instrucţiunilor de latenţă ridicată (Mul/Div etc.). Astfel, pentru calculul puterii statice şi dinamice, am putea utiliza în cadrul simulatorului SMT îmbunătăţit de noi, simulatorul *Wattch* (http://www.eecs.harvard.edu/~dbrooks/wattch-form.html) care este deja integrat în mediul M-Sim utilizat. Acesta se bazează pe simulatorul *SimpleScalar sim-outorder*, ver. 3.0 şi calculează puterea disipată pentru structuri de tip vector, structuri asociative de tip *Reorder Buffer, Load Store Queue* etc., logica combinaţională (pt. partea de comenzi ale procesorului şi unităţi funcţionale) etc. Astfel vom extinde calculul puterii disipate la întregul procesor (până în acest moment am calculat puterea doar la nivelul unor structuri regulate, de tip *cache*). De asemenea, ne va interesa şi calculul puterii statice, în special în cadrul unor memorii de capacităţi mari precum L2 *caches*. În particular, unii autori au integrat simulatorul *Simics* cu *Wattch*-ul şi au implementat un simulator performant numit *SimWattch* (simulează inclusiv procesarea sistemului de operare + putere disipată).

Un modul pentru investigarea efectelor de disipaţie termică aferente acestor tehnici, implementat ca un API al simulatorului M-Sim, ar fi de mare utilitate. Se ştie că *hotspot*-urile conduc la erori de procesare iar temperatura prea mare micşorează durata de viaţă a circuitului. Este interesant de evaluat efectul acesteia în contextul noilor structuri anticipative pe care noi le-am introdus în arhitectura SMT M-Sim. În acest sens deja am startat un studiu, în cooperare cu Politehnica din Milano, asupra problemelor, metodologiilor şi instrumentelor utilizate în cercetarea efectelor disipaţiei termice şi de energie în cadrul microprocesoarelor. Cele două procese nu sunt corelate. Apoi, va trebui să interfaţăm pe simulatoarele deja dezvoltate de noi, un *software* specializat în analiza disipaţiei termice (de exemplu simulatorul *HotSpot* - v. http://lava.cs.virginia.edu/hotspot, larg utilizat şi care nu necesită costuri de achiziţie). În continuare, vom analiza aceste efecte în cadrul arhitecturilor deja dezvoltate de noi, pe diverse *benchmark*-uri (în special în cadrul arhitecturii SMT îmbunătăţite cu DIR şi VP selective). Analiza comparativă a hărţilor termale, cu şi fără îmbunătăţirile arhitecturale propuse de noi, va fi deosebit de utilă. Pe baza puterilor dinamice calculate pentru fiecare modul într-un interval (ex. 10 kcicli), a frecvenţei de tact, a tehnologiei de integrare şi a geometriei de plasare a modulelor pe pastila de siliciu (*floorplan,* matrice de adiacentă), *HotSpot*-ul va calcula densitatea de temperatură pentru fiecare modul. Unul dintre obiectivele acestei cercetări constă în proiectarea unui *run-time thermal manager* adaptiv care să valideze-invalideze modulele de DIR şi VP în funcţie de comportamentul termic şi energetic al modulelor microprocesorului. În acest scop există deja elaborate mai multe strategii: oprirea tactului, adaptarea tensiunii de alimentare sau a frecvenţei tactului funcţie de temperatură, mutarea calculului în module de rezervă etc. Ideea de bază este aceea de a reduce densitatea de putere în punctele fierbinţi. Aceste cercetări vor fi apoi extinse pe noile simulatoare *multicore* pe care le vom dezvolta.

Concluzii

Am realizat un studiu asupra impactului arhitecturilor *multicore* şi *manycore* în cadrul ingineriei calculatoarelor. S-a pornit de la geneza acestor sisteme, determinată de limitele paradigmei monoprocesor. S-au identificat şi s-au analizat în mod sistematic, pe baza unei literaturi de specialitate recente, următoarele provocări importante pentru cercetarea şi dezvoltarea acestor sisteme:

- Arhitecturi *multicore* omogene vs. eterogene
- Exploatarea sinergică a tipurilor de paralelism
- Ierarhia de memorii *cache*
- Coerenţa şi consistenţa variabilelor partajate
- Reţele de interconectare
- Modele de programare paralelă
- Paralelizarea aplicaţiilor
- Simularea ca instrument de cercetare
- *Benchmarking*
- Explorarea automată a spaţiului parametrilor
- Arhitecturi *multicore* cu procesări anticipative – o idee originală
- Putere consumată, disipaţie termică

Concluzia de bază este că sunt necesare progrese importante în toate aceste domenii, pentru ca proiectarea şi utilizarea sistemelor *multicore* şi *manycore* să fie adecvate. Cu sau fără voia noastră, aceste sisteme vor constitui dispozitivele de calcul universale. Abordarea oricăreia dintre aceste provocări trebuie să fie una de tip holistic. S-a arătat cum vor schimba aceste noi arhitecturi, paradigma ştiinţei ingineriei calculatoarelor. În particular, programarea acestor sisteme constituie o provocare majoră, pentru care nu suntem încă pregătiţi suficient.

Bibliografie selectivă

1. [Asa06] Asanovic K. et al., *The Landscape of Parallel Computing Research: A View from Berkeley*, Technical Report No. UCB/EECS-2006-183, December 2006
2. [Cra09] Craeynest K. Van, Eyerman S., Eeckhout L., *MLP-Aware Runahead Threads în a Simultaneous Multithreading Processor*, Proceedings of The 4th HiPEAC International Conference, pp. 110-124, Paphos, Cyprus, January 2009
3. [Cul99] Culler D., Singh J., with Gupta A., *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers, 1999
4. [Des08] Desmet V., Girbal S., Temam O., *ArchExplorer.org: Joint Compiler/Hardware Exploration for Fair Comparison of Architectures*, The 6-th HiPEAC Industrial Workshop, Thales, Paris, Nov. 26th, 2008
5. [Gab98] Gabbay F., Mendelsohn A., *Using Value Prediction to Increase the Power of Speculative Execuţion Hardware*, ACM Transactions on Computer Systems, vol. 16, nr. 3, 1998
6. [Gel09] A. Gellert, A. Florea, L. Vintan, *Exploiting Selective Instruction Reuse and Value Prediction în a Superscalar Architecture*, Journal of Systems Architecture, vol. 55, issue 3, Elsevier, 2009
7. [Gio09] Giorgi R., Popovic Z., Puzovic N., *Implementing fine/medium grained TLP support în a many-core architecture*, International Symposium on Systems, Architectures, Modeling and Simulation, Samos, Greece, July 20-23, 2009
8. [Ham04] Hammond L. et al., *Transactional Memory Coherence and Consistency*, The 31st Annual International Symposium on Computer Architecture (ISCA), Munich, 2004
9. [Hen07] Hennessy J., Patterson D., *Computer Architecture. A Quantitative Approach*, 4th Edition, Morgan Kaufmann Publishers, 2007
10. [HiPEAC] High-Performance Embedded Architecture and Compilation (HiPEAC) FP6/FP7 Network of Excellence, *Research Challenges în High-Performance Embedded Architecture and Compilation*, http://www.HiPEAC.net, 2008
11. [Jor03] Jordan H., Alaghband G., *Fundamentals of Parallel Processing*, Pearson Education, Inc., Prentice Hall, 2003
12. [Liu08] Liu S., Gaudiot J. L., *The Potential of Fine-Grained Value Prediction în Enhancing the Performance of Modern Parallel Machines*, The 13th IEEE Aşia-Pacific Computer Systems Architecture Conference (ACSAC 2008), Taiwan, August 2008
13. [Mar01] Martin M., et al*., Correctly Implementing Value Prediction în Microprocessors that Support Multithreading or Multiprocessing,* Proceedings of the 34-th Annual ACM/IEEE International Symposium on Microarchitecture, Austin, Texas, December 3-5, 2001

14. [Mog09] Moga A., Dubois M., *A comparative evaluation of hybrid distributed shared-memory systems*, Journal of Systems Architecture, vol.55, issue 1, pp. 43-52, Elsevier, 2009
15. [Suk07] Sukhun Kang and Rakesh Kumar, *Magellan: A Framework for Fast Muti-core Design Space Exploration and Optimization Using Search and Machine Learning*, Illinois at Urbana-Champaign, CRHC Technical Report CRHC-07-05, October 2007
16. [Vin07] Vinţan L., *Prediction Techniques în Advanced Computing Architectures* (in limba engleza), Matrix Rom Publishing House, Bucharest, 2007; http://www.matrixrom.ro/romanian/editura/domenii/informatica.php?id=867#867
17. [Vin08] Vinţan L., Florea A., Gellert A., *Random Degrees of Unbiased Branches,* Proceedings of The Romanian Academy, Series A: Mathematics, Physics, Technical Sciences, Information Science, Volume 9, Number 3, Bucharest, 2008 - http://www.academiaromana.ro/sectii2002/proceedings/doc2008-3/13-Vintan.pdf
18. [Wat09] Watkins M., McKee S., Schaelicke L., *Revisiting Cache Block Superloading*, Proceedings of The 4th HiPEAC International Conference, Paphos, Cyprus, January 2009
19. [Yi06] Yi J., Lilja D., *Simulation of Computer Architectures: Simulators, Benchmarks, Methodologies, and Recommendations*, IEEE Transactions on Computers, vol. 55, No. 3, 2006