Universitatea *"Politehnica"* Timişoara
Facultatea de Automatică şi Calculatoare
Departamentul de Calculatoare

CONTRIBUŢII LA PROIECTAREA HARDWARE/SOFTWARE A SISTEMELOR NUMERICE MULTIPROCESOR HETEROGENE

Teză de doctorat

Conducător ştiinţific:
Prof.dr.ing. CRIŞAN STRUGARU

Doctorand:
ing. ALEXA DOBOLI

1997

CUPRINS

Capitolul 1

Introducere în proiectarea hardware/software

1.1 Proiectarea hardware/software o soluție pentru proiectarea de sistem

Necesitățile curente din economie, dar și din viața casnică, impun construirea de sisteme numerice cu o funcționalitate din ce în ce mai complexă și care trebuie să corespundă unor constrângeri de funcționare tot mai precise și mai restrictive. Proiectantul sistemelor numerice are la dispoziție o paletă largă de tehnologii de implementare - mergând de la tehnologii software la cele hardware și de aceea procesul de proiectare - în care el decide modul de realizare al sistemului, este departe de a fi unul trivial. În mod obișnuit soluția este o combinație de hardware și software, astfel încât să asigure cerințele impuse de viteză, flexibilitate, putere consumată, dar să fie și cât mai ieftină. Funcționalitatea complicată, pe de-o parte, și variantele numeroase de implementare, pe de-altă parte, fac imposibilă parcurgerea manuală a întregului ciclu de proiectare și de aceea sunt necesare instrumente software pentru automatizarea ciclului de proiectare sau cel puțin a unor pași de proiectare.

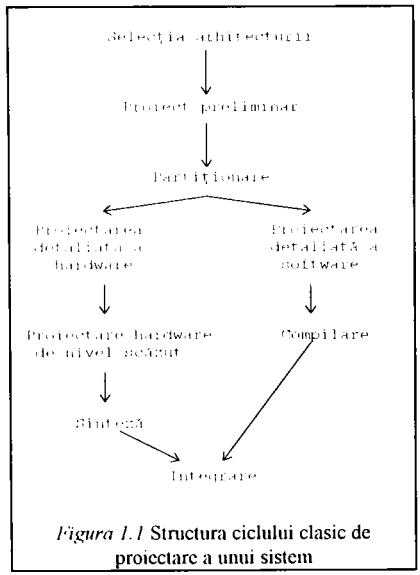


Figura 1.1 Structura ciclului clasic de proiectare a unui sistem

O metodă de proiectare tradiţională a sistemelor hardware/software [LM88] este indicată în figura 1.1 Foarte devreme în ciclul de proiectare este decisă arhitectura sistemului, după care este realizat un proiect preliminar şi decisă maniera de partiţionare a sistemului în componentele hardware şi software. După partiţionarea hardware/software implementarea software-ului şi sinteza de hardware au loc complet independent, iar în final ele sunt integrate într-o etapă dedicată. Examinând ciclul de proiectare se poate observa că foarte devreme - când puţină informaţie este disponibilă, sunt luate deciziile de proiectare cele mai importante, sinteza arhitecturii şi partiţionarea, şi care în final hotăresc calitatea implementării. Suplimentar, deciziile vizând sinteza hardware-ului nu ţin cont de software-ul pe care-l execută şi reciproc, ceea ce este un alt motiv pentru o calitate scăzută a implenetării finale. *Proiectarea hardware software* (*hardware software codesign*) îşi propune să evite aceste două "capcane" din ciclul de proiectare.

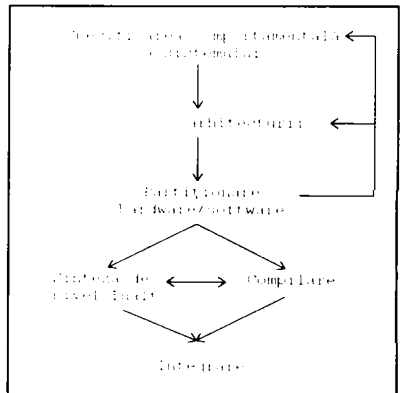


Figura 1.2 O structură posibilă pentru ciclul de proiectare hardware/software

Proiectarea hardware/software integrează metodele de proiectare hardware şi software, având obiectivul ca pe parcursul ciclului său, componentele hardware şi software ale sistemului să fie abordate ca un tot unitar folosindu-se o metodologie unică. Studiul *împreună în permanenţă* a părţilor hardware şi software este caracteristica principală a acestei metode de proiectare. Figura 1.2 prezintă un ciclu posibil de proiectare hardware/software. Proiectarea are natură *iterativă*, în primul pas fiind realizată specificarea sistemului, care poate fi rafinată pe baza informaţiei de implementare obţinută pe parcursul proiectării. Deciderea arhitecturii şi a felului de partiţionare are loc mai târziu de cât pentru metodele clasice, ceea ce permite luarea unor decizii superioare. Deasemenea, paşii pentru implementarea hardware-ului şi a software-ului sunt parcurşi astfel încât fiecare ţine cont de caracteristicile componetei complementare. Proiectarea poate continua folosindu-se informaţia descoperită în iteraţiile precedente, până când calitatea implementării este cea impusă prin specificarea sistemului. Ciclul de proiectare hardware/software trebuie să fie în întregime sau în cea mai mare parte automatizat, astfel încât să poată fi analizate cât mai multe soluţii distincte de implementare.

În continuarea capitolului 1 prezentăm principalele avantaje ale proiectării unui sistem prin metoda hardware/software comparat cu abordările tradiționale. Principalele domenii beneficiare în urma metodei sunt enumerate în paragraful următor. Paragraful 1.4 detaliază activitățile de pe parcursul ciclului de proiectare hardware/software și concepția noastră cu privire la structurarea ciclului de proiectare. Sunt prezentate nouă din cercetările cele mai reprezentative pentru abordarea noastră, iar paragraful 1.6 definește obiectivele urmărite pe parcursul acestei teze. Paragraful 1.7 încheie capitolul introductiv printr-o scurtă prezentare a structurii tezei și a problematicii studiate în fiecare capitol.

1.2 Avanatajele pe care le oferă proiectarea hardware/software

Abordarea unitară hardware/software a proiectării sistemelor reunește avantajele a două metode "complementare", proiectarea hardware și cea software. Informația accesibilă unuia este folosită pentru a crește calitatea deciziilor celuilalt. Pericolul de a fi "sufocați" de volumul mare de informație este evitat prin folosirea tehnicilor automate de parcurgere a ciclului de proiectare. Paragraful curent enumeră principalele argumente în favoarea proiectării automate hardware/software în comparație cu metodele de proiectare tradiționale.

Proiectare flexibilă și refolosibilă

Specificarea unui sistem trebuie foarte des modificată pe parcursul ciclului de proiectare și implementare. Acest modificări se datoresc fie informației dobândite pe parcursul muncii de proiectare sau a schimbărilor ce apar în cerințele funcționale ale sistemului dorit. Ideal este să operăm modificările necesare, fără a renunța la munca parcursă până în acel punct.

Produsele sunt de regulă grupate în familii, din cauză că apar noi soluții tehnologice pe care producătorul dorește să le folosească sau pur și simplu pentru că vrea să ofere unui alt segment de clienți un alt compromis cost-performanță. Versiunile diferite sunt înrudite, iar diferențele între ele apar din cauză că anumite funcții au fost modificate sau au fost implementate în mod diferit.

Cele două situații motivează suficient necesitatea de a refolosi proiecte existente. Proiectarea hardware/software crește gradul de flexibilitate și refolosibilitate a proiectării prin imaginea unitară a sistemului, pe care o menține pe parcursul tuturor pașilor de proiectare.

Explorarea eficientă a spatiului de soluții

O soluție de implementare este un compomis între performanță și cost. Pentru că folosește tehnici automate, proiectarea hardware/software are posibilitatea de a analiza un număr mai mare de soluții și de a găsi un compromis preț-performanță superior celui obținut prin metode "ad-hoc". Stabilind soluții mai bune de alocare, legare, planificare și soluția finală găsită este superioară ca performanță și cost.

Proiectarea unui sistem este o activitate complexă, ce nu poate fi gestionată ca un tot unitar. Ea este descompusă în subsisteme, acestea sunt rezolvate, iar soluțiile lor integrate în implementarea finală. Din cauza complexității problemei, dar și din lipsă de informație, metodele manuale de proiectare studiază fiecare subproblemă ruptă din contextul global. Acest lucru elimină nejustificat un număr mare de alternative de implementare și deasemenea reduce capacitatea proiectantului de a hotărî decizii eficiente de proiectare. Proiectarea hardware/software încearcă să mențină o vedere de ansamblu cît mai detaliată și cuprinzătoare. Deși subproblemele sunt abordate separat, orice decizie luată pentru o subproblemă ține cont de caracteristicile celorlalte și reciproc, decizia luată pentru o subproblemă este reflectată și spre celelalte.

Găsirea soluțiilor cu costuri reduse

Filosofia proiectării hardware/software integrează în permanență informații ce provin din arii (subprobleme) diferite. De exemplu, într-un mod foarte simplist putem afirma că conceperea părții de hardware ține cont de software și invers. Folosirea informațiilor ce aparțin unei arii în luarea deciziilor de proiectare dintr-o altă arie, are darul de a evita optime locale și de a găsi soluții optime după criterii globale. Aceste soluții economisesc resurse și au un cost mai redus. Metoda de explorare a spațiului soluțiilor este una sistematică și eficientă, într-un timp acceptabil din punct de vedere practic. De aceea, este mult mai probabil că proiectarea automată hardware/software va găsi soluții superioare celor obținute prin metodele tradiționale. Produsele care sunt astfel obținute sunt mai ieftine și calitativ superioare.

Prototipizare

Prototipul unui sistemului are rolul de a prezice performanțele sistemului, înainte ca acesta să fie efectiv produs. Prototipul este similar cu sistemul real în acele aspecte, care sunt determinante din punct de vedere al performanțelor dorite. De exemplu, prototipul poate fi simularea funcționării hardware-ului cuplat cu componentele software. Tot un prototip este și sistemul în care hardware-ul este implementat în FPGA, software-ul este executat de un procesor programabil, iar cele două componente comunică prin interfețele necesare. Proiectarea hardware/software ușurează dezvoltarea prototipurilor din cauza abordării de ansamblu a sistemului și a capacității sale de a realiza în timp foarte scurt versiuni diferite ale aceluiași sistem.

Reducerea fazei de integrare și testare

Pe parcursul întregului ciclu de proiectare este păstrată o reprezentare unificată a sistemului, ceea ce elimină riscul de a avea incompatibilități la faza de integrare sau testare a subsistemelor. Etapa de integrare și testare a întregului sistem este mult mai scurtă în timp de cât cea pentru metodele clasice și deasemenea este eliminat riscul de a opera la acest nivel modificări majore în structura implementării.

Scăderea timpului necesar ca produsul să ajungă pe piață

Un criteriu economic fundamental este acela ca un produs să fie introdus pe piață cât mai repede posibil. Acest lucru are două avantaje, primul este că concurența pe piață este mică și al doilea că clienții care au nevoie de produs, sunt dispuși să plătească

preţuri mult mai mari. Proiectarea hardware/software uşurează refolosirea proiectelor deja dezvoltate, facilitează explorarea eficientă şi rapidă a spaţiului soluţiilor, ajută prototipizarea şi simplifică faza de integrare şi testare a întregului sistem. Consecinţa acestor avantaje este că timpul de proiectare şi de aducere pe piaţă a produselor este mult diminuat.

1.3 Domenii tehnice unde proiectarea hardware/software poate fi *solutia*

Intenţia noastră nu este să creăm impresia că proiectarea hardware/software este "piatra filosofală" a proiectării sistemelor numerice şi nici măcar că ar fi metoda aplicabilă în orice situaţie practică. Am observat că activităţile de cercetare în domeniu din ultimii ani se referă foarte frecvent la exemple cum sunt controlerul braţului de robot, transformata Fourier rapidă şi mai nou, circuitul pentru protocolul ATM. Toate aceste exemple practice au trăsătura comună că sunt formate din componente hardware şi software, ce interacţionează strâns şi care trebuie să îndeplinească constrângeri de proiectare precise. În continuare, enumerăm patru domenii tehnice, care pot fi considerate beneficiare ale proiectării hardware/software, totuşi, fără ca ele să fie şi singurele.

Telecomunicatii

Telecomunicaţiile este unul dintre cele mai interesante domenii de aplicare ale proiectării hardware/software, atât prin natura tehnică a problemei, dar şi din cauza aspectelor economice, piaţa produselor pentru telecomunicaţii fiind una dintre cele mai dinamice. Produsele pentru telecomunicaţii trebuie de regulă să satisfacă cerinţe de funcţionare foarte pretenţioase, la un preţ de cost minim, în condiţiile în care concurenţa pe piaţă este ridicată. Astfel, timpul de lansare pe piaţă a produsului trebuie să fie cât mai scurt. Produsele fac parte din familii, care sunt modernizate în versiuni din ce în ce mai performante şi mai ieftine. În consecinţă, este crucial ca anumite proiecte anterioare să fie refolosite într-o proporţie cât mai mare, ceea ce implică o metodă de proiectare cât mai flexibilă. Preţul de cost mic al produsului înseamnă folosirea unor componente programabile, comune şi ieftine, de exemplu microprocesoare sau FPGA-uri, iar satisfacerea unor constrângeri de timp foarte stânse necesită folosirea ASIC-urilor.

Procesarea semnalelor digitale

Sistemele pentru procesarea semnalelor trebuie să realizeze operaţii intens computaţionale, pentru care sunt fixate constrângeri de timp foarte severe. Astfel de sisteme sunt heterogene, hardware-ul fiind folosit pentru realizarea procesărilor constrânse temporal şi pentru comunicarea cu exteriorul, iar software-ul este destinat activităţilor de administrare. Apariţia procesoarelor pentru prelucrarea semnalelor (digital signal processors) rapide şi programabile permite ca ele să realizeze multe din procesările efectuate înainte prin ASIC-uri.

Sisteme embedded (Embedded Systems)

Sistemele embedded sunt sisteme numerice, având funcționalitatea compusă dintr-un set de funcții, destinate modurilor diferite în care pot funcționa. Sistemele embedded sunt foarte populare în viața casnică (automobile, mașini de spălat, cuptoare cu microunde, roboți pentru telefon, etc.), sunt produse într-un număr foarte mare și trebuie să fie mai ales ieftine și mici. Uneori, sistemele embedded trebuie să satisfacă cerințe de funcționare în timp real.

Sistemele pentru automatizare

Sistemele numerice sunt foarte des folosite în controlul proceselor industriale. Modelarea proceselor industriale este o activitate dificilă și pretențioasă, iar deobicei specialiștii cunosc doar modele parțiale ale sistemelor reale. Procesul industrial este perceput ca un mediu exterior, ce produce stimuli de intrare și cărora li se răspunde prin semnale de control corespunzătoare. Sistemele pentru automatizare sunt formate din hardware pentru interfațarea cu sistemul industrial de controlat și software pentru funcțiile de conducere. Ele trebuie să fie în primul rând fiabile, uneori trebuie să aibă proprietăți de funcționare în timp real pentru a putea lua decizii rapide la evenimentele de intrare apărute și să fie flexibile, astfel încât să fie adaptabile la modificări ale mediului controlat.

1.4 Activitățile ce intră în componența proiectării hardware/software. Ciclul de proiectare hardware/software

Ciclul de proiectare hardware/software al unei aplicații particulare este o instanțiere a metodologiei generale *specifică-explorează-rafinează* (*specify-explore-refine*). În primul pas este construită specificarea funcțională a aplicației, apoi sunt explorate un număr cât mai mare de implementări posibile ale specificării și este aleasă acea soluție care satisface cel mai bine criteriile de proiectare dorite și o iterație a ciclului se încheie cu rafinarea specificării, prin descrierea deciziilor de implementare luate la pasul precedent. O iterație a ciclului de proiectare adaugă informații de implementare la specificare (arhitectură, legarea funcionalității, planificare, etc.) și coboară nivelul de abstractizare a specificării, apropiind-o de implementarea finală.

În continuare, facem o scurtă prezentare a activităților ce intră în structura unui ciclu de proiectare hardware/software.

Descrierea funcționalității

Scopul acestei etape este *construirea modelului* pentru sistem, *codificarea* lui folosind un limbaj de specificare de sistem și *validarea* lui prin simulare și verificare. Specificarea rezultată descrie funcționalitatea sistemului și nu cuprinde nici un fel de detalii de implementare. *Crearea modelului* este activitatea de învățare, înțelegere, organizare și definire a funcționalității sistemului [GV95]. Din punct de vedere al procesului de proiectare, construirea modelului este etapa cea mai importantă, de care

depinde fundamental calitatea implementării finale. Modelul este format din subsisteme, între care există relații pentru precizarea ordinii lor de execuție și a transferurilor de date între ele. Tipurile de modele cele mai populare sunt grafurile cu dependențe de date, garfurile cu dependențe de date și de control, rețelele Petri, modelele orientate pe obiecte, mașini cu număr finit de stări, CSP, etc. Alegerea tipului de model pentru descrierea sistemului depinde esențial de caracteristicile acestuia. Modelul stabilit este *codificat* folosind un limbaj de specificare de sistem. Limbajele cele mai folosite sunt VHDL, Verilog, HardwareC, SpecCharts, StateCharts, C++, etc., dar continuă cercetările prin care sunt propuse limbaje noi de specificare de sistem. Principala cerință pe care un astfel de limbaj trebuie să o aibă este ca el să fie simulabil, astfel încât erorile funcționale ale specificării să poată fi descoperite în fazele timpurii ale proiectării. Dacă proiectantul are la dispoziție mai multe limbaje, alegerea finală trebuie să ușureze munca de codificare și este preferat limbajul a cărui construcții au cea mai apropiată corespondență cu elementele modelului.

Explorarea spatiului de solutii

Ciclul de proiectare explorează un număr cât mai mare de soluții de implementare pentru a o identifica pe aceea ce corespunde cel mai bine intențiilor noastre (cerințe de cost, viteză de execuție, dimensiune, putere consumată). Specificarea de intrare a sistemului este transformată într-o reprezentare intermediară, ce concentrează elementele relevante pentru luarea următoarelor decizii de proiectare și ascunde detaliile neimportante. Deciziile de proiectare realizează *alocarea resurselor fizice* și *maparea funcționalității* sistemului pe aceste resurse. Cele două etape se desfășoară într-un proces iterativ și sunt ghidate de mărimi ce *estimează* calitatea deciziilor luate.

Etapa de alocare stabilește numărul și tipul resurselor ce intră în componența arhitecturii finale. Resursele alocate sunt microprocesoare, microcontrolere, ASIP-uri, ASIC-uri, procesoare paralele, FPGA-uri, circuite de memorie, controlere DMA, etc. În general alegerea trebuie făcută între resurse scumpe și cu performanțe ridicate și resurse ieftine și cu performanțe mai slabe. Cercetările în problema alocării discută trei situații: cazul arhitecturilor generale - în care uneltele software de proiectare găsesc soluțiile arhitecturale menite să satisfacă constrângerile de proiectare, cazul arhitecturilor particulare - când arhitectura considerată este una particulară, deobicei formată dintr-un procesor, un ASIC, o memorie și o magistrală sau un procesor și un ASIC și cazul în care proiectantul propune soluțiile de arhitectură, iar instrumentele software de estimare îi oferă feedback rapid despre performanțele și costurile obținute.

Funcționalitatea sistemului este asignată resurselor alocate ale arhitecturii în *etapa de partiționare*. Un caz special de partiționare este *partiționarea hardware/software*, care decide ce părți ale funcționalității sunt mutate în hardware și care în software. Elementele ce sunt subiectul partiționării sunt de trei tipuri: variabilele, care trebuie mapate elementelor de memorie, comportamentul ce trebuie atribuit la procesoare programabile sau ASIC-uri și canalele de comunicare între comportamente și care sunt legate de magistrale. Proiectarea unui sistem pentru partiționarea trebuie să discute următoarele aspecte:

- *Granularitatea de lucru* - stabilește obiectele de lucru ale procesului de proiectare. Aceste obiecte pot fi într-o gamă variată, de la o granularitate ridicată (procese, subrutine, cicluri, blocuri de instrucțiuni) și până la granularitatea fină (instrucțiuni,

operații). Alegerea granularității potrivite proiectării este în ultimă instanță, stabilirea unui compromis între timpul de proiectare și calitatea ce se poate obține.

- *Metrica ce definește o partiție bună* - metricile cele mai populare sunt viteza de execuție, costul, consumul de putere, aria de siliciu, caracteristicile de testabilitate, dimensiunea programelor, etc.
- *Modelul de estimare al metricilor* - introduce tehnicile (algoritmii) de evaluare ale metricilor unei partiții.
- *Definirea calității partițiilor* - combinarea metricilor de diferite naturi într-un parametru unic, folosit pentru compararea calitativă a două soluții. De regulă, metricile ponderate sunt integrate într-o funcție de cost, astfel că partiției optime îi corespunde valoarea minimă sau maximă a funcției de cost.
- *Alegerea algoritmului de partiționare* - algoritmul trebuie să parcurgă într-un timp scurt spațiul soluțiilor și să găsească o soluție cât mai bună din punctul de vedere al obiectivelor de proiectare dorite. Algoritmii cei mai populari pentru partiționarea hardware/software sunt algoritmi de clustering, algoritmi genetici, cu îmbunătățirea iterativă a soluțiilor (iterative improvement algorithms) sau algoritmi dedicați.

Criteriile avute în vedere în etapa de modelare a sistemului nu sunt neapărat cele ce ajută etapa de sinteză. Modelarea poate fi condusă de cerințe diferite, ca lizibilitate, flexibilitate, mentenabilitate, testabiliatate a modelului, care sunt importante în acest pas, dar auxiliare pentru co-sinteză. Modelele create în primul pas pot avea o legătură slabă cu cerințele etapei de sinteză sau pot să cuprindă aspecte care dacă ar fi sintetizate ar deteriora grav performanțele implementării. De aceea, specificarea trebuie supusă eventual unui *pas de transformare*, care să o adapteze la cerințele sintezei. Principalele tehnici de transformare sunt reorganizarea specificării, contopirea și desfacerea proceselor, expandarea apelurilor de funcții, gruparea instrucțiunilor în proceduri și a variabilelor în tablouri, etc. Problema creșterii calității implementării prin rescrierea specificărilor este încă insuficient studiată în literatura de specialitate.

Calitatea alocării și a partiționării trebuie evaluată pentru a stabili măsura în care sunt îndeplinite constrângerile de proiectare impuse. *Estimarea* implementării cuprinde doi parametrii, *performanța* și *costul* (aria) său. Maniera de evaluare implicită este aceea de a sintetiza hardware-ul și de a produce software-ul, iar apoi culegerea informațiilor de estimare dorite. Această tehnică este însă foarte consumatoare ca și timp și resurse de calcul și practic nu permite analiza de cât a unui număr redus de soluții de proiectare. Din fericire, la nivelul etapelor de proiectare nu sunt necesare estimări foarte exacte. Oricum, *principial* nu s-ar putea construi tehnici de estimare foarte exacte din cauza optimizărilor operate la diferite nivele și care fac ca relația între funcționalitate și elementele din hardware și software să nu fie unu la unu. Optimizările pentru software cuprind folosirea memoriei cache, pipelining, instrucțiuni multiple, etc. în scopul de a reduce dimensiune codului și de a mări viteza lui de execuție. În hardware, optimizările vizează reducerea logicii de control, minimizarea stărilor și maniera de mapare a resurselor din partea de date. O metodă de estimare rapidă este cea a *modelelor de așteptare* (*queing models*), unde fiecare funcționalitate are asociate valori statistice ale timpilor de execuție și ratelor de comunicare și de unde sunt obținute aprecieri statistice ale timpilor de execuție și ratelor de comunicare pentru întregul sistem. O metodă mult mai precisă este cea a *estimării la nivel de program*. Performanța medie a sistemului este stabilită folosind informații culese prin execuția

specificării pentru valori de intrare tipice (*profiling*). Funcțiile din hardware sunt mapate componentelor de nivel RTL şi este stabilită frecvența de execuție minimă /maximă/medie pentru fiecare pas de control. Timpul de excuție este obținut înmulțind frecvențele de execuție ale paşilor de control cu timpii lor de execuție. Software-ul este compilat şi cunoscând, prin profiling, frecvența medie de execuție a fiecărei instrucțiuni este estimat timpul său total de execuție. Estimările pentru software trebuie să includă faptul că taskurile concurente legate aceluiaşi procesor trebuie executate secvențializat.

Rafinarea specificărilor

După ce în etapa de explorare este aleasă alternativa de implementare, detaliile devenite astfel disponibile sunt adăugate la specificarea de sistem. Detaliile adăugate se referă la *memorii*, *interfețele* dintre componente şi *schema de arbitrare* a accesului la resursele comune mai multor procese. Specificarea rafinată ce rezultă este compusă dintr-un set de componente interconectate între ele, fiecare componentă fiind descrisă comportamental. Putem afirma că specificarea rafinată este o combinație între descriere comportamentală şi descriere structurală. Adăugarea detaliilor referitoare la *memorie* este în trei paşi: gruparea variabilelor în memorii, dezvoltarea modelului de funcționare a memoriilor şi adăugarea protocolului de acces la memorie în toate locurile specificării, unde sunt accesate variabilele. Componentele alocate prin partiționare la resurse diferite trebuie să comunice între ele. Principalele trei probleme tratate la *interfațare* sunt: stabilirea lățimii magistralei, generarea protocolului de comunicare şi adaptarea protocoalelor de comunicare diferite. Când este fixată lațimea magistralei, se are în vedere cerința ca să se folosească cât mai puține fire de conexiune, astfel încât constrângerile de timp să fie satisfăcute. Generarea protocolului stabileşte felul controlului (*handshake*, *half handshake*, *fixed time acces*), modul de diferențiere a datelor destinate adreselor diferite şi maniera de descompunere a datelor pentru transmiterea serială. Etapa de arbitrare are rolul de a soluționa accesele simultane la aceeaşi resursă, astfel încât un singur proces să o acceseze la un moment dat. Schemele de arbitrare sunt în principiu de două feluri: cu *prioritate statică* şi cu *prioritate dinamică*. Dacă există mai multe procese care doresc să acceseze aceeaşi resursă, întotdeauna este ales cel cu prioritatea mai mare. Prioritatea statică este fixată proceselor înaintea execuției lor, implică regie de sistem mai mică, dar conduce spre performanțe mai mici de cât prioritatea dinamică. Prioritatea dinamică este atribuită proceselor în timpul execuției funcție de felul în care ele accesează resursa comună, dar aceasta creşte regia necesară.

Sinteza componentei software

Sinteza componentei software este activitatea prin care descrierile de sistem sunt convertite în programe software, care pot fi compilate şi executate pe un procesor programabil. Principalele dificultăți legate de sinteza software provin de la aspectele specificării de sistem care nu pot fi traduse direct în construcții ale limbajului de programare. Secvența de cod ce se produce trebuie să fie echivalentă semantic cu construcția din specificarea de sistem. Generarea software-ului trebuie să țină cont de constrângerile de timp care sunt descrise pentru fiecare task şi să le planifice pe acelea ce sunt mapate aceluiaşi procesor programabil.

Sinteza părtii hardware

Componenta hardware este sintetizată ca ASIC-uri sau implementată folosind FPGA-uri. Sinteza hardware parcurge următoarele etape:

- *Sinteza la nivel înalt* (*High Level Synthesis*) transformă descrierea comportamentală a sistemului într-una la nivelul transferului între regiştri (*register transfer level*). Descrierea RTL cuprinde un controler, realizat ca o maşină cu un număr finit de stări, şi o parte de date (*data path*) pentru realizarea operaţiilor necesare. Partea de date este formată din regiştrii, ALU, multiplexoare, etc.
- *Sinteza secvenţială* (*Sequential Synthesis*) şi *sinteza la nivel logic* (*Logic Synthesis*) transformă maşina cu număr finit de stări într-o implementare compusă dintr-un registru de stare, circuitele combinaţionale ce produc starea următoare şi semnalele de ieşire ale controlerului. Activităţile de proiectare executate la acest nivel sunt minimizarea numărului de stări, codificarea stărilor şi minimizări logice.
- *Maparea tehnologică* (*Technology Mapping*) transformă reţeaua logică independentă de tehnologie, produsă în etapele de proiectare precedente, într-o reţea standard pentru o bibliotecă tehnologică anume.

Simulare şi cosimulare

În urma fiecărui pas de rafinare, specificarea este validată prin *simulare* sau *verificare formală*. Specificarea trebuie să fie corectă şi completă. Metoda *verificării formale* defineşte pentru o specificare un set de proprietăţi, care sunt demonstrate prin tehnici specifice ca fiind adevărate. Metoda populară de validare este *simularea*, care execută specificarea şi compară valorile de ieşire astfel produse cu valorile unei funcţionări corecte. Pentru că proiectarea hardware/software adresează nivele diferite de abstractizare, este necesar ca fiecare din ele să poată fi simulate. *Cosimularea* este procesul prin care sunt integrate simulatoare destinate la nivele diferite de abstractizare. De exemplu, o situaţie foarte frecventă este cea în care modelul RTL al componentei hardware este simulat împreună cu modelul de executare pe un procesor a instrucţiunilor software. Principala decizie ce trebuie luată la cosimulare este raportul între viteza de simulare şi corectitudinea ei. Cu cât simularea este mai corectă, cu atât trebuie culese mai multe informaţii, ceea ce înseamnă automat şi un timp de simulare mai lung. Nivelele de abstractizare la care se desfăşoară cosimularea sunt: *nivelul funcţional* este util pentru definirea produsului, marketing sau specificarea lui către producători. *Nivelul sistem* permite stabilirea resurselor puternic constrânse (*bottleneck*) şi verificarea modului de satisfacere a constrângerilor de timp. *Nivelul RTL* validează funcţionarea corectă a ASIC-urilor din arhitectura sintetizată şi a componentei software. *Nivelul fizic* verifică proprităţile electrice şi temporale ale ASIC-urilor şi ale chipurilor standard.

Ciclul de proiectare hardware/software

Construirea unui mediu integrat de proiectare hardware/software înseamnă rezolvarea a două probleme de naturi diferite:

- Construirea uneltelor software pentru activităţile de proiectare componente, de exemplu partiţionare, planificare, sinteza arhitecturii, etc. Acest lucru presupune implementarea unei anumite tehnici, care este dependentă de criteriile urmărite prin

procesul de proiectare. De exemplu, algortimii de planificare depind covârşitor de obiectivele planificării: minimizarea timpului de execuţie al sistemului, obţinerea unui grad de paralelism cât mai ridicat al soluţiei, minimizarea timpului de răspuns pentru anumite componente, etc.

- Reunirea activităţilor de proiectare într-un ciclu de proiectare. Acest lucru presupune determinarea etapelor de proiectare ce trebuie parcurse şi fixarea ordinii în care ele sunt executate. Forma ciclului de proiectare depinde atât de criteriile de proiectare dorite, dar el poate fi adaptat şi optimizat pentru o aplicaţie particulară.

Majoritatea cercetărilor în domeniul proiectării automate hardware/software sunt orientate pentru aplicaţiile de un anumit tip (sisteme în timp real, sisteme embedded, sisteme dominate de control, etc.). În consecinţă ciclul de proiectare are o structură fixă, ce acomodează în medie cel mai bine proprietăţile aplicaţiilor clasei respective. Acestei abordări i se pot opune următoarele contraargumente:

- În general este greu de încadrat o aplicaţie într-o clasă unică, pentru că ea cuprinde caracteristici aparţinând la mai multe tipuri de aplicaţii. Un exemplu tipic este cel al sistemelor embedded, care trebuie să aibă şi restricţii de timp real.
- Aplicaţiile dintr-o clasă au particularităţi funcţie de care se poate eficientiza ciclul de proiectare. Etapele necesare ale ciclului de proiectare depind de aplicaţia concretă şi nu are sens parcurgerea etapelor nenecesare. De exemplu, nu are sens să studiem alternative de execuţie paralelă a aplicaţiilor ce nu cuprind activităţi concurente.
- O anumită tehnică de proiectare (de exemplu de planificare) poate avea o comportare superioară pentru o aplicaţie şi mai slabă pentru alta. De exemplu, anumiţi algoritmi de planificare se comportă diferit pentru grafuri (sisteme) de diferite forme. Pe bază de experimente se pot indica proprietăţile sistemelor ce sunt potrivite unei anumite tehnici şi deasemenea pe cele ale sistemelor pentru care soluţiile obţinute sunt slabe. Deasemenea, pe parcursul timpului apar algoritmi noi din ce în ce mai eficienţi şi mai inteligenţi, care trebuie să înlocuiască în ciclul de proiectare algoritmii inferiori. Este de dorit ca această substituţie să aibă loc cât mai uşor şi cu cât mai puţine modificări ale mediului integrat de proiectare.

De aceea, ni se pare mult mai practic ca în loc să încercăm să propunem un ciclu de proiectare cu o structură unică şi fixată, să lăsăm la latitudinea proiectantului să instanţieze acest ciclul funcţie de caracteristicile aplicaţiei particulare. Totodată, în acest mod el îşi foloseşte experienţa proprie de proiectare atunci când propune activităţile de proiectare şi secvenţa de parcurgere a lor.

Un mediu CAD pentru proiectarea (automată) hardware software trebuie să includă pe lângă uneltele software pentru activităţile componente şi pe cele pentru gestiunea ciclului de proiectare. Aceste unelte trebuie să permită următoarele taskuri:

- *Specificarea fluxului de proiectare*: Proiectantul stabileşte care sunt paşii de proiectare ce trebuie parcurşi şi în ce secvenţă. Informaţia folosită în luarea deciziilor de proiectare depinde de obiectivele dorite şi este culeasă explicit din specificare sau rezultă ca urmare a unor etape precedente de proiectare. Acest lucru oferă proiectantului un indiciu despre activităţile de proiectare ce trebuie cuprinse în ciclu. Construcţiile pentru descrierea ciclului de proiectare trebuie să permită descrierea activităţilor succesive, a execuţiei iterative a unor grupuri de activităţi şi a execuţiei condiţionate de anumite mărimi. Descrierea ciclului de proiectare este

folosită pentru instanțierea (automată) ciclului folosit pentru implementarea unei aplicații particulare.

- *Instanțierea uneltelor particulare pentru fiecare pas al proiectării*: mediul de proiectare poate include unelte diferite destinate aceluiași pas de proiectare. Proiectantul trebuie să aibă posibilitatea de a alege și experimenta fiecare unealtă dorită.
- *Interfațarea automată și cât mai ușoară a diverselor instrumente de proiectare*: de regulă instrumentele au formate specifice pentru datele de intrare și de ieșire. Este necesară ca adaptarea formatelor diferite să fie automată și cu cât mai puțină intervenție din partea proiectantului uman.

1.5 Nouă perspective asupra proiectării hardware/software

Acest paragraf prezintă nouă viziuni diferite asupra problemei proiectării hardware /software. Discutăm în special abordările cu un impact major asupra evoluției cercetării în domeniu. Ele se deosebesc în principal prin aria tehnică cărora le sunt destinate (sisteme embedded, sisteme în timp real, prelucrarea semnalelor digitale) și în consecință prin activitățile de proiectare pe care le studiază, soluțiile pe care le propun pentru acestea și maniera în care activitățile de proiectare sunt combinate într-un ciclu unitar de proiectare.

PRISM-I (Processor Reconfiguration Through Instruction Set Mertamorphosis)

Sistemul PRISM-I [AS93] este bazat pe principiul că majoritatea timpului de execuție al unui program este petrecută într-o porțiune restrânsă a codului său. Dacă arhitectura sistemului este adaptată acestei porțiuni, timpul de execuție al programului poate fi îmbunătățit în mod spectaculos. PRISM-I identifică părți din funcționalitate pentru a le muta în hardware (practic crește setul de instrucțiuni ale procesorului), astfel încât să accelereze viteza sistemului. PRISM-I integrează activitățile de identificare/extragere a porțiunilor de mutat în hardware, de sinteză a părții mutate și de compilare a părții rămase în software. Elementul prin care autorii urmăresc să accelereze execuția unui program este folosirea unei arhitecturi adaptate, fără însă a adresa al doilea mijloc și anume acela de a crește gradul de concurență al soluției. Arhitectura implementării cuprinde un procesor programabil și o componentă reconfigurabilă (FPGA), conectate între ele printr-o magistrală. PRISM-I produce imaginea hardware pentru inițializarea FPGA-ului și imaginea software, care este încărcată în memoria procesorului și care cuprinde părțile rămase în software și rutinele pentru comunicarea cu hardware-ul.

Specificarea sistemului este descrisă în C și cuprinde întreaga funcționalitate a sistemului. Etapa următoare este identificarea/extragerea funcțiilor ce sunt mutate în hardware. Candidații pentru această mutare sunt funcțiile cu impactul cel mai mare asupra timpului total de execuție al implementării. Determinarea candidaților folosește o metodă de estimare a timpului de execuție al sistemului, folosind modele pentru arhitectura hardware și instrucțiunile ei, probabilitățile de execuție ale secvențelor de

cod, etc. Estimarea este limitată de factori ca influenţa execuţiei asupra dependenţelor de date, necunoaşterea numărului de iteraţii ale ciclurilor, inexactităţi ale predicţiilor privind execuţia instrucţiunilor *if*, etc. De aceea procesul de identificare/extragere este unul hibrid, pentru că PRISM-I indică o listă a candidaţilor pentru mutarea în hardware, iar proiectantul decide pe cele care sunt mutate efectiv.

Părţile extrase sunt sintetizate în vederea implementării lor în FPGA. Autorii consideră acest pas ca fiind cel mai dificil al metodologiei lor. Compilatorul C produce o reprezentare intermediară, tradusă într-un graf cu dependenţe de date, care apoi este convertit într-o descriere structurală pe baza bibliotecii de resurse hardware disponibile. Descrierea structurală este tradusă într-un set de ecuaţii booleene şi optimizate în vederea implementării lor în resursa reconfigurabilă. Sinteza hardware produce imaginea necesară configurării platformei FPGA. Partea rămasă neextrasă formează componenta software. Ea este compilată şi i se adaugă apelurile funcţiilor pentru accesarea părţilor mutate în hardware. Comunicarea între software şi hardware este iniţiată întotdeauna de software, care trimite mediului reconfigurabil argumentele de intrare necesare, mediul reconfigurabil îşi execută funcţia, după care returnează rezultatul către software. Implementarea mixtă hardware/software duce la micşorarea timpului de execuţie, doar dacă timpul celor trei etape este inferior timpului de execuţie a funcţiei în software. Imaginea software-ului cuprinde codul compilat şi biblioteca fucţiilor de acces la hardware.

Arhitectura experimentată practic este formată din procesorul Motorola 68010 la 10Mhz şi Xilinx 3090, interconectate printr-o magistrală de 16 biţi. Rezultatele obţinute au fost promitătoare (accelerări de până la 50 de ori), deşi autorii indică şi alte elemente pentru creşteri ulterioare ale performanţelor. Folosirea microprocesoarelor moderne, având un timp de acces la magistrală mai scurt, poate îmbunătăţi cu un ordin de mărime performanţele soluţiei.

Chinook

Chinook [BC96][CB94] este destinat proiectării sistemelor reactive, în timp real. Aplicaţiile adresate sunt cele dominate de control, iar arhitectura produsă este formată din componente off-the-shelf. Intenţia autorilor a fost ca Chinook să automatizeze etapele de proiectare, care rezolvate eficient aduc cele mai mari beneficii calităţii implementării finale, sunt mari consumatoare de timp de proiectare şi care abordate manual introduc erori cu o probabilitate mare. Chinook porneşte de la specificarea de nivel înalt a funcţionalităţii sistemului şi produce specificarea globală necesară implementării.

Specificările de intrare folosesc limbajul Verilog pentru descriere şi sunt formate din două componente: specificarea *comportamentală* a sistemului reactiv în timp real şi componenta *structurală* formată din procesoarele folosite, dispozitivele periferice şi interfeţele de comunicare. Proiectantul decide manual resursa unde este executată fiecare activitate a funcţionalităţii şi deasemenea stabileşte numărul şi tipul resurselor folosite. Specificarea comportamentală este compusă din *moduri*, care descriu comportarea sistemului pentru o secvenţă de intrare. Modurile pot avea impuse

constrângeri de timp pentru activitățile componente. Constrângerile de timp sunt orientate către evenimente observabile și sunt modelate prin separări minime/maxime între evenimentele de I/O. Ele se traduc la nivel scăzut în timpi de setup, hold, secvențieri între operații de I/O consecutive, etc. și în constrângeri ale timpilor de răspuns și ale ratei de prelucrare la nivel înalt. Dacă constrângerile de timp nu pot fi satisfăcute, părți din funcționalitate sunt puse în hardware sau sunt alocate resurse noi.

Arhitectura finală este compusă din unul sau mai multe procesoare care execută funcționalitatea sistemului și pentru care Chinook sintetizează automat interfețele de comunicare. După ce proiectantul indică pentru fiecare mod procesorul căreia îi este legat, Chinook determină automat datele care trebuie comunicate și sintetizează hardware-ul interfeței și generează driverele folosite.

Sintezei interfețelor este o problemă de proiectare hardware/software, mai exact de bipartiționare hardware/software, astfel încât implementarea să satisfacă un set impus de constrângeri de timp și costul său să fie minim. Interfața este modelată printr-un graf ponderat, care este apoi partiționat în două mulțimi cu algoritmul lui Kernighan-Lin, astfel încât numărul conexiunilor între ele să fie minim.

Sinteza interfețelor acoperă mai multe nivele. La nivelul cel mai scăzut, Chinook sintetizează interfețele folosind constrângerile de timp indicate pentru semnalele de I/O și produce logica necesară și codul pentru drivere. La nivel mai ridicat, conectarea procesoarelor cu perifericele lor poate fi prin una din următoarele două metode: alocând porturile de I/O ale procesorului sau mapând porturile de I/O în memorie. Porturile procesorului pot fi alocate astfel încât un port să deservească mai multe periferice și Chinook produce logica necesară pentru folosirea în comun a portului. Dacă procesorul nu dispune de porturi de I/O dedicate, atunci ele sunt mapate în memorie. Chinook alocă adresele de memorie necesare, generează logica pentru decodificarea adreselor și generează codul pentru instrucțiunile *load/store* de acces la porturi.

Operațiile modurilor și cele din driverele sintetizate sunt planificate static și nepreemptiv, pentru a le mări viteza de execuție. Planificarea folosește o euristică de planificare bazată pe liste.

COMET (COdesign METhodology)

Obiectivul principal al proiectării hardware-software în sistemul COMET [KP96] este stabilirea optimă a părții de hardware, respectiv a părții de software a sistemului de implementat. Astfel, COMET adresează în special problema partiționării hardware-software.

Etapele de proiectare asupra cărora se concentrează COMET sunt:

- Specificarea de sistem
- Estimarea componentei hardware și software
- Partiționarea automată hardware-software

Specificarea de sistem este descrisă în VHDL şi C. Argumentele autorilor pentru o descriere mixtă sunt legate de faptul că ambele limbaje sunt foarte populare în domeniile cărora se adresează. Există şi sunt bine cunoscute compilatoare de C şi instrumente de simulare sau sinteză pentru VHDL. Dacă specificarea de sistem s-ar face într-un limbaj L_x, atunci ar fi necesară traducerile L_x în VHDL şi L_x în C. Deasemenea, ar trebui construite instrumente pentru managementul ciclului de proiectare, astfel că deciziile de proiectare operate asupra codului C sau VHDL să fie reflectate şi asupra specificărilor L_x.

Deoarece există criterii de proiectare pentru care C sau VHDL nu oferă construcţii, specificarea este completată cu un fişier de reguli (numit şi engineering specifications). Acest fişier descrie restricţiile de arie şi timp de execuţie ale implementării finale, opţiuni de partiţionare, opţiuni pentru interfeţele între hardware şi software, estimări ale costului pentru interfeţele hardware-software şi software-hardware etc.

Prin simularea specificării, se colectează informaţie de profiling despre iteratiile şi instrucţiunile if ale specificării. Pentru fiecare bloc de instrucţiuni se înregistrează de câte ori a fost acesta executat, iar apoi se determină probabilitatea de execuţie a alternativelor pentru if-uri şi numărul mediu de iteraţii pentru cicluri.

Specificarea iniţială este prepartiţionată folosind regulile din fişierul de reguli. În absenţa unor indicaţii explicite, funcţiile C formează partiţia software, iar cele VHDL pe cea hardware. Apoi, cele două partiţii sunt interfaţate folosind o structură generică pentru interfeţe. Astfel, interfeţele hardware-software sunt compuse dintr-o parte pentru generarea întreruperilor şi transmiterea datelor şi una pentru interfaţarea cu segmentul de cod ISR. Interfeţele software-hardware sunt formate din componenta pentru maparea în memorie a datelor transmise din software şi interfaţa hardware mapată în memorie.

Estimarea componetelor hardware şi software se face folosind informaţia obţinută prin profiling. Estimatorul software traduce codul C în limbaj de asamblare, pe care apoi îl foloseşte în a stabili numărul de biţi şi timpul necesar execuţiei fiecărei instrucţiuni. Estimatorul hardware construieşte graful dependenţelor de date şi de control, îl planifică, alocă un număr de unităţi funcţionale şi stabileşte timpul necesar pentru executarea codului VHDL.

Pentru partiţionarea hardware-software se construieşte un graf, având noduri pentru funcţiile şi procesele specificării, care sunt etichetate cu timpii lor medii de execuţie. Arcele între două noduri indică de câte ori unul îl apelează pe celălalt. Algoritmul de partiţionare urmăreşte realizarea unui compromis între timpul implementării şi mărimea memoriei necesare (acest criteriu a fost identificat de autori ca fiind restrictiv la această oră). Partiţionarea foloseşte modelul McCullock-Pitts pentru neuronul binar.

Cosyma (Cosynthesis for Embedded Architectures)

Cosyma [EHB93] este un instrument CAD pentru proiectarea hardware-software a sistemelor embedded cu microcontrolere. Cercetarea autorilor se concentrează asupra

specificării de sistem, a definirii unei reprezentări intermediare a sistemului menite să descrie atât hardware-ul, cât şi software-ul şi asupra partiţionării automate hardware-software. Deasemenea, ei indică două metode ca şi modalităţi de analiză a calităţii implementării: sintetizând componetele software şi hardware, simulând la nivelul transferului între regiştrii şi verificând satisfacerea constrângerilor de timp sau apelând la o metodă aproximativă, dar mult mai rapidă, de verificare a constrângerilor de timp [YE93]. În cazul în care constrângerile de proiectare nu sunt satisfăcute, procesul de proiectare continuă cu repartiţionarea hardware-software a sistemului. Cosyma urmăreşte să plaseze cât mai mult din funcţionalitate în software, lăsând pentru hardware interfeţele de I/O ale procesorului, precum şi porţiuni de funcţionalitate ce duc la violarea constrângerilor de timp impuse prin proiectare. Din acest motiv, metodologia de proiectare introdusă prin Cosyma este numită *orientată spre software* (software oriented).

Specificarea de sistem este descrisă în C^x, un supraset al limbajului C, şi menită să fie implementată în software. Specificarea poate cuprinde atât construcţii specifice software-ului (d.e. date dinamice), dar şi funcţii ce descriu structuri hardware predefinite. Constrângerile de proiectare cuprind timpi de execuţie minimi şi maximi, precum şi întârzieri minime şi maxime între perechi de instrucţiuni din specificare. Specificarea este transformată într-o reprezentare intermediară (graful sintactic extins), care este un hibrid între arbori sintactici (specifici reprezentărilor pentru software) şi grafuri cu dependenţe de date şi de control (deobicei folosite pentru reprezentarea structurilor hardware). În continuare, reprezentarea intermediară este cea folosită în ciclul de proiectare, pentru că ea permite migrarea uşoară a unei componente între hardware şi software. Verificarea corectitudinii funcţionale a specificării se face simulând GSE-ul specificării. Totodată, prin simulare se face şi profiling, menit să colecteze informaţie statistică privind execuţia porţiunilor din specificare (timpul de execuţie al unei porţiuni, de câte ori este executată o porţiune).

Arhitectura folsită de Cosyma se compune dintr-un procesor pentru software şi un coprocesor pentru hardware. Componenta software este compilată folosind un compilator pentru procesorul ţintă, iar hardware-ul este sintetizat cu sistemul Olympus [MI90]. Comunicarea între hardware şi software este sintetizată automat, detectându-se datele ce trebuie comunicate şi generându-se interfeţele pe bază de template-uri. Modelul de execuţie al sistemului consideră ca software-ul este cel care "activează" coprocesorul, transmitându-i datele necesare pentru execuţie, după care aşteaptă terminarea execuţiei acestuia. Prin urmare Cosyma simplifică implementarea, deoarece nu consideră paralelism între hardware şi software, fără a mai menţiona că nu foloseşte paralelism între componentele software, ce ar putea apare folosind o arhitectură cu mai multe procesoare programabile. Consecinţa acestui model simplificat al execuţiei, precum şi al faptului că arhitectura este dată, este că proiectarea hardware-software se reduce la partiţionarea hardware-software a specificării.

Autorii decid că partiţionarea trebuie efectuată la o granularitate scăzută (la nivel de bloc) pentru a mări potenţialul de optimizare a performanţelor. Ca şi algoritm de partiţionare este folosit simmulated annealing, împreună cu o funcţie de cost care surprinde variaţia de performanţă obţinută prin mutarea unui bloc din software în hardware. Această variaţie depinde de diferenţa între timpii de execuţie ai blocului în hardware şi software, de timpul de comunicare necesar pentru transmiterea datelor

către bloc şi de numărul de ori cât blocul este executat. Prin urmare, funcţia de cost surprinde doar constrângerea de timp din specificare, costul sistemului rezultând din numărul de unităţi funcţionale precizate de proiectant. Adăugarea costului sistemului la funcţia de cost este menţionată ca una din intenţiile de cercetare viitoare ale grupului.

O extindere a sistemului Cosyma este prezentată în [BE95]. Autorii folosesc aceleaşi ipoteze legate de arhitectura implementării şi de modelul de execuţie al implementării, dar consideră că specificarea sistemului este formată dintr-un set de procese cu dependenţe între ele. În primul pas, setul de procese este liniarizat, respectându-se restricţiile de dependenţă. Apoi pentru setul liniarizat se verifică constrângerile de timp, iar dacă acestea nu sunt satisfăcute se calculează accelerarea necesară sistemului pentru a se încadra în constrângerile indicate. Stabilirea accelerării necesare se face prin căutare binară şi verificând pentru fiecare valoare a accelerării dacă ea duce la satisfacerea limitelor de timp. Apoi este folosit sistemul Cosyma pentru a sintetiza un coprocesor pentru secvenţa liniarizată de procese, astfel că accelerarea obţinută să fie egală cu accelerarea stabilită prin căutare binară.

Sistemul Ptolemy

Principiul sistemului Ptolemy [KL93] este original, pentru că spre deosebire de restul sistemelor de proiectare automată, ce încearcă să combine aspecte heterogene într-o reprezentare unitară, Ptolemy adaptează şi interfaţează semantici aparţinând la modele de reprezentare diferite. Autorii motivează că sistemele surprind aspecte de diverse naturi, care trebuie tratate prin metode specifice. Ptolemy este un mediu pentru prototipizarea şi simularea sistemelor heterogene, destinat în mod particular proiectării procesoarelor de semnal (DSP), aplicaţii cu o complexitate computaţională medie, dar viteză ridicată de funcţionare.

Modelarea sistemelor este în tehnologie orientată pe obiecte, integrate într-un tot unitar. Un model este compus din *blocuri* (echivalentul proceselor), care comunică folosind mesaje şi interfeţe de comunicare, denumite *portholes*. Un bloc poate fi activat când datele sale de intrare devin disponibile, iar rezultatele sale sunt comunicate către porthole-urile de ieşire. Secvenţa de execuţie a blocurilor este stabilită de un planificator, funcţie de semantica operaţională a setului de blocuri ce interacţionează. Ierarhii complexe de blocuri pot fi obţinute prin mecanismul de derivare. *Target* este derivat din clasa bloc şi stabileşte maniera de interpretare (semantica) a blocurilor. Dacă descrie o semantică orientată pe simulare, atunci target-ul apelează planificatorul pentru a stabili ordinea de execuţie a blocurilor, iar pentru semantica orientată pe sinteză, generează codul blocurilor pentru DSP şi îl execută.

Principiul de simulare în Ptolemy este cel multiparadigmă şi anume modelul global al sistemului este format din modelele subsistemelor, reprezentate în stilurilor de modelare şi nivele de abstractizare diferite. *Domeniul* descrie complet un model computaţional şi este format din ierarhia de blocuri, target-ul ataşat şi planificatorul pentru implementarea semanticii operaţionale dorite. Subsistemele unui sistem pot fi modelate folosind modele diferite, dar sunt integrate printr-un mecanism de interfatare, denumit *wormholes*, şi care realizează transferurile de date necesare şi interacţiunea

între schedulerele domeniilor. Domeniile implementate în Ptolemy sunt cele pentru fluxuri sincronizate de date (*SDF*) şi hardware digital (*Thor*). SDF este folosit pentru modelarea componentelor off-the-shelf, cu o funcţionare binecunoscută şi presupune că operaţiile sunt planificate static şi devin pregătite pentru execuţie când operanzii necesari sunt disponibili. Thor simulează la nivelul RTL funcţionarea hardware-ului digital. Simulatorul simulează setul de instrucţiuni, funcţionarea bussului şi modificarea semnalelor la pinii circuitului. Valorile prezente la pini sunt transformate în valori semnificative pentru restul simulatorului, care la rândul său produce semnale tipice către pinii procesorului. Conectarea modelelor Thor şi SDF, aflate la nivele diferite de abstarcţiem, este printr-o wormhole.

Sinteza foloseşte target-urile pentru semantica orientată pe sinteză. Hardware-ul este sintetizat în doi paşi: este generat cod Silage pentru fincţiile din hardware, iar codul este sintetizat folosind sistemele existente pentru sinteză la nivel înalt. Codul pentru software-ul este generat funcţie de procesorul ţintă, felul limbajului (C, limbaj de asamblare), tipul arhitecturii (arhitecturi uniprocesor, multiprocesor, multiprocesor cu memorie comună, etc.), etc. Generarea codului include partiţionarea sa pe procesoare, palniifcarea sa şi adăugarea comunicărilor interprocesoare.

Un ciclu tipic de proiectare hardware/software cu Ptolemy decurge în felul următor. Specificarea sistemului este descrisă în mod unitar în domeniul SDF, fără a include nici un fel de aspecte de implementare sau constrângeri de proiectare. Apoi, specificarea este partiţionată automat [KL94] sau manual în hardware şi software. Ptolemy generează automat interfeţele de legătură (wormhole) între cele două partiţii, ceea ce păstrează unitatea modelului deşi au fost identificate două componente de natură diferite. Proiectantul decide manual numărul şi tipul procesoarelor programabile, ceea ce instanţiază felul în care este generat codul. Deasemenea el stabileşte interfeţele hardware-software, adică fie folosirea porturilor seriale sau maparea porturilor în memorie, şi logica de interfaţare necesară. Pentru această structură hardware, Ptolemy produce domeniile de simulare Thor. Proiectantul verifică prin simulare modul în care sunt satisfăcute constrângerile (mărimea codului, constrângerile de timp) şi dacă ele nu sunt satisfăcute, atunci sunt adăugate resurse noi, blocurile sunt replanificate sau mutate pe alte resurse mai libere sau sistemul este repartiţionat. Ciclul de proiectare este reluat iterativ până când soluţia de implemetare este cea indicată prin specificare.

Sistemul SOS (Synthesis of Systems)

SOS [PP92] este un instrument CAD pentru sinteza optimală a sistemelor heterogene, multiprocesor, dedicate aplicaţiilor. SOS răspunde la următoarele probleme de proiectare: sintetizează arhitectura stabilind numărul şi tipul elementelor de procesare şi a interconexiunilor între ele, alocă taskurile procesoarelor şi conexiunilor şi planifică taskurile alocate aceleiaşi resurse.

SOS modelează funcţionalitatea unui sistem, printr-un graf cu dependenţe de date şi ia în considerare comunicările între procese. Nodurile grafului sunt adnotate cu timpii de execuţie ai activităţilor modelate şi aceşti timpi sunt numere întregi.

SOS generează arhitecturi formate din procesoare software, ce serializează execuția taskurilor alocate lor. Fiecare procesor are memoria sa locală şi sistemul nu are memorie globală. Procesoarele au porturi de I/O, iar comunicarea are loc concurent cu execuția altor procese. Fiecare procesor este caracterizat prin cost, performanțe (un proces are timpi de execuție diferiți pe procesoare diferite) şi funcționalitate (deoarece nu orice proces este executabil pe orice procesor). Un proces este executat de un singur procesor în manieră nepreemptivă. Obiectivul proiectării este minimizarea costului total al sistemului şi al timpului toatl de execuție.

Autorii descriu formal sinteza sistemului prin programare matematică. Ei dezvoltă un model MILP (mixed integer linear programming), cu relații pentru secvențierea activităților, alocarea activităților pe resurse, respectarea dependențelor de date şi cu funcția obiectiv de a minimiza timpul de execuție şi costul sistemului. Rezolvând modelul MILP, SOS găseşte soluția optimă de implementare a sistemului.

Din cauza metodei de proiectare, SOS nu poate fi aplicat de cât pentru sisteme mici, cu un număr redus de noduri (exemplele autorilor au 4, respectiv 9 noduri). Totuşi, SOS este flexibil şi extensibil prin incorporarea unor obiective şi cerințe noi de proiectare. Deasemenea, autorii consideră metoda foarte importantă pentru înțelegerea problemei proiectării sistemelor heterogene şi creză baza pentru abordări ulterioare, cum ar fi proiectarea de euristici eficiente. SOS incorporează interconectări poit-to-point şi prin bus, dar el poate fi extins şi pentru cele de tip ring sau mixte. Deasemenea, el poate fi extins pentru a acoperi şi alte lacune de modelare, cum sunt: modelarea memoriei locale şi globale, modelarea operațiilor de I/O, respectiv situația când operațiile de I/O nu pot fi suprapuse peste restul calculelor, costul operațiilor de I/O, etc.

Metodologia grupului de la Carnegie Mellon University

Activitatea grupului este focalizată pe co-simulare şi co-sinteză. Scopul proiectării hardware/software a grupului de la CMU este să accelereze execuția unui sistem prin mutarea unor părți ale funcționalității din hardware în software [TAS93]. Implementarea finală trebuie să satisfacă constrângerile impuse folosind minimul de hardware. Metoda consideră că controlul concurenței este foarte important pentru calitatea finală a proiectării. Concurența este definită de funcționalitatea sistemului şi de interacțiunile între componente. Gradul de concurență dorit pentru implementare este obținut prin restructurarea specificării (spargerea sau contopirea proceselor) sau mutarea funcționalității între componentele hardware şi software.

Specificarea de sistem este formată dintr-un set de procese Verilog, care interacționează între ele conform modelului CSP. Comunicarea între procese foloseşte primitivele pentru transferuri sincronizate şi nesincronizate de date şi sincronizări fără transfer de date. Autorii consideră că specificarea sistemului trebuie să satisfacă următoarele cerințe: să descrie funcționalitatea sistemului, să ascundă detalii de nivel scăzut cum sunt sistemul de operare folosit sau arhitectura hardware, să indice explicit concurența prezentă în sistem şi să faciliteze restructurarea sa în vederea creşterii gradului de concurență şi a partiționării hardware/software.

Co-simularea sistemului permite obținerea unei specificări funcțional corectă, dar care nu satisface anumite constrângeri de proiectare sau nu poate fi realizată în tehnologia curentă.

Deoarece procesele nu sunt întotdeauna cele mai bune grupări în vederea partiționării, există posibilitatea ca ele să fie descompuse sau regrupate în secvențe netriviale de operații, denumite taskuri. *Partiționarea hardware/software* este la granularitatea ridicată stabilită de task-uri. Implementarea unor taskuri este decisă de funcționalitatea lor. Taskurile cu multe apeluri de sistem sunt mapate în software, iar cele care interacționează intens cu exteriorul sunt plasate în hardware. Criteriile de partiționare cuprind costul implementării hardware, diferența timpului de execuție între implementările software și hardware ale unui task și respectiv impactul pe care timpul de execuție al unui task particular îl are asupra performanței întregului sistem. Taskurile destinate hardware-ului sunt cele care au un paraleleism ridicat, pot beneficia în urma prezenței memoriei dedicate sau au thread-uri multiple de control.

Arhitectura sintetizată este formată dintr-un singur CPU, ASIC-uri, magistrale, memorii, periferice de I/O și sistemul de întreruperi.

Metodologia grupului de la Stanford

Metoda aceasta de proiectare [GM93] este una dintre cele mai populare și este denumită în literatură ca fiind "orientată pe hardware" (*hardware oriented*). Ea este destinată sintezei sistemelor mixte hardware/software în timp real, care sunt executate pe o arhitectură formată dintr-un procesor programabil pentru componenta software și un ASIC pentru cea hardware. Cercetarea este focalizată pe trei probleme principale: estimarea performanțelor unei implementări, maparea funcționalității în hardware sau software și verificarea modului în care implementarea satisface proprietățile funcționale și temporale impuse. Principiul proiectării este acela de a porni de la o soluție inițială ce pune funcționalitatea cu precădere în hardware, care apoi este îmbunătățită prin mutarea operațiilor în software, astfel încât constrângerile temporale să rămână satisfăcute, iar costul implementării să scadă.

Funcționalitatea sistemului este descrisă în HardwareC și autorii insistă că metodologia lor de proiectare nu depinde de descrierea sistemului în acest limbaj particular. HardwareC a fost ales pentru a folosi sistemul Olympus [MI90] pentru sinteza componentei hardware. Specificarea este compusă din procese concurente ce comunică prin mesaje. Procesele sunt executate într-un ciclu infinit, astfel că ele sunt repornite în momentul în care execuția corpul lor este încheiată. Corpul proceselor este format atât din blocuri de instrucțiuni secvențiale cât și din blocuri executate concurent. Constrângerile temporale ale specificării sunt acoperitoare pentru a modela cerințele aplicațiilor de timp real și sunt descrise prin întârzieri *min/max* între operații sau indicând limita de timp între două execuții consecutive ale aceleiași operații.

Specificarea este tradusă într-o reprezentare de tip graf, ce descrie atât funcționalitatea sistemului cât și constrângerile de timp. Nodurile indică activitățile specificării, iar arcele dependențele între ele. Reprezentarea evidențiază în mod explicit concurența

sistemului şi uşurează analiza proprietăţilor sale. Graful conţine operaţii ce au asociat un timp de execuţie dependent de resursa ce le execută, dar şi operaţii cu timpi *nedeterminaţi* (operaţii ND), corespunzătoare ciclurilor cu număr neprecizat de iteraţii sau apelurilor lui *receive*. Constrângerile de timp ale specificării sunt incluse în graf prin ponderile pozitive şi negative ce adnotează arcele grafului.

O etapă de bază, dar şi individualizantă, a acestei metodologii de proiectare o reprezintă *analiza modelului*. Scopul ei este de a verifica după un set de reguli stabilite, dacă constrângerile de timp pot fi satisfăcute de implementarea curentă. Dificultăţile de analiză provin din prezenţa în graf a operaţiilor ND. Activitatea de verificare identifică două situaţii de satisfacere a constrângerilor temporale. O constrângere de timp este *satisfăcută* dacă realizarea ei are loc pentru orice timp de execuţie al operaţiilor ND implicate şi este *satisfăcută marginal*, dacă ea este realizată pentru timpi de execuţie ai operaţiilor ND din anumite intervale. Satisfacerea marginală este utilă practic, pentru că introduce anumite criterii de implementare ale operaţiilor. Pentru a lărgi categoria specificărilor ce pot fi verificate pentru satisfacerea constrângerilor de timp, se consideră că în software, execuţia operaţiilor ND poate avea loc concurent cu cea a altor operaţii ce nu depind de ele. Software-ul cuprinde o mulţime de thread-uri concurente, ce încep sau se termină cu o operaţie ND, dar nu cuprind în interior astfel de operaţii. În acest mod, analiza pentru satisfacerea marginală a constrângerilor de timp este extinsă la operaţii din thread-uri diferite.

Efectul mapării funcţionalităţii la partiţiile hardware şi software este redat printr-o funcţie de cost. Procesul de partiţionare este ghidat de această funcţie, astfel încât partiţiei optime să îi corespundă valoarea minimă a funcţiei de cost. Funcţia de cost descrie atât costul şi mărimea hardware-ului şi a software-ului, cât şi performanţele temporale ale implementării. Autorii consideră că aspectele de timp sunt dificil de cuprins în funcţia de cost, din cauză că au o natură globală, pentru care modificările incrementale ale partiţiilor sunt greu de reflectat. Performanţele temporale ale implementării sunt caracterizate de latenţa sistemului, ce depinde de gradul de încărcărcare al procesorului şi de lăţimea de bandă a magistralei. Costul hardware-ului este aproximat funcţie de costul elementelor hardware particulare din componenţa sa, iar cel al interfeţei pe baza numărului de porturi necesare. Euristica de partiţionare porneşte de la o soluţie iniţială pe care o îmbunătăţeşte iterativ, mutând operaţii între cele două componente. Soluţia iniţială mapează toate operaţiile ND în software, iar restul în hardware. Mutând o operaţie în software, se modifică timpul său de execuţie, latenţa thread-ului din care face parte, gradul de încărcare al procesorului şi lăţimea de bandă a magistralei. Partiţia nouă este reanalizată pentru a verifica dacă constrângerile sale de timp rămân satisfăcute. Între mai mulţi candidaţi la mutare în software este ales cel care păstrează cât mai mică comunicarea hardware-software.

Componenta software este formată fie din subrutine sau din corutine. În practică este mai utilă generarea software-ului ca fiind compus din corutine, din cauza dependenţelor de date şi temporale ce apar între operaţii. Pentru a micşora regia de sistem necesară la execuţie, operaţiile unui thread sunt liniarizate, iar thread-urile sunt planificate dinamic în timpul execuţiei lor, din cauză că nu se cunoaşte static secvenţa în care ele se încheie. Planificarea thread-urilor are loc după strategia FIFO, în ordinea în care datele lor de intrare devin disponibile.

Metoda grupului de la Princeton (Ti-Yen Yen şi Wayne Wolf)

Ti-Yen Yen şi Wayne Wolf [YW95] studiază proiectarea hardware-software a sistemelor embedded, distribuite, pentru aplicaţii în timp real. Arhitectura menită să execute aceste aplicaţii este una heterogenă, formată din procesoare programabile de diferite feluri şi interconectate în diverse topologii. Potrivit celor doi cercetători, proiectarea hardware-software a unei aplicaţii stabileşte *arhitectura hardware*, formată din numărul, tipul şi modul de interconectare a elementelor de procesare, precum şi *arhitectura software*, compusă din modul de legare a funcţionalităţii sistemului pe elementele de procesare şi planificarea proceselor mapate aceluiaşi procesor. Implementarea finală oferă funcţionalitatea descrisă în specificare şi satisface în mod optim constrângerile impuse de proiectare.

Aplicaţiile sunt modelate prin grafuri aciclice, nodurile corespunzând proceselor, iar arcele indicând dependenţele de date între noduri. Un grup de procese cu dependenţe de date formează un task, care este caracterizat prin perioadă de execuţie, precum şi constrângeri proprii *hard* şi *soft* de timp. Constrângerile hard sunt cele care trebuie satisfăcute în mod obligatoriu de implementarea finală, iar cele soft trebuie optimizate, dar nu neapărat şi satisfăcute. Fiecare proces are un interval de execuţie căruia îi aparţine timpul său de execuţie pentru orice execuţie a sa. Intervalul de execuţie al unui proces reflectă structura sa de control (stabilită de instrucţiuni if sau de ciclare), dar include şi aspecte de arhitectură greu de surpins în expresia timpului de execuţie (de exemplu pipelining, memorie cache, etc.).

Elementele de prelucrare ale arhitecturii sunt deosebite prin cost (preţ, arie, putere consumată, etc.). Interval de execuţie al unui proces este dependent de elementul de execuţie, căruia i-a fost mapat. Procesele legate aceluiaşi procesor sunt planificate preemptiv folosind o prioritate statică, derivată din *inverse-deadline priority assignment* [LW82] şi stabilită euristic pe baza estimării timpului de execuţie al sistemului [YW96].

Constrângerile de proiectare ce trebuie să fie satisfăcute de implementarea finală cuprind constrângeri hard şi soft pentru timpul de execuţie ale taskurilor din graf şi pentru costul total al sistemului.

Metodologia de proiectare dezvoltată de Ti-Yen Yen şi Wayne Wolf este desfăşurată pe trei nivele:
1) Primul nivel sintetizează arhitectura. Acesta porneşte de la soluţia iniţială în care fiecare proces este mapat unui element de procesare distinct şi încearcă să elimine resursele slab utilizate, să echilibreze gradul de concurenţă al sistemului, respectiv să reducă costul total al implementării prin înlocuirea elementelor de procesare cu altele mai ieftine şi care satisfac atât funcţionalitatea proceselor mapate, cât şi constrângerile de timp impuse. Sinteza arhitecturii este ghidată de funcţii de cost specifice obiectivului urmărit la pasul curent. Dacă se doreşte eliminarea resurselor slab utilizate, atunci componenta funcţiei de cost pentru resursa cea mai puţin încărcată este produsul între preţul componentei şi gradul ei de încărcare. Prin minimizarea costului are loc migrarea proceselor de pe elementul cel mai puţin folosit spre alte componente. Când se

balansează concurența implementării, funcția de cost este suma costurilor elemetelor de prelucrare din arhitectură.
2) Nivelul doi găseşte pentru o arhitectură dată, legările proceselor pe resurse, astfel încât performanțele soluției să fie optime. Metoda folosită este cea a căutării bazate pe gradient (*gradient search method*), care analizează îmbunătățirea soluției prin remaparea la un pas a unui singur proces. Algoritmul de legare încearcă toate remapările posibile ale proceselor pe alte resurse de cât cea curentă şi pentru fiecare stabileşte gradul de îmbunătățire a performanțelor (se foloseşte mărimea numită *senzitivitate* şi care indică gradul de apropiere sau depărtare a soluției de optim). Dacă nici una din remapări nu duce la o creştere a calității implementării, atunci se încearcă alocarea unei resurse noi şi maparea către ea a unui proces. Din mai multe variante de îmbunătățire a performanțelor, algoritmul selectează acea remapare, care duce la cea mai bună creştere. Nivelul doi continuă până când remapările proceselor din arhitectură nu mai cresc performanțele implementării.
3) Nivelul trei reevaluează performanțele sistemului pentru o arhitectură şi o legare a proceselor date. Reevaluarea se face atribuind priorități proceselor, planificându-le şi verificând satisfacerea constrângerilor de timp impuse.

1.6 Definirea problematicii tezei de doctorat

Obiectivul global al acestei teze de doctorat este acela al *studiului proiectării automate hardware software a sistemelor numerice*. În mod particular, teza tratează trei activități ale ciclului de proiectare: *partiționarea automată hardware/software*, *planificarea activităților unui sistem* şi *generarea codului pentru componenta software*.

Partiționarea hardware/software are rolul de a reduce timpul de execuție al unei implementări a sistemului, atunci când costul hardware-ului şi al software-ului este păstrat sub anumite limite. Ceea ce distinge munca noastră de alte abordări destinate *partiționării hardware /software* este:

- Modelarea dependenței performanței implementării sistemului de partiția hardware /software. Noi considerăm că performanța implementării creşte prin reducerea comunicărilor între hardware şi software şi mărirea gradului de paralelism al implementării. Gradul de paralelism se referă atât la paralelismul în interiorul proceselor, cât şi la cel între procese.
- Formularea problemei partiționării hardware/software ca şi o problemă a bipartiționării unui graf şi implementarea euristicilor corespunzătoare. Euristicile trebuie să găsească soluții foarte bune (cât mai apropiate celor optime) şi să fie cât mai rapide, astfel încât să poată partiționa şi grafuri cu un număr foarte mare de noduri.
- Punerea la punct a unor modele realiste pentru experimentarea instrumentelor CAD. Modelele trebuie să fie din industrie, astfel încât să putem aprecia atât calitatea soluțiilor obținute, dar şi gradul în care CAD-ul proiectat acoperă nevoile practicii.

Planificarea proceselor trebuie stabilită în mod static pentru a prevede în mod determinist comportarea sistemului şi a garanta un anumit timp de execuție pentru cazul cel mai defavorabil (aspect esențial pentru sistemele în timp real). Scopul planificării noastre este acela de a stabili secvența statică de execuție a activităților pe

resursele arhitecturii, astfel încât timpul total al sistemului să fie minim. Principalele elemente specifice pe care tehnica noastră de planificare trebuie să le asigure sunt:

- Toate strategiile de planificare dezvoltate la această oră se bazează pe prezumția că arhitectura este formată exclusiv dintr-un număr fix de procesoare programabile și acestea sunt toate identice între ele. Această ipoteză este inacceptabilă având în vedere complexitatea problemelor de proiectare actuale și particularitățile arhitecturilor folosite.
- Algoritmii de planificare existenți stabilesc în același timp pentru fiecare proces resursa unde este executat și momentul când începe execuția sa. Din cauză că arhitectura noastră este heterogenă, timpul de execuție al proceselor depinde fundamental de resursa legată, astfel că în momentul în care sunt planificate procesele, ele trebuie să fi fost deja legate.
- Al treilea factor este acela că noi considerăm pe lângă dependențele de date între activități și dependențele de control. Aceste dependențe sunt introduse prin construcții *if* și *while* ale specificării și funcție de setul condițiilor adevărate este activată doar o parte din funcționalitatea globală (de exemplu, procesele din două ramuri complementare ale aceluiași *if* nu pot fi simultan executate). Acest fapt duce la algoritmi specifici pentru toate etapele proiectării și în mod particular la algoritmi tipici de planificare.

Deoarece folosirea dependențelor de control în planificarea proceselor nu a fost abordată anterior, nu există date de referință pentru compararea rezultatelor noastre experimentale. De aceea, trebuie propusă o manieră de experimentare a algoritmilor implementați și care să permită obținerea unor concluzii pertinente.

Pentru procesele din software este *generat automat cod C* independent de arhitectura țintă. Codul C trebuie să fie echivalent semantic cu procesele pe baza căruia a fost produs și trebuie să cuprindă toate construcțiile limbajului de specificare de sistem. Pentru a asigura migrarea ușoară a proceselor din hardware în software și invers, este de dorit ca software-ul să fie generat pe baza unei reprezentări interne cât mai apropiată de cea folosită pentru sinteza de hardware. Codul generat trebuie să fie cât mai rapid și să ocupe cât mai puțină memorie. De aceea generatorul de cod C efectuează optimizări independente de arhitectură pentru reprezentarea variabilelor și reducerea codului generat și tratează construcțiile proceselor și comunicările interprocese astfel încât timpul de execuție să fie cât mai mic.

1.7 Structura tezei

Capitolul 2 prezintă metoda de partiționare hardware/software avută în vedere și implementată de noi. El discută specificarea sistemului folosind VHDL, rafinarea specificării și stabilirea granularității de lucru pentru partiționare, metricile de partiționare și gruparea lor într-o funcție de cost. Euristicile de partiționare folosite sunt simulated annealing și tabu search.

Capitolul 3 reia problema planificării grafurilor cu dependențe de date și o particularizează pentru grafurile legate, având dependențe de date. În mod special

pentru grafurile legate sunt rediscutate mărimile folosite în planificare: ASAP, ALAP şi drumul critic. Aceste mărimi sunt utilizate pentru definirea unui criteriu rapid de estimare a timpului de execuţie a unui graf, fără planificarea explicită a nodurilor sale. Deasemenea, discutăm planificarea optimă prin branch-and-bound şi ILP. Capitolul se încheie cu prezentarea euristicii noastre bazate pe liste, în care definirea priorităţilor ţine cont de reinterpretarea noţiunii de drum critic pentru un graf legat.

Capitolul 4 defineşte formal reprezentarea internă folosită pentru proiectarea bazată pe dependenţe de date şi de control şi introduce restricţiile pe care reprezentarea trebuie să le îndeplinească. Deasemenea, definim formal problema planificării cu dependenţe de date şi de control şi propunem un set de reguli de transformare a unei specificări de sistem într-o reprezentare internă cu dependenţe de date şi de control.

Capitolul 5 tratează problema planificării grafurilor legate cu dependenţe de date şi de control. Capitolul prezintă arhitectura folosită pentru planificarea acestui tip de grafuri, identifică modul de manifestare a condiţiilor pe parcursul activităţii de planificare - funcţie de care propune o clasificare a condiţiilor şi intoduce algoritmii pentru stabilirea categoriilor de condiţii într-un graf. Apoi, identifică cerinţele generale pe care trebuie să le îndeplinească o soluţie de planificare deterministă şi discută diferite moduri de asigurare a cerinţelor funcţie de tipul condiţiilor. Capitolul tratează două euristici de planificare, una bazată pe liste şi cu nodurile având prioritate unică, iar a doua folosind tehnica ajustării şi potrivirii planificărilor pentru trace-urile individuale.

Capitolul 6 este destinat etapelor târzii ale co-sintezei hardware/software, adică sinteza de nivel înalt a componentei hardware şi generarea codului pentru componenta software. În capitol discutăm reprezentarea intermediară folosită atât pentru sinteza hardware-ului cât şi pentru generarea software-ului. Sunt indicate regulile după care sunt tratate acele construcţii ale specificării de sistem, ce nu au corespondent direct în limbajul de programare folosit pentru componenta software. În încheiere detaliem câteva din optimizările posibile ale programelor generate.

Capitolul 7 descrie experienţa noastră dobândită în urma proiectării hardware/software a blocului F4 din circuitul ATM. Ciclul de proiectare este particularizat pentru aplicaţia concretă şi pentru că ea are o funcţionalitate complexă şi complicată, o mare parte a efortului de proiectare este destinat rafinării şi întreţinerii specificării. Specificarea iniţială este adaptată în vederea co-sintezei, sunt estimaţi timpii de execuţie ai activităţilor ei şi sunt generate grafurile dependenţelor de date şi de control. Pentru aceste grafuri sunt propuse diferite soluţii de arhitecturi, le sunt asignate nodurile la resurse şi sunt estimaţi timpii globali de execuţie prin planificare automată.

Capitolul 8 prezintă concluziile noastre finale şi recapitulează concluziile desprinse pentru fiecare capitol în parte. Teza se încheie prin discutarea aspectelor ce pot fi subiecte de continuare a acestei munci.

Capitolul 2

Partiționarea hardware/software a specificărilor de sistem

Rezumat

Acest capitol prezintă două euristici pentru partiționarea automată hardware/software a specificărilor de sistem. Partiționarea are loc la o granularitate ridicată, corespunzătoare blocurilor de instrucțiuni, cicluri, subprograme şi procese. Obiectivele partiționării sunt acelea de a optimiza performanța implementării sistemului, atunci când costurile componetelor hardware şi software sunt păstrate sub anumite limite. Capitolul introduce metricile folosite pentru partiționare, precum şi funcția de cost care controlează evoluția procesului de partiționare spre obiectivul de proiectare dorit. Noi considerăm că performanțele implementării cresc prin minimizarea costului comunicărilor între partiția hardware şi cea software, cât şi prin mărirea gradului de paralelism în sistem (atât paralelismul între procese cât şi în interiorul fiecărui proces). Noi am formulat problema partiționării hardware/software ca şi o problemă de partiționare a unui graf şi am propus două euristici: una bazată pe simulated annealing, iar a doua pe tabu search. Superioritatea partiționării prin tabu search față de cea prin simulated annealing este justificată prin numeroase experimente şi partiționarea a două exemple inspirate din realitate.

Ideile şi rezultatele conținute în acest capitol au fost prezentate în [EP94][EP96] [EP97].

2.1 Introducere

Partiționarea hardware/software stabileşte ce părți din funcționalitatea unui sistem sunt plasate în hardware şi care în software. Partiționarea hardware/software este una din etapele cele mai importante din ciclul de proiectare al unui sistem, pentru că ea are un impact determinant atât asupra performanțelor ce se obțin pentru implementare, cât şi asupra costului ei.

Când se partiționează sisteme mici a căror funcționalitate este bine cunoscută şi înțeleasă, numărul alternativelor de realizare a celor două partiții este mic. În aceste situații, maparea funcționalității sistemului pe componentele hardware şi software poate fi făcută prin metode de "bun simț", bazate pe experiența şi ingeniozitatea

proiectantului [BC96]. Aspectele ce rămân subiecte de cercetare sunt: sinteza componentelor hardware și software, generarea interfețelor și co-simularea.

Problema partiționării hardware/software devine foarte complicată, dacă sistemul proiectat este mare, având o funcționalitate complexă, exprimată printr-un număr mare de procese ce interacționează între ele. În acest caz numărul alternativelor de construire a partițiilor hardware și software este foarte mare, iar impactul acestora asupra calității și costului final al implementării nu poate fi studiat în absența unui instrument CAD [GV95]. Literatura de specialitate discută diferite metode de partiționare hardware/software, care în general folosesc euristici performante, funcții de cost și estimări cât mai precise. În acest context, un instrument pentru partiționarea automată hardware/software este o componentă de bază pentru orice mediu de co-sinteză [BS96].

Cercetările în domeniu au pornit de la abordări diferite pentru problema partiționării. Ele se deosebesc prin ipotezele de lucru folosite, granularitatea la care se face partiționarea, gradul de automatizare al procesului de partiționare, funcția de cost folosită pentru ghidarea partiționării și algoritmul aplicat. [GM93][EH93][BR94] [OH94][VG94][AT95][NM96] partiționează automat specificările de sistem, iar [AS93][EF94][IJ95][CO95] o fac manual. [GM93][EH93][BR94] consideră că partiționarea trebuie făcută la o granularitate de lucru fină, iar [AS93][KL94][VG94] [AT95][YW95] folosesc o granularitate mare.

Euristicile bazate pe *îmbunătățiri succesive* (*iterative improvement algorithms*) sunt foarte populari în soluționarea partiționării hardware/software. Aceste euristici sunt astfel proiectate încât să evite blocarea lor în minime locale și să exploreze pe cât posibil întregul spațiu al soluțiilor. Majoritatea tehnicilor de partiționare sunt fundamentate pe simulated annealing [AT95][EH93][PK93]. Acest lucru este explicat de faptul că simulated annealing este o euristică simplă, ușor de implementat și care poate fi folosită pentru probleme de diferite naturi. În schimb, dezavantajele principale ale acestui algoritm provin din faptul că timpul său de execuție este relativ mare și necesită experimente foarte laborioase pentru adaptarea sa anumitor cazuri.

[VG94] propune un algoritm de partiționare, care combină o euristică de tip *hill climbing* cu o metodă de căutare binară. Obiectivul urmărit prin partiționare este minimizarea costului final al implementării și satisfacerea constrângerilor de performanță. Acest obiectiv este diferit de cel al proiectării noastre, care urmărește să maximizeze performanțele implementării finale pentru anumite limite de cost predefinite. Strategia de partiționare introdusă în [KL94] combină un algoritm de tip greedy cu un ciclu exterior, care consideră anumite măsuri globale. Această abordare, în mod similar cu [OH94][YW95], consideră că pentru fiecare task sunt cunoscuți timpii exacți de execuție ai implementării lor în hardware și în software și deasemenea timpii de comunicare sunt disponibili. Aceste cerințe implică constângeri puternice, pe care specificările de intrare trebuie să le satisfacă. Metoda noastră nu implică satisfacerea acestor constrângeri, motiv pentru care aria ei de aplicabilitate este mai largă. [NM96][BE96] formulează partiționarea hardware/software ca pe o problemă de programare liniară.

Abordarea din [AT95] este similară cu a noastră în sensul că obiectivul urmărit este minimizarea performaței (exprimată ca şi timp de execuție al implementării), cu satisfacerea anumitor limite pentru costurile componetei hardware şi ale celei software. Estimările folosite de autori, consideră că componenta software este formată dintr-un singur proces, planificat static.

Mediul nostru de proiectare acceptă specificări de intrare, care nu cuprind nici un fel de informații legate de implementare. Implementarea finală a unei aplicații trebuie să ofere performanțe maxime (formulate ca viteză de execuție) pentru anumite constrângeri de cost. Din punctul nostru de vedere, acest obiectiv este atins prin partiționare automată la o granularitate mare (blocuri de instrucțiuni, cicluri, subprograme, procese). Pentru că la acest nivel, estimările costurilor şi ale performanțelor sunt în mod firesc aproximative, metodologia noastră de proiectare permite repartiționări ulterioare, prin mutarea funcționalității necesare între cele două componenete [SP94].

Strategia noastră de partiționare foloseşte valori colectate prin simulare (*profiling*), analiză statică a specificării şi estimări ale costului. Factorii pe care îi considerăm majori din punct de vedere al partiționării specificării de sistem sunt minimizarea comunicărilor între componetele software şi hardware şi obținerea unui grad de paralelism cât mai ridicat al implementării finale. Rezultatele relativ modeste formulate în [EF94][OH94] se datorează mai ales comunicărilor între hardware şi software, a căror cost nu a fost minimizat.

Primul algoritm de partiționare implementat a fost cel bazat pe simulated annealing. Importante reduceri ale timpului de execuție pentru simulated annealing pot fi obținute adăugându-i îmbunătățiri specifice problemei. Apoi, am proiectat al doilea algoritm de partiționare folosind tabu search. Experimente relevante ne-au permis să concluzionăm că euristica cu tabu search are timpi de execuție mult mai scurți de cât cea care foloseşte tabu search.

Capitolul este structurat în şapte paragrafe. Paragraful 2 prezintă mediul nostru pentru co-sinteza hardware/software şi introduce ipotezele noastre legate de specificările de intrare partiționate şi de structura arhitecturii implementării finale. În continuare, discuția este axată pe folosirea limbajului VHDL pentru specificare de sistem. Paragraful 4 introduce etapele de partiționare, metricile şi funcția de cost folosită. Capitolul continuă detaliind folosirea lui simulated annealing şi tabu search pentru partiționarea automată, iar apoi evaluându-le performanțele şi comparându-i pe baza unor experimente largi. Capitolul se încheie formulând concluziile noastre.

2.2 Mediul pentru co-sinteza hardware/software

Figura 2.1 detaliază structura mediului nostru pentru co-sinteză hardware-software. Specificarea de intrare descrie funcționalitatea sistemului, însă fără a preciza care componete sunt implementate în hardware şi care în software şi în general, fără a include orice fel de detalii de implementare. Una din ipotezele importante de lucru este că specificarea de sistem este compusă din procese, care interacționează între ele prin mesaje transmise prin canale de comunicare. Deasemenea, presupunem că aceste

specificări sunt executabile, iar în urma simulărilor pentru seturi de test specifice se obţin inforamaţii statistice (*profiling information*).

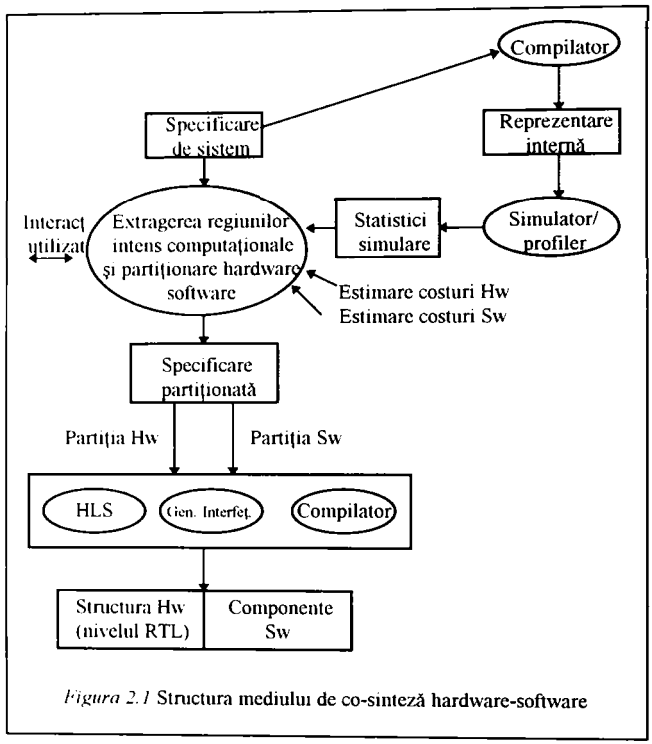


Figura 2.1 Structura mediului de co-sinteză hardware-software

Mediul pentru co-sinteză implementat de noi acceptă specificări VHDL de intrare, în care procesele interacţionează folosind mecanismul sincron bazat pe mesaje. Folosirea limbajului VHDL pentru specificare de sistem şi explicarea mecanismului de comunicare prin mesaje sunt detaliate în paragraful 2.2.

Mediul nostru pentru cosinteza hardware/software permite simularea, pariţionarea şi sinteza componetelor hardware şi software pentru o specificare VHDL de intrare.

După ce etapa de partiţionare s-a încheiat, componenta hardware este sintetizată folosind sistemul CAMAD de sinteză la nivel înalt [PK94]. Hardware-ul este interpretat ca şi un coprocesor specializat, care interacţionează cu software-ul produs prin translatarea reprezentării interne în C. În ceea ce priveşte arhitectura ţintă s-au făcut următoarele ipoteze:

- Arhitectura include un singur procesor programabil (microprocesor) pentru execuţia componentei software. Sistemul de operare este unul minim, care implementează în timpul execuţiei o strategie de planificare dinamică.
- Microprocesorul şi coprocesorul hardware lucrează în paralel. Arhitectura nu impune excluderea mutuală între hardware şi software.

- Performanța globală a implementării creşte prin reducerea volumului de comunicări între microprocesor (componenta software) şi coprocesor (componenta hardware).

Acest capitol detaliază partea de front-end a mediului de co-sinteză, care are ca şi scop extragerea în procese noi a regiunilor intens computaționale, şi partiționarea hardware-software.

2.3 Specificarea de sistem în VHDL

Sistemele digitale complexe sunt deobicei descrise ca un set de procese ce interacționează între ele. Un proces reprezintă o secvență de activități (instrucțiuni). VHDL a fost definit inițial ca limbaj pentru descrierea structurilor hardware, dar el posedă caracteristici care-l fac propriu şi pentru realizarea specificărilor de sistem. Principalele argumente în favoarea folosirii limbajului VHDL pentru specificări de sistem sunt:

- Oferă construcții pentru abstractizarea datelor şi a controlului.
- Permite realizarea de ierarhii structurale.
- Dispune de mecanisme pentru specificarea concurenței, a comunicărilor şi sincronizărilor între procese.
- Se pot introduce specificări de timp.
- Suportă o metodă de proiectare de tip top-down, ce se pretează atunci când sunt specificate nivele succesive de abstractizare ale procesului de proiectare.
- Specificările VHDL sunt executabile (simulabile).
- VHDL poate fi folosit atât pentru descrierea structurilor hardware, cât şi a celor software.

Deşi a fost definit pentru descrierea structurilor hardware, VHDL moşteneşte o serie de construcții de bază ale limbajului Ada, fapt ce creşte disponibilitatea sa pentru specificarea sistemelor software. Proiectantul de sistem poate specifica în VHDL funcționalitatea unor sisteme numerice complexe, fără a fi preocupat dacă componentele sistemului sunt executate în hardware sau în software. Specificările de sistem sunt executabile, iar prin simularea lor se verifică dacă funcționalitatea descrisă coincide cu cea dorită şi deasemenea, pot fi colectate date statistice, folosite apoi pentru paşii ulteriori ai proiectării. Specificările rafinate în urma paşilor de partiționare sau de sinteză pot fi descrise în VHDL, indiferent dacă componetele sunt implementate în hardware sau în software. Pentru folosirea limbajului VHDL în vederea co-sintezei, alături de argumentele legate de puterea sa de modelare, se pot formula şi argumente pragmatice. La ora aceasta, sunt disponibile un număr mare de instrumente CAD pentru simulare şi sinteză de nivel înalt, care acceptă specificări de intrare descrise în VHDL. Instrumentele disponibile pot fi integrate în medii noi de proiectare hardware/software.

Folosirea limbajului VHDL ca şi limbaj de intrare pentru un mediu de co-sinteză, implică ca specificarea rafinată prin etapele de proiectare să fie echivalentă semantic cu specificarea inițială şi să aibă comporatmentul la simulare al acesteia. Din această perspectivă, aspectul nevralgic îl reprezintă modul în care procesele interacționează

între ele. Standardul VHDL arată că sincronizarea şi comunicarea între procese se realizează prin instrucțiuni *wait* şi atribuiri de semnale, a căror semantică este definită în termenii ciclului de simulare pentru limbajul VHDL. Prin urmare, pentru a păstra echivalența semantică între specificare şi implementare, sinteza unei specificări VHDL în care procesele interacționează prin semnale, înseamnă sinteza ciclului de simulare [EK94]. Din păcate, mecanismul sincronizării şi al comunicării proceselor prin semnale şi instrucțiuni *wait* este unul de nivel coborât, care complică studiul comunicărilor între procese şi fac ca sinteza şi partiționarea să fie ineficiente şi greoaie.

Folosind un mecanism de nivel ridicat pentru comunicarea între procese, studiul proceselor şi al interacțiunii între ele este mult simplificat. Deasemenea, un astfel de mecanism ar putea fi implementat eficient atât în hardware, cât şi în software. În [EK93][EK94] este propus un model de comunicare între procesele VHDL, folosit pentru realizarea descrierilor de sistem, care sunt destinate a fi sintetizate în hardware. Acest model poate fi folosit cu succes şi pentru co-sinteza hardware/software. Potrivit acestui model, specificarea de sistem este compusă din procese care interacționează sincronizat, prin mesaje folosind primitivele predefinite *send* şi *receive*. Semnalele VHDL reprezintă canale de comunicare. Operațiunea send atribuie o valoare unui semnal, iar receptarea de către un proces a unei valori transmise printr-un semnal se face prin receive.

Procesele ce comunică prin mecanismul send-receive sunt slab cuplate. Ele pot fi implementate fără a cuprinde sincronizările puternice, implicate de ciclul de simulare. Sinteza unor sincronizări strânse este foarte ineficientă, în special la granița între hardware şi software [EK94]. Folosind primitivele send-receive, interfețele între procese pot stabilite şi modificate uşor, odată cu crearea proceselor noi sau cu mutarea funcționalității între procese pe parcursul partiționării.

Ca şi încheiere putem formula următoarea concluzie: specificările VHDL corespunzătoare acestui model de comunicare între procese pot fi simulate, partiționate, compilate în alte forme de reprezentare şi sintetizate, păstrând în permanență echivalența semantică cu specificarea inițială de sistem.

2.4 Etapele partitionării hardware-software. Funcția de cost.

Algoritmul de partiționare porneşte de la o specificare inițială a sistemului, descrisă ca un set de procese ce comunică între ele prin canale de comunicație. Rezultatul partiționării cuprinde două mulțimi de procese ce interacționează între ele: prima mulțime este formată din procesele menite a fi implementate în hardware, iar a doua cuprinde procesele destinate a fi executate în software. Scopul principal al partiționării hardware/software este maximizarea performanței implementării sistemului, unde performanța este definită ca fiind viteza de execuție a implementării. Euristica de partiționare distribuie funcționalitatea sistemului între componetele hardware şi software, luând în considerare costul comunicărilor necesare şi gradul de paralelism al implementării.

Din punctul nostru de vedere, creşterea vitezei de execuţie a implementării finale depinde de următoarele trei elemente principale:
1) Identificarea *regiunilor de bază* (procese, subprograme, cicluri, blocuri de instrucţiuni) unde este petrecută cea mai mare parte a timpului total de execuţie al sistemului. Aceste regiuni sunt implementate cu precădere în hardware.
2) Minimizarea comunicărilor între componenta hardware şi cea software.
3) Obţinerea unui grad ridicat de paralelism pentru implementarea finală a sistemului.
Algoritmul nostru de partiţionare urmăreste paralelismul sistemului pe trei nivele:

- Paralelismul intern al proceselor hardware. Etapa sintezei de nivel înalt planifică operaţiuni în paralel, în limita numărului unităţilor funcţionale disponibile.
- Paralelismul existent între procesele atribuite componetei hardware.
- Paralelismul ce apare între coprocesorul hardware şi microprocesorul destinat execuţiei proceselor software.

Informaţia folosită pentru partiţionare este obţinută prin simulare, analiza statică a specificărilor de sistem şi estimări ale costului implementării sistemului. În permanenţă, proiectantul poate interacţiona cu instrumentul CAD pentru a ghida strategiile majore de proiecatre. Datele statistice legate de simulare sunt obţinute prin compilarea specificării sistemului şi execuţia lui pentru stimuli de intrare tipici. În urma acestor execuţii, instrumentul CAD colectează două categorii de date, necesare partiţionării:

- *Complexitatea computaţională* a unei regiuni de bază (*CL*) măsoară volumul de calcule executate de acea regiune, luând în considerare toate activările ei de pe parcursul etapei de simulare. Complexitatea computaţională exprimă numărul total al operaţiilor executate şi care aparţin acelei regiuni. Fiecare operaţie este ponderată printr-un coeficient ce exprimă complexitatea ei relativă la alte operaţii.

  $$CL_i = \sum N\_act_j \times \Phi_{opj},$$

  unde suma este calculată pentru toate $op_j \in BR_i$. $[AD1]N\_act_j$ este numărul de execuţii ale operaţiei op_j ce aparţine regiunii de bază BR_i, iar Φ_{opj} este ponderea asociată lui op_j. *Complexitatea computaţională relativă* (*RCL*) Pentru un bloc de instrucţiuni, ciclu sau subprogram este raportul între complexitatea computaţională a regiunii în cauză şi complexitatea computaţională a procesului căruia îi aparţine. Complexitatea computaţională relativă a unui proces este raportul între complexitatea computaţională a procesului, respectiv cea a întregului sistem.
- *Intensitatea comunicărilor* unui canal între două procese (*CI*) este definită ca fiind numărul total de operaţii *send* care folosesc acel canal.

2.5.1 Etapele partitionării hardware-software

Figura 2.2 prezintă cei patru paşi ai partiţionării:

1) *Extragerea* blocurilor de instrucţiuni, a ciclurilor şi a subprogramelor şi încapsularea lor în procese noi. Acest pas examinează individual fiecare proces şi identifică regiunile critice din punct de vedere al performanţei (viteză de execuţie). Aceste regiuni "consumă" cea mai mare parte a timpului total de execuţie a sistemului şi sunt caracterizate prin CL-uri mari. În general aceste regiuni sunt cicluri sau subprograme, fără ca aceasta să fie însă o regulă. Proiectantul decide identificarea şi extragerea

regiunilor intens computaționale și astfel fixează granularitatea de lucru pentru următoarele etape ale partiționării. În acest scop, el poate decide între următoarele două strategii:

- Să identifice regiuni menite a fi extrase în procese noi, pe care apoi le atribuie componentei hardware sau software. De exemplu, dacă specificarea cuprinde o secvență de instrucțiuni care trebuie să satisfacă o constrângere de timp realizabilă doar implementând-o în hardware, atunci proiectantul decide că secvența în cauză trebuie plasată în hardware.
- Impunând două limite inferioare folosite pentru ghidarea procesului de extragere:
  - o limită X, astfel că doar procesele cu RCL > X sunt examinate pentru extragerea de regiuni de bază aparținându-le lor.
  - limita Y, astfel că doar blocurile de instrucțiuni, ciclurile și subprogramele cu RCL > Y sunt candidate pentru etapa de extragere.

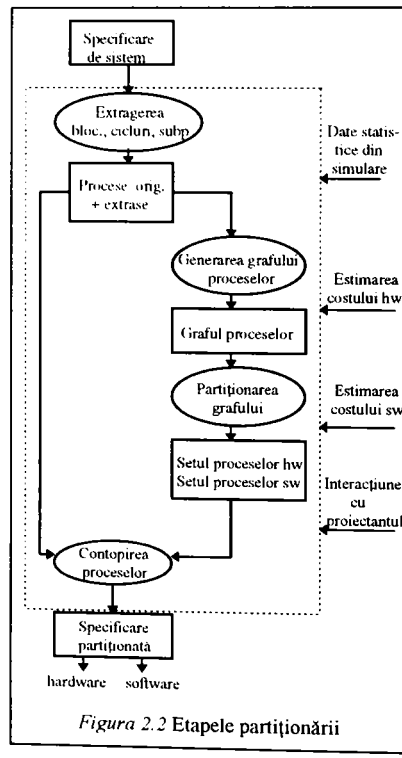


Figura 2.2 Etapele partiționării

Examinarea regiunilor candidate pentru extragere se face în manieră bottom-up, începându-se cu regiunile de bază interioare, neconținute în alte regiuni. Dacă RCL-ul regiunii este mai mic ca și Y-ul fixat de proiectant, atunci căutarea continuă pentru regiunea de bază care o cuprinde. Dacă însă regiunea examinată are RCL-ul său superior lui Y, atunci ea este extrasă într-un proces distinct având funcționalitatea regiunii. Bineânțeles, proiectantul poate inhiba anumite decizii ale instrumentului CAD privind extragerea unor regiuni de bază în procese noi.

2) *Generarea grafului proceselor*. Pasul doi al algoritmului de partiționare construiește o structură de date internă numită *graful proceselor*. Acest pas este analizat în detaliu în următoarea secțiune.

3) *Partiționarea grafului proceselor*. Acest pas formulează problema partiționării hardware-software ca și o problema partiționării grafului proceselor. Paragraful 4 al acestui capitol discută doi algoritmi folosiți de noi pentru partiționarea grafului și compară performanțele lor.

4) *Contopirea proceselor*. Primul pas extrage unul sau mai multe *procese fiu* dintr-un *proces părinte*. Dacă în urma partiționării grafului proceselor, unele din procesele fiu aparțin aceleiași partiții ca și procesul părinte din care au fost extrase, atunci opțional ele pot fi contopite la loc.

Încheiem această secțiune discutând maniera în care o regiune de bază este extrasă într-un proces nou. Discuția noastră este particularizată pentru limbajul VHDL pe care noi l-am folosit pentru specificare de sistem, însă în mod similar, ea poate avea loc pentru orice limbaj pentru specificare de sistem. Discuția reflectă maniera în care procesul extras comunică cu părintele său, astfel încât se respectă semantica procesului inițial.

Comunicarea între procesul părinte și cel extras se bazează pe mecanismul send-receive. Pentru că sincronizarea prin send-recive implică doar procesele participante la comunicare, înseamnă că semantica limbajului VHDL se păstrează [EK94]. Subprogramele candidate pentru extragere sunt tratate diferit, după cum ele sunt apelate dintr-un singur proces sau din mai multe procese. Subprogramele apelate din mai multe procese sunt extrase automat în procese distincte, ceea ce rezolvă implicit și aspectul protecției subprogramelor comune din punct de vedere al sintezei automate [EK93]. Subprogramele apelate dintr-un singur proces sunt parte integrantă a procesului apelant, excepție făcând situațiile când ele sunt extrase explicit prin pasul de extragere.

Exemplul următor detaliază extragerea unui ciclu, respectiv a unui subprogram din următoarele procese:

```
P1: process                    P2: process
...                            ...
LOOP_1:
  while x < k loop             procedure p (a: in integer;
                                b: out integer) is
  ...                          begin
  x := c + k;                        ...
  ...                                b := ... a ... ;
  end loop LOOP_1;                   ...
  ...                          end p;
end process P1;                begin
                                     ...
                                     p (7, z);
                                     ...
                               end process P2;
```

Dacă RCL-ul ciclului, respectiv a subprogramului este mai mare ca și limita Y specificată de proiectant, atunci ciclul `LOOP_1` și subprogamul `p` sunt extrase în doua

procese noi. Acestea sunt denumite P1_LOOP1 și respectiv P2_PROC_p. Pentru cele două procese extrase sunt necesare canale de comunicare înspre și dinspre ele, care sunt conforme cu dependențele de date ce apar între ele și procesele părinte. În exemplul nostru, comunicarea între părintele P1 și P1_LOOP_p1 se face prin semnalele s_P1_c, s_P1_k, s_P1_x_to și s_P1_x_from, iar cea între P2 și P2_PROC_p prin s_P2_a și s_P2_b.

În urma pasului de extragere, codul VHDL care reflectă noua situație este:

```
signal s_P1_c, s_P1_k, s_P1_x_to, s_P1_x_from,
       s_P2_a, s_P2_b: integer;

P1:process
...
send (s_P1_c, c, s_P1_x_to,
   x, s_P1_k, k);
... -- paralelism adițional
receive (s_P1_x_from);
x := s_P1_x_from;
...
end process P1;

P1_LOOP_1: process
      variable x: integer;
begin
      receive (s_P1_c,
       s_P1_x_to, s_P1_k);
      x := s_P1_x_to;
      LOOP_1:while
            x < s_P1_k loop

            ...
            x := s_P1_c +
                 s_P1_k;
            ...
      end loop LOOP_1;
      send (s_P1_x_from, x);
end process P1_LOOP_1;

P2: process
...
send (s_P2_a, 7);

... -- paralelism adițional
receive (s_P2_b);
z := s_P2_b;
...
end process P2;

P2_PROC_p: process
      variable b: integer;
begin
      receive (s_P2_a);
      ...
      b := ... s_P2_a ...

      ...
      send (s_P2_b, b);
end process P2_PROC_p;
```

Prin încapsularea anumitor regiuni de bază în procese separate, poate avea loc o creștere a gradului de paralelism a procesului părinte. Funcție de dependențele de date existente, anumite instrucțiuni ce urmează secvenței extrase sunt mutate între instrucțiunile send și receive necesare sincronizării cu procesul extras.

Decizia finală dacă procesele extrase sunt păstrate ca și module separate sau nu este luată în pasul partiționării grafului proceselor. Deși nu este considerat pe parcursul etapei de extragere, intensitatea comunicărilor între procesul părinte și fiul său este unul din criteriile importante luate în calcul de către algoritmul de partiționare.

2.5.2 Graful Proceselor

Mulțimea proceselor ce compun o specificare este folosită în pasul de partiționare. Unele procese sunt definite de proiectant prin specificarea inițială a sistemului, iar altele au fost create în urma pasului de extragere a regiunilor de bază. Complexitățile

computaționale ale proceselor extrase, respectiv intensitățile comunicărilor pentru noile canale de comunicare, sunt recalculate automat pe parcursul etapei de extragere. Structura de date folosită de algoritmul de partiționare hardware/software este *graful proceselor*. Nodurile acestui graf corespund proceselor din specificarea rezultată în urma etapei de extragere, iar două noduri sunt conectate printr-un arc dacă și numai dacă există cel puțin un canal de comunicare între procesele corespunzătoare.

Algoritmul de partiționare folosește ponderile atribuite fiecărui nod și arc din graf. Ponderea unui nod reflectă măsura în care procesul reprezentat de el este potrivit unei implementări în hardware. Ponderea unui arc depinde de "intensitatea" comunicărilor între procesele nodurilor conectate prin el, precum și numărul sincronizărilor ce apar între ele. Ponderile sunt stabilite pe baza informațiilor statistice colectate prin simularea sistemului (CL, RCL și CI), precum și a informațiilor colectate prin analiza statică a sistemului sau a reprezentării interne, obținute în urma compilării sale. Analiza statică a sistemului furnizează următoarele mărimi:

- Nr_op_i este numărul de operații în graful dependențelor de date pentru procesul i.
- $Nr\_kind\_op_i$ reprezintă numărul felurilor distincte de operații, cuprinse în procesul i.
- L_path_i este lungimea drumului critic (calculat pe baza grafului dependențelor de date) pentru procesul i.

Ponderea asociată nodului i are două componente: prima, pe care o notăm $W1_i^N$, este egală cu CL-ul procesului respectiv. Componenta a doua este calculată prin următoarea formulă:

$$W2_i^N = M^{CL} \times K_i^{CL} + M^U \times K_i^U + M^P \times K_i^P - M^{SO} \times K_i^{SO}$$

unde termenii au următoarele semnificații:

- K_i^{CL} este egal cu RCL-ul procesului i și deci este o măsură a complexității computaționale a procesului reprezentat
- K_i^U reflectă gradul de uniformitate a operațiilor din procesul i. Cu cât procesul i conține operații de mai puține feluri, cu atât K_i^U este mai mare. Procese cu un K_i^U mare sunt încurajate în hardware. Formula pentru calcularea lui K_i^U este:

  $$K_i^U = [\text{AD2}] \frac{Nr\_op_i}{Nr\_kind\_op_i}$$

- K_i^P este o măsură a potențialului de paralelism intern al procesului i. K_i^P este obținut prin relația:

  $$K_i^P = \frac{Nr\_op_i}{L\_path_i}$$

- K_i^{SO} reflectă măsura în care procesul i se pretează unei implementări software. Formula pentru calcularea lui K_i^{SO} este:

  $$K_i^{SO} = \frac{\sum_{op_j \in SP_i} w_{opj}}{nr\_op_i}$$

SP_i este mulțimea operațiilor din procesul i, care ar trebui implementate cu precădere în software. De exemplu, astfel de operații ar fi: operații în virgulă flotantă, operații pentru lucrul cu fișiere, apeluri recursive de subprograme, operații cu pointeri, etc. Mărimea w_{opj} este atribuită operației op_j, astfel încât să reflecte măsura în care op_j este

mai potrivită unei implementări software, de cât uneia hardware. Un w_{op} ridicat implică realizarea operaţiei *op* în software, indiferent de alte criterii de proiectare.

Importanţa parametrilor K^{CL}, K^{C}, K^{P} şi K^{SO} pentru procesul de partiţionare hardware-software este stabilită prin coeficienţii M^{CL}, M^{C}, M^{P} şi M^{SO}, a căror valoare este stabilită de proiectant

Ponderea asociată unui arc are deasemenea două componente. Dacă arcul conectează nodurile *i* şi *j*, atunci mărimea ponderii sale trebuie să arate volumul comunicărilor între *i* şi *j*. Prima componentă depinde de cantitatea totală de date transmise între nodurile *i* şi *j*, iar a doua de numărul de interacţiuni între *i* şi *j*, exprimat prin numărul de sincronizări necsare între *i* şi *j*.

$$W1^{E}_{ij} = \sum_{c_k \in Ch_{ij}} wd_{ck} \times CI_{ck} \qquad W2^{E}_{ij} = \sum_{C_k \in Ch_{ij}} CI_{ck}$$

Ch_{ij} este mulţimea canalelor folosite pentru comunicarea între procesele *i* şi *j*, wd_{ck} este lăţimea canalului c_k (exprimat prin numărul biţilor transportaţi), iar CI_{ck} este intensitatea comunicărilor pentru c_k.

2.5.3 Functia de Cost şi Constrângerile de partiţionare

Folosind graful proceselor generat pentru o specificare de intrare, partiţionarea hardware/software este formulată ca problema partiţionării acestui graf. Ghidarea activităţii de partiţionare spre obiectivul de proiectare urmărit este controlată printr-o funcţie de cost. Aceasta este o relaţie între ponderile nodurilor şi arcelor din graf şi trebuie să aibă o astfel de formă, încât să poată fi uşor actualizată pe parcursul algoritmului de partiţionare.

Euristicile noastre de partiţionare sunt controlate de următoarea funcţie de cost, a cărei valoare este minimizată pe parcursul execuţiei algoritmilor:

$$C(Hw, Sw) = Q1 \times \sum_{(ij)\in cut} W1^{E}_{ij} + Q2 \times \frac{\sum_{(i)\in Hw} \frac{\sum_{\exists(ij)} W2^{E}_{ij}}{W1^{N}_{i}}}{N_H} + Q3 \times \left(\frac{\sum_{(i)\in Hw} W2^{N}_{i}}{N_H} - \frac{\sum_{(i)\in Sw} W2^{N}_{i}}{N_S} \right)$$

Hw şi *Sw* sunt mulţimile ce reprezintă partiţiile hardware şi software. N_H şi N_S sunt cardinalităţile celor două mulţimi, *cut* este mulţimea arcelor ce conectează noduri în cele două partiţii, *(ij)* indică arcul ce leagă nodurile *i* şi *j*, iar *(i)* reprezintă nodul *i* din graf.

Îndeplinirea obiectivelor de proiectare formulate la începutul secțiunii 2.3 corespunde minimizării funcției de cost. Obiectivele de proiectare sunt modelate prin cei trei termeni ai funcției de cost:

- Primul termen descrie volumul comunicărilor între partiția hardware şi cea software. Dacă reducem acest termen al funcției de cost, atunci scade volumul total al comunicărilor între componenta hardware şi cea software, iar acest fapt este unul din obiectivele majore ale proiectării. În acelaşi timp, pentru că scade gradul de interacțiune între cele două partiții, va creşte paralelismul între procesele executate în hardware şi cele din software.
- Al doilea termen încurajează maparea în hardware a proceselor având un grad scăzut de interacțiune cu restul sistemului. Pentru a evita migrarea în hardware a unor procese simple din punct de vedere al complexității lor computaționale, volumul interacțiunilor unui proces este raportat la complexitatea sa computațională. Dacă există alocate suficiente resurse fizice, această strategie favorizează un paralelism real ridicat între procesele executate în hardware. Pentru procesul i, termenul $\left(\sum_{\exists(ij)} W2_{ij}^{E}\right) \Big/ W1_{ij}^{E}$ reprezintă raportul între numărul interacțiunilor în care el este implicat şi complexitatea sa computațională. Termenul funcției de cost este media acestor valori calculată pentru toate nodurile partiției hardware.
- Termenul al treilea din funcția de cost favorizează o diferență mare între ponderile medii ale nodurilor în cele două partiții şi astfel, trimite nodurile cu pondere mare în hardware, iar pe cele cu pondere mică în software. Acest obiectiv este unul de bază pentru metoda noastră de proiectare, pentru că "pune" regiunile intens computaționale în hardware , astfel că paralelismul lor intern poate fi exploatat.

Criteriile combinate în funcția de cost nu sunt ortogonale şi uneori ele se influențează reciproc. De exemplu, deplasând în hardware un nod cu pondere mare, putem creşte volumul comunicărilor între cele două partiții. Proiectantul are posibilitatea de a controla interferențele ce apar între termenii funcției de cost, prin factorii Q_1, Q_2 şi Q_3, a căror valori hotăresc importanța celor trei metrici în ansamblul procesului de proiectare.

Minimizarea funcției de cost trebuie să respecte constrângerile de proiectare impuse. De aceea, euristica trebuie să stabilească partiția care minimizează funcția de cost $C(Hw, Sw)$, iar costurile componentelor hardware şi software se situează sub anumite limite stabilite:

$$\sum_{(i)\in Hw} H\_\cos t_i \le Max^H \text{ şi } \sum_{(i)\in Sw} S\_\cos t_i \le Max^S$$

Proiectantul poate fixa anumite procese în una din cele două componente. Acest lucru îl realizează prin ponderile asociate nodurilor. Nodurile cu ponderea mai mică de cât o limită fixată sunt plasate şi fixate automat în software, iar cele având ponderea mai mare ca şi o altă limită stabilită sunt implementate automat în hardware.

$$W2_i^N \ge Lim1 \Rightarrow (i) \in Hw; \qquad W2_i^N \le Lim2 \Rightarrow (i) \in Sw;$$

Pentru fiecare proces al specificării trebuie estimate costurile implementării hardware, respectiv software. Estimările au loc înaintea pasul de partiționare. Costul hardware este apreciat folosind sistemul de sinteză de nivel înalt CAMAD, care exprimă acest cost ca și aria necesară implementării hardware. Costul software este reprezentat prin dimensiunea memoriei necesare și poate fi stabilit pentru fiecare proces fie prin compilarea VHDL în C [GR95] sau ETPN în C [DE96].

2.6 Algoritmii de partiționare. Rezultate experimentale

Pasul trei al patiționării hardware/software este cel care bipartiționează graful proceselor în două mulțimimi: una a proceselor destinate implementării hardware și una pentru procesele menite a fi executate în software. Felul în care algoritmul de partiționare atribuie nodurile grafului celor două mulțimi satisface constrângerile de proiectare impuse și minimizează funcția de cost.

Problema partiționării hardware/software formulată ca cea a partiționării unui graf, are un caracter NP-complet. Pentru a explora în mod eficient spațiul soluțiilor, trebuie proiectate euristici care să conveargă spre soluții optime sau situate în apropierea celor optime. Noi am implementat două euristici pentru partiționare: una este bazată pe simulated annealing (SA), iar a doua pe tabu search (TS). Ambele explorează la un moment dat o vecinătate, definită în anumite limite, dar la anumite nivele execută în mod controlat miscări pentru a evita blocarea lor în minime locale.

Acest paragraf prezintă cei doi algoritmi, evaluează performanțele lor și îi compară pe bază de rezultate experimentale extensive.

2.6.1 Evaluarea experimentală a algoritmilor de partiționare

Cei doi algoritmi de partiționare am fost evaluați experimental folosind *grafuri aleatoare* și *grafuri geometrice* [YC95], precum și grafurile rezultate prin compilarea specificărilor VHDL pentru două exemple complexe, din industrie.

Grafurile aleatoare sunt definite de două marimi caracteristice: numărul nodurilor și probabilitatea ca două noduri să fie conectate printr-un arc. Grafurile geometrice sunt caracterizate prin numărul și gradul mediu al nodurilor. Grafurile geometrice, spre deosebire de cele aleatoare, nu au nodurile conectate uniform, ci ele sunt grupate în clustere.

Ponderile și costurile nodurilor și arcelor din grafuri sunt fixate în timpul generării grafurilor. Experimentele noastre au folosit în total 32 de grafuri, 16 grafuri aleatoare și 16 geometrice. Pentru fiecare din cele patru variante, cu 20, 40, 100 și 400 de noduri, s-au produs câte 8 grafuri, 4 aleatoare și 4 geometrice. Caracteristicile celor 16 grafuri aleatoare sunt cuprinse în tabelul următor.

Numărul nodurilor	Probabilitate			
20	0.1	0.15	0.2	0.25
40	0.05	0.1	0.15	0.2
100	0.015	0.02	0.05	0.1
400	0.008	0.015	0.025	0.04

Tabelul 1. Caracteristici ale grafurilor aleatoare

Pentru fiecare din cele 4 dimensiuni, grafurile geometrice corespund gradelor medii de 0.85, 2, 4 şi 10. Experimentele parcurse de noi au avut în vedere următoarele două obiective principale:

- Adaptarea algoritmilor pentru fiecare dimensiune a grafurilor şi stabilirea parametrilor generici, astfel ca algoritmii de partiţionare să conveargă spre soluţiile optime într-un timp minim şi cu o probabilitate mare.
- Compararea eficienţei celor doi algoritmi.

În acest moment trebuie definită noţiunea de optim pentru contextul nostru. Pentru grafurile cu 20 de noduri, a fost posibil să prelucrăm întregul spaţiu al soluţiilor şi să obţinem *optimul real* pentru problema partiţionării hardware/software a unui graf. Acest optim l-am folosit apoi ca şi mărime de referinţă pentru compararea rezultatelor obţinute prin euristici. Pentru grafurile cu dimensiuni mai mari ca 20, timpul de explorare a întregului spaţiu al soluţiilor devine prohibitiv de mare, astfel că nu mai dispunem de optimul real al problemei. Într-o fază preliminară celei de experimentare a euristicilor, am executat atât SA cât şi TS folosind resurse de calcul foarte puternice şi cu o limită a timpului de execuţie foarte mare. SA a folosit o răcire foarte lentă, iar TS un număr mare de reporniri. Execuţiile au pornit de la diferite configuraţii de început, iar cea mai bună soluţie găsită (soluţia pentru care funcţia de cost este minimă) pe parcursul acestor execuţii, a fost considerată ca fiind optimul pentru restul experimentelor.

Experimentele cu SA au ridicat o dificultate suplimentară datorită caracterului aleator al acestei euristici. Aceeaşi implemetare a algoritmului, cu parametrii neschimbaţi poate produce pentru acelaşi graf rezultate diferite la execuţii diferite. Dacă pe parcursul experimentelor am constat că o anumită configuraţie a parametrilor lui SA găseşte pentru 100 de execuţii consecutive aceeaşi soluţie optimă pentru un graf, atunci ea produce cu o probabilitate suficient de mare optimul pentru acel graf.

Toate experimentele care vor fi prezentate în continuare, au fost executate pe staţii SUN SPARC 10.

2.6.2 Partitionarea cu Simulated Annealing

Simulated annealing selectează întotdeauna aleator o soluţie vecină celei curente şi o acceptă dacă ea o îmbunătăţeşte pe cea curentă. Euristica poate accepta cu o anumită probabilitate şi soluţii mai slabe de cât cea prezentă, unde probabilitatea depinde de măsura în care calitatea soluţiei este stricată şi de un parametru de control denumit *temperatură* (temperature) [KG83].

În continuare detaliem pseudocodul algoritmului de SA.

```
Pasul 1. Construieşte o configuraţie iniţială x^now = (Hw_0, Sw_0);
Pasul 2. Iniţializează temperatura T = TI;
Pasul 3. 3.1 for i = 1 to TL
            Generează aleator o soluţie învecinată, x' ∈ N(x^now);
            Calculează variaţia fucţiei de cost, ΔC = C(x') - C(x^now);
            if (ΔC ≤ 0)
               x^now = x';
            else
               Generează aleator, q = random (0, 1);
               if q < e^(-ΔC/T)
                  x^now = x';
         3.2 fixează temperatura nouă, T = α * T;
Pasul 4. if (criteriul de oprire nu este îndeplinit) goto Pasul 3;
Pasul 5. return soluţia pentru care funcţia de cost este minimă;
```

x indică o soluţie formată din două mulţimi, Hw şi Sw. x^{now} este soluţia curentă, iar $N(x^{now})$ este vecinătatea lui x^{now} în spaţiul soluţiilor.

Implementarea lui SA este completă odată cu fixarea lui TI - *temperatura iniţială* (initial temperature), TL - *lungimea temperaturii* (temperature length), α - *rata de răcire* (cooling ratio) şi a *criteriului de terminare*. Aceşti parametrii stabilesc *modul de răcire* (cooling schedule) şi au un impact determinant asupra calităţii partiţiilor care se obţin, respectiv a timpului de CPU necesar pentru găsirea soluţiilor. Tabela următoare prezintă valorile lui TI, TL şi α care au fost experimentate pentru grafurile cu diferite dimensiuni. Pentru fiecare caz sunt indicate două valori, SM corespunde lui SA care produce soluţiile noi prin *mişcări simple* (simple moves), iar IM este folosit pentru SA cu *mişcări îmbunătăţite* (improved moves). SA cu modurile de răcire definite de parametrii din tabelă stabileşte soluţiile optime pentru grafurile cu dimensiunile corespunzătoare.

numărul	TI		TL		α	
nodurilor	SM	IM	SM	IM	SM	IM
20	400	400	90	75	0.96	0.95
40	500	450	200	150	0.98	0.97
100	500	450	500	200	0.98	0.97
400	1400	1200	7500	2750	0.998	0.995

Tabelul 2. Metode de răcire

Criteriul de oprire al algoritmului presupune că sistemul este îngheţat, atunci când SA nu acceptă soluţii noi pentru trei temperaturi consecutive.

Construirea unei soluţii noi x', pornind de la cea curentă x^{now}, este dependentă de problema particulară, rezovată prin algoritmul de SA. Noi am implementat şi experimentat două strategii pentru generarea unor soluţii noi: *mişcările simple* şi *mişcările îmbunătăţite*. În cazul *mişcărilor simple*, un nod este selectat aleator şi apoi mutat în cealaltă partiţie. Configuraţia care rezultă în urma acestui pas devine noua soluţie x'. Dacă selecţia aleatoare a nodului mutat contravine unor constrângeri de proiectare, atunci pasul selecţiei aleatoare a unui nod pentru mutare este repetat. *Mişcările îmbunătăţite* urmăresc să accelereze convergenţa algoritmul de SA (prin urmare scăderea timpului în care optimul este găsit) mutând împreună cu nodul selectat

şi pe unii din vecinii săi direcţi (noduri care sunt conectate printr-un arc cu selectatul şi aparţin aceleiaşi partiţii ca şi cel selectat). Dacă are loc o îmbunătăţire a funcţiei de cost şi dacă nu se încalcă nici una din constrângerile de proiectare, atunci un vecin direct este mutat împreună cu nodul selectat. Această strategie încurajează mutarea între partiţii a unor grupuri de procese, mai de grabă de cât a proceselor individuale. Experimentele efectuate au pus în evidenţă un efect negativ ce însoţeşte această strategie şi anume, mişcarea repetată între partiţii a aceluiaşi grup sau a grupuri similare de noduri, fapt ce limitează spaţiul soluţiilor vizitat. Pentru a explora în mod eficient spaţiul soluţiilor, am combinat mutările unor grupuri de noduri cu mutările unor noduri singulare. Astfel, strategia mişcărilor îmbunătăţite a fost la rândul ei îmbunătăţită, deoarece nodurile sunt mutate în grup doar cu o anumită probabilitate. În urma experimentelor, această probabilitate a fost fixată la valoarea 0.75. Secvenţa de cod următoare indică construirea unei soluţii candidat prin startegia mişcărilor îmbunătăţite.

```
Pasul 1. Generează k = random (1, numărul_nodurilor_în_graf), până
         când prin mutarea nodului k nu se încalcă constrângerile
         de proiectare;
Pasul 2. Generează din x'' soluţia x', mutând nodul k;
Pasul 3. Generează q = random (0,1);
Pasul 4. if q ≥ p
            return x';
            exit;
Pasul 5. 5.1. for ∀ nod k',vecin direct ai lui k şi care aparţine
              aceleiaşi partiţii ca şi k
              if (mutând nodul k' nu se încalcă constângerile
              şi ΔC ≤ 0)
                 Generează din x'curent soluţia x' mutând nodul k';
         5.2 return x';
```

numărul nodurilor	timp CPU (s)		accelerare
	SM	IM	
20	0.28	0.23	22%
40	1.57	1.27	24%
100	7.88	2.33	238%
400	4036	769	425%

Tabelul 3. Timpii de partiţionare cu SA

Metoda mişcărilor îmbunătăţite este specifică problemei partiţionării hardware-software. Influenţa ei asupra metodei de răcire este indicată în tabelul 2. Scăderea lui TI, TL şi α reduce timpul de răcire şi automat timpul de partiţionare. Timpii de partiţionare şi accelerările obţinute prin mişcările îmbunătăţite sunt prezentate în tabelul 3 şi în figura 2.5. Timpii indicaţi sunt timpii medii de CPU, necesari execuţiei algoritmului de SA pentru toate grafurile de o anumită dimensiune, când SA funcţionează cu parametrii tabelului 2. Accelerarea este semnificativă chiar şi pentru grafuri mici, creşte odată cu numărul nodurilor şi devine peste 400% pentru grafuri cu 400 de noduri. Figurile 2.6 şi 2.7 corespund la grafuri cu 100 de noduri, iar figurile 2.8 şi 2.9 la cele cu 400 de noduri. Ele arată modul în care SA explorează spaţiul soluţiilor. Comparând curbele din figurile 2.6 şi 2.8, cu cele din figurile 2.7 şi 2.9, se observă o convergenţă mult mai rapidă a lui SA, pentru mişcarea îmbunătăţită faţă de cea simplă.

Pentru grafurile cu 400 de noduri, figurile 2.8 şi 2.9 compară calitatea medie a soluţiilor vizitate de SA cu mişcarea simplă şi de SA cu mişcarea îmbunătăţită. Euristica cu mişcarea îmbunătăţită găseşte mult mai rapid partiţia optimă, din cauză că parcurge un drum mult mai apropiat optimului de cât variantă simplă.

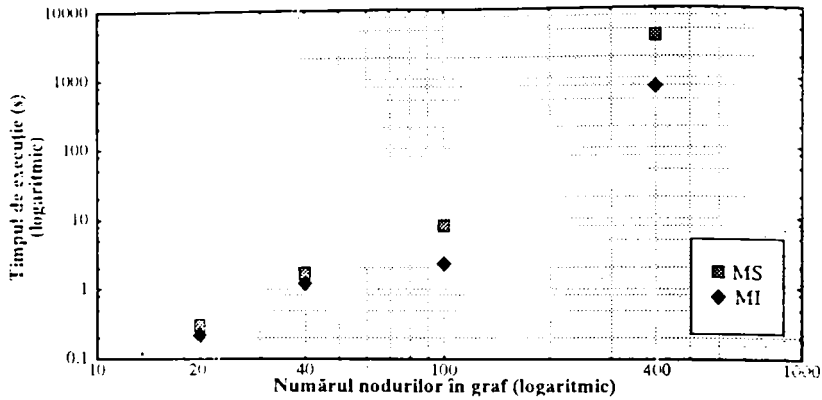


Figura 2.5 Partiţionarea cu SA: mutări simple (MS) şi mutări îmbunătăţite (MI)

2.5.3 Partitionarea cu Tabu Search

Tabu search acceptă în mod inteligent şi controlat soluţii mai slabe de cât cea curentă, în tentativa sa de a ieşi din minimele locale (*uphill moves*). Diferenţa ceasta este una de principiu între TS şi SA, acesta din urmă acceptând în mod aleator soluţiile mai slabe. Pentru selecţia soluţiilor calitativ mai proaste şi pentru convergenţa sa spre optimul global, TS foloseşte structuri de date pentru memorarea "istoriei" (secvenţei trecute) paşilor de căutare. Această istorie este fundamentală pentru selectarea următorului pas de căutare [GT93].

```
Pasul 1. Construieşte configuraţia iniţială x^now = (Hw_0, Sw_0);
Pasul 2. for (∀ soluţie x_k ∈ N (x^now))
             Calculează variaţia funcţiei de cost
             ΔC_k = C(x_k) - C(x^now);
Pasul 3. 3.1 for (∀ ΔC_k < 0, în ordinea crescătoarea a lui ΔC_k)
                 if (x_k nu este tabu sau nu este tabu aspirat)
                    x^now = x_k;
                    goto Pasul 4;
         3.2 for (∀ soluţie x_k ∈ N (x^now))
                 Calculează ΔC'_k = ΔC_k + penalty (x_k);
         3.3  or (∀ ΔC'_k în ordinea descrescătoare a lui ΔC'_k)
                 if (x_k nu este tabu)
                    x^now = x_k;
                    goto Pasul 4;
         3.4 Generează x^now executând cel mai vechi tabu;
Pasul 4. 4.1 if (numărul iteraţiilor trecute de când s-a găsit cea
                 mai bună soluţie < Nr_f_b)
                    goto Pasul 2;
```

```
        4.2 if (numărul repornirilor < Nr_r)
              Generează o configurație inițială x^new, considerând
              frecvențele;
              goto Pasul 2;
Pasul 5. return partiția ce corespunde la costul minim;
```

Figura 2.10 Algoritmul de tabu search

Memoria pe termen scurt şi cea pe *termen lung* (*short* and *long term memory*) sunt structurile de date de bază pentru algoritmul de TS. *Memoria pe termen scurt* stochează informaţii legate de paşii de căutare cei mai recenţi şi împiedică ciclarea algoritmului, atunci când în urma unei mutări se revine la o configuraţie vizitată recent. *Memoria pe termen lung* păstrează informaţii despre starea globală a algoritmului. Aceste informaţii se referă mai ales la frecvenţele de apariţie a anumitor evenimente şi sunt folosite pentru diversificarea (*diversification*) căutării în spaţiul de soluţii.

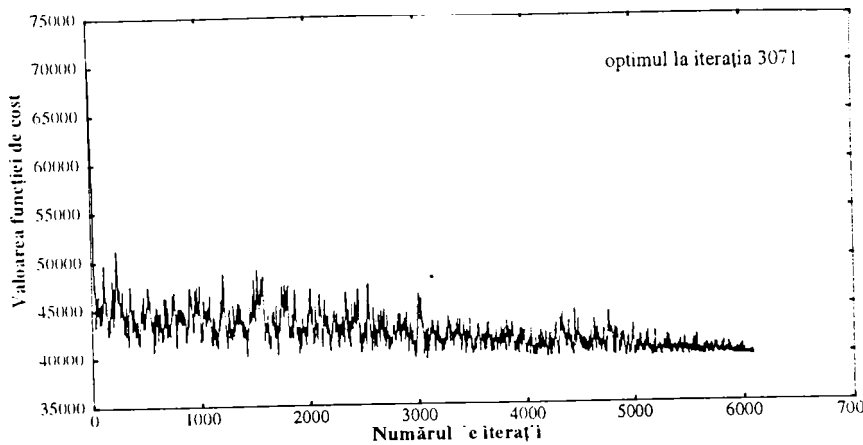


Figura 2.6 Variaţia funcţiei de cost pentru SA la 100 de noduri şi mişcări simple

Figura 2.10 introduce psudocodul algoritmului nostru de TS. Fiecare pas al algoritmului de TS încearcă să continue cu o soluţie, care să fie superioară celei curente. Dacă o astfel de soluţie nu există (sau este un tabu şi nu este aspirat), atunci îi sunt aplicate penalizări funcţie de frecvenţă funcţiei de cost şi este selectată cea mai bună mişcare ce nu este tabu. O astfel de mişcare poate fi una uphill. În ultima instanţă, se executată mişcarea, cea mai aproape de a-şi părăsi starea de tabu.

Soluţiile candidat x_k sunt obţinute mutând nodurile k din partiţia în se găsesc pentru soluţia curentă x^{now}, în cealaltă partiţie. Restricţia pe care aceste mişcări trebuie să le satisfacă, este să nu violeze constrângerile de proiectare impuse. *Lista tabu*-urilor păstrează secvenţa miscărilor inverse, pentru ultimele τ mişcări efectuate. Mutările din această listă sunt definite ca fiind tabu-uri. τ - lungimea listei tabu-urilor (*tabu tenure*) este un parametru de bază al algoritmului. Tabelul 2.4 conţine valorile optime pentru τ, aşa cum rezultată din experimentele noastre.

Numărul nodurilor	τ	*Nr_f_b*	*Nr_r*	*CPU_time (s)*
20	7	30	0	0.008
40	7	50	0	0.04
100	7	50	0	0.19
400	18	850	2	30.5

Tabelul 2.4 Parametrii și timpul CPU pentru TS

În anumite condiții, unele mișcări tabu pot fi totuși selectate pentru execuție (spunem că tabu-ul a fost aspirat - *tabu aspiration*). Un tabu este aspirat dacă acea mișcare produce îmbunătățirea căutării și nu determină o ciclare a procesului de căutare. Algoritmul nostru ignoră caracterul de tabu al unei mișcări, atunci când efectuând acea mișcare se obține cea mai bună soluție cunoscută până în acel moment.

Diversificarea algoritmului de TS se face folosind informația care arată câte iterații și-a petrecut fiecare nod în partiția hardware. Această informație este memorată în memoria pe termen lung. Implementarea noastră pentru TS folosește trei moduri de a realiza diversificarea:

- Când este generată o configurație nouă (figura 2.10), nodurile sunt ordonate conform unei funcții de cost penalizate, ce favorizează transferul nodurilor care au petrecut un număr mare de iterații în partiția curentă.

$$\Delta C'_k = \Delta C_k + \frac{\sum_i |\Delta C_i|}{Nr\_of\_nodes} \times pen(k), \text{ unde}$$

$$pen(k) = -C_H \times \frac{Node\_in\_Hw_k}{N_{iter}}, \text{ dacă } k \in Hw \text{ și}$$

$$pen(k) = -C_S \times \left(1 - \frac{Node\_in\_Hw_k}{N_{iter}}\right), \text{ dacă } k \in Sw$$

$Node\_in\_Hw_k$ este numărul de iterații pe care nodul k le-a petrecut în hardware. N_{iter} este numărul total de iterații. Experimental, pentru C_H și C_S s-au fixat valorile 0.4 și 0.15.

- O mișcare este tabu, dacă frecvența de apariție a nodului implicat în partiția curentă este mai mică de cât o limită fixată. O mișcare poate fi acceptată dacă:

$$\frac{Node\_in\_Hw_k}{N_{iter}} > T_H, \text{ dacă } k \in Hw \text{ și}$$

$$\left(1 - \frac{Node\_in\_Hw_k}{N_{iter}}\right) > T_S, \text{ dacă } k \in Sw$$

  Limitele folosite în experimente sunt $T_H = 0.2$ și $T_S = 0.4$.

- Un sistem este înghețat (*frozen*) atunci când au trecut un număr de *Nr_f_b* iterații de la acea care a găsit cea mai bună soluție. Pentru un sistem înghețat, o căutare nouă poate începe, pornind de la o soluție de start diferită de configurațiile întâlnite în căutare până în acel moment.

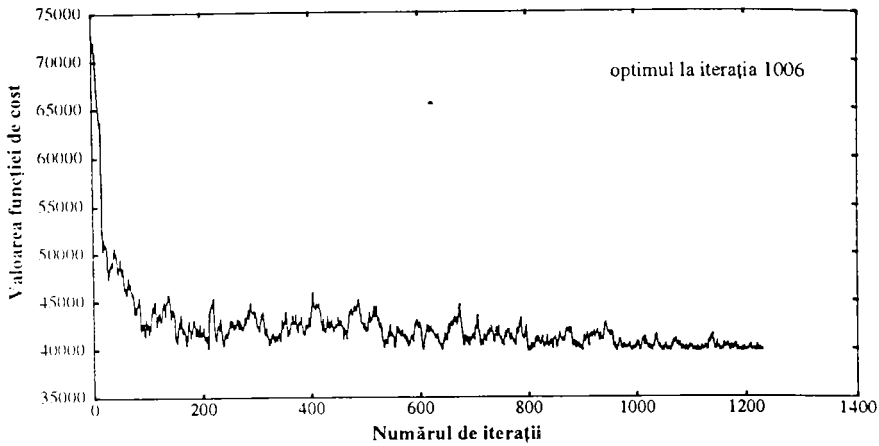


Figura 2.7 Variaţia funcţiei de cost pentru SA la 100 de noduri şi mişcări îmbunătăţite

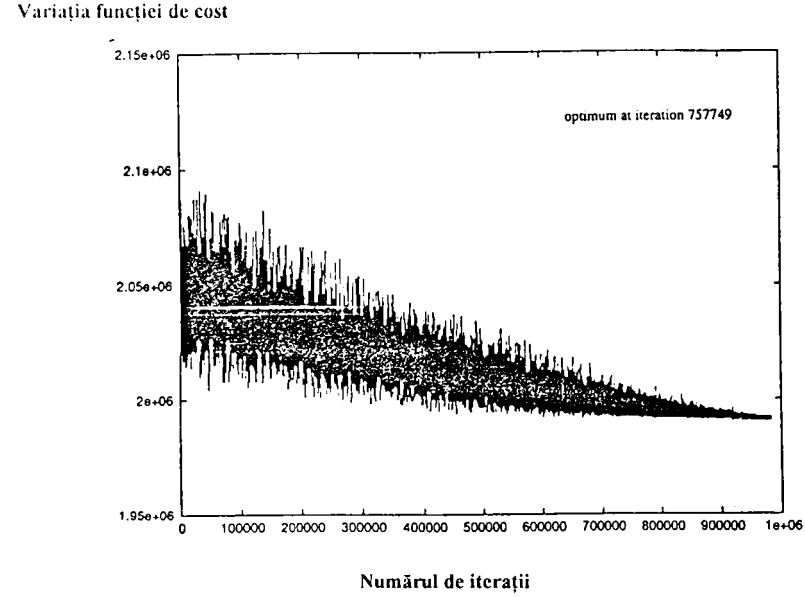


Figura 2.8 Variaţia funcţiei de cost pentru SA la 400 de noduri şi mişcări simple

Numărul total de iteraţii parcurse de algoritmul de TS depinde de parametrii *Nr_f_b* (stabileşte numărul de iteraţii consecutive fără îmbunătăţirea soluţiei cunoscute, după care sistemul este considerat îngheţat) şi *Nr_r* (numărul de reporniri ale căutării dintr-o configuraţie nouă). Tabelul 2.4 prezintă valorile minime pentru cei doi parametrii şi timpii de CPU corespunzători, pentru care TS găseşte partiţionarea optimă a tuturor

grafurilor având dimensiunea indicată. Timpii din tabel au fost calculaţi ca fiind media tuturor timpilor de partiţionare ce rezultă pentru grafurile cu acea dimensiune. Un aspect intersant observat în timpul experimentelor este acela că doar pentru grafurile cu 400 de noduri au fost necesare reporniri ale TS din soluţii de start noi.

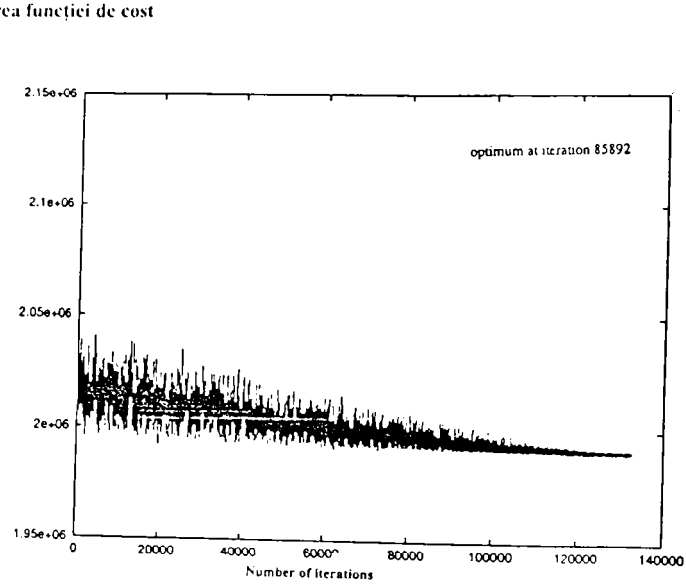


Figura 2.9 Variaţia funcţiei de cost pentru SA la 400 de noduri şi mişcări îmbunătăţite

Startegiile de căutare ale TS pentru grafuri cu 400 de noduri şi 100 de noduri apar în figurile 2.11 şi 2.12. Figura 2.12 arată că pentru grafuri cu 100 de noduri, TS converge foarte rapid spre optim, fără a fi nevoie de repornirea sa dintr-o configuraţie de start nouă. Figura 2.11 indică faptul că partiţionarea grafurilor cu 400 de noduri a apelat la diversificare, fiind necesare euristicii două reporniri din situaţii noi. Cele două poze suplimentare din figura 2.11, detaliază spaţiul din vecinătatea optimului. Algoritmul controlează procesul de căutare a optimului printr-o succesiune de diversificări, mişcări *uphill* şi mutări ce îmbunătăţesc funcţia de cost.

2.5.4 Partitionarea cu Simulated Annealing vs. Partitionarea cu Tabu Search

Experimentele prezentate în secţiunile precedente permit formularea următoarelor concluzii cu privire la folosirea SA şi TS pentru partiţionarea hardware-software:

- Atât SA cât şi TS oferă soluţii apropiate optimelor pentru problema partiţionării hardware-software.

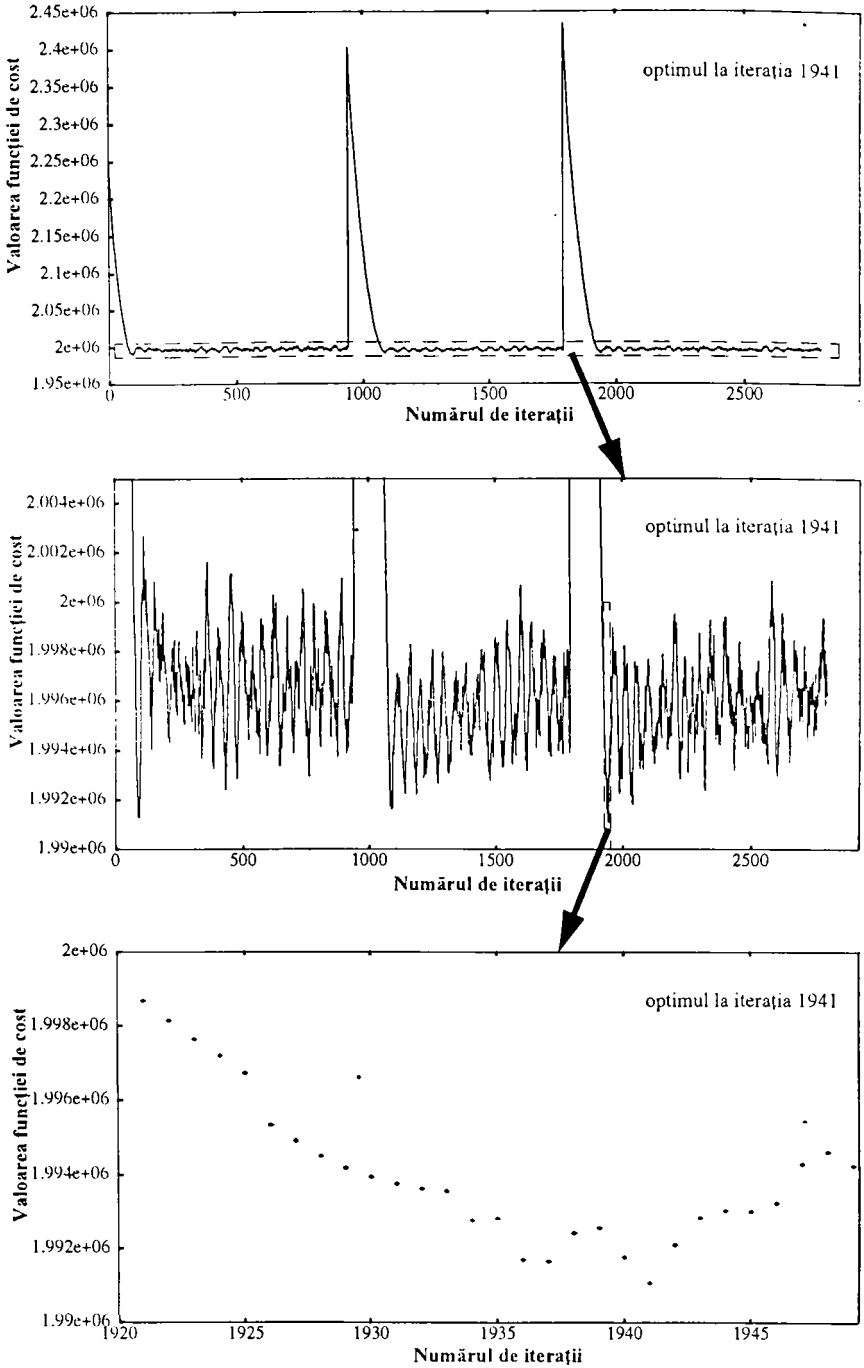


Figura 2.11 Variaţia funcţiei de cost pentru tabu search la 400 de noduri

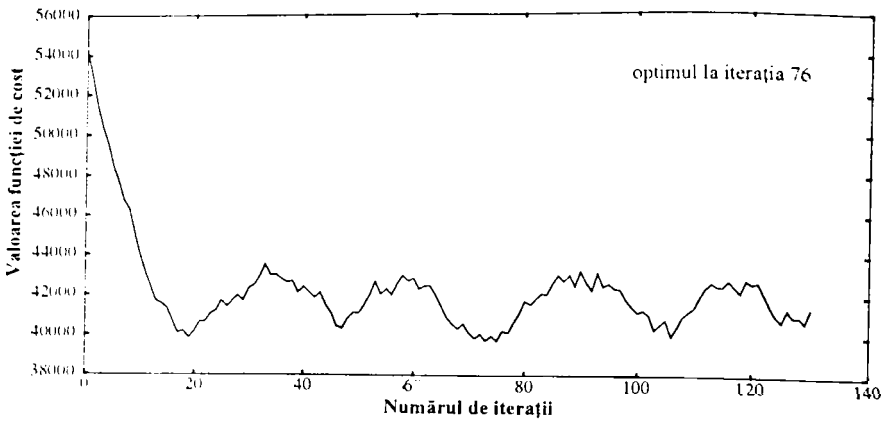


Figura 2.12 Variaţia funcţiei de cost pentru tabu search la 100 de noduri

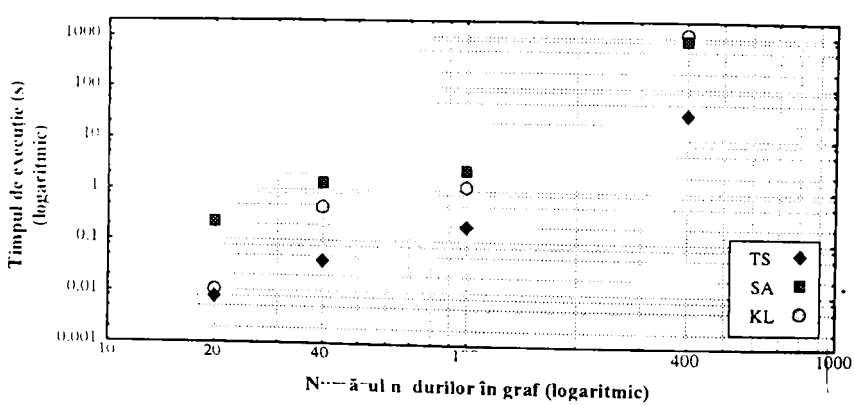


Figura 2.13 Timpii de partiţionare cu SA, TS şi KL

- SA explorează în mod aleator vecinătăţile, în timp ce TS este un algoritm determinist. Caracterul determinist al lui TS face ca adaptarea experimentală a implementării să fie mult mai simplă de cât pentru SA. Adaptarea strategiei de căutare a SA pentru o problemă particulară este simplă şi nu implică un studiu profund al aspectelor specifice problemei. Totuşi, sunt posibile anumite îmbunătăţiri specifice problemei, care să rezulte în creşteri semnificative ale performanţei euristicii SA. Din acest punct de vedere TS este mult mai pretenţios. Elaborarea sa este mult mai complicată şi implică cunoaşterea amănunţită a unor aspecte caracteristice problemei.
- După cum arată tabelul 2.5 şi figura 2.13, performanţele partiţionării cu TS sunt excelente şi mult superioare celei cu SA (în medie de 20 de ori mai rapidă). Acest

lucru este cu atât mai important cu cât după cunoştinţele noastre, TS nu a fost încă folosită pentru partiţionarea automată hardware-software, în timp ce SA este una din euristicile de partiţionare automată cele mai populare.

Euristicile noastre de partiţionare prin SA şi TS le-am comparat cu o metoda clasică de iterative-improvement, ţi anume algoritmul lui Kernighhan-Lin [KL70]. Algoritmul KL are o capacitate restânsă de a evada din minimele locale şi calitatea soluţiei găsită este putenic dependentă de configuraţia iniţială de la care porneşte. De aceea, experimentele pentru acelaşi graf au cuprins execuţii pentru diferite configuraţii aleatoare de start. Numărul execuţiilor diferite pentru acelaşi graf diferă de dimensiunea acestuia. Acest număr este astfel ales încât toate grafurile de o anumită dimensiune să fie partiţionate optim (în sensul definit anterior) cu o probabilitate suficient de mare. Figura 2.13 arată că pentru grafuri mici şi mijlocii timpii de partiţionare cu KL sunt inferiori celor cu SA. Pentru grafuri cu 400 de noduri, SA se comportă superior lui KL. TS este pentru grafuri cu 40 şi 100 de noduri în medie de 10 ori mai rapid ca şi KL, iar pentru grafuri de 400 de noduri de aproximativ 30 de ori mai rapid.

Numărul nodurilor	timp CPU (s)		t_{TS}/t_{SA}
	SA[1] (t_{SA})	TS (t_{TS})	
20	0.23	0.008	0.034
40	1.27	0.04	0.031
100	2.33	0.19	0.081
400	769	30.5	0.039

[1]SA cu mişcări îmbunătăţite

Tabelul 2.5 Timpii de partiţionare cu SA şi TS

2.7 Partiţionarea hardware/software a unor exemple din industrie

Concluziile paragrafului precedent au fost formulate pe baza experimentelor cu grafuri aleatoare şi geometrice. Abordarea noastră pentru patiţionarea specificărilor de sistem am validat-o folosind două exemple din industrie: *coprocesorul pentru reţeaua Ethernet* şi *blocul OAM (F4/F5) al circuitului ATM*. Pentru ambele exemple, specificările lor au fost descrise în VHDL. Folosind informaţiile strânse prin simulare, am extras regiunile de bază pentru fiecare model, iar apoi am construit graful proceselor. Funcţia de cost pentru algoritmii de partiţionare este cea descrisă în paragraful 2.4, iar constrângerea pentru costul componentei hardware a fost fixată la 30% din costul obţinut, când întregul sistem este implementat în hardware. Partiţionarea a fost efectuată cu ambele euristici, atât cea cu SA, cât şi cea cu TS.

Coprocesorul pentru reţeaua Ethernet este un exemplu foarte popular în literatura de specialitate. El a fost folosit ca şi exemplu de specificare de sistem în limbajul SpecCharts [NV92]. Deasemenea, versiuni ale sale în HardwareC sunt utilizate în [GM92] şi [GU95]. Modelul rescris în VHDL este compus din 10 procese ce interacţionează între ele. Lungimea codului sursă este de 730 de linii. Fig. 2.14 ilustrează prin dreptunghiuri cele 10 procese. Coprocesorul transmite şi recepţionează pachetele de date transmise în reţea, prin protocolul CSMA/CD (Carrier Sense

Multiple Access with Collision Detection). Coprocesorul are rolul de a elibera CPU-ul de gestiunea activităților de comunicare. CPU-ul programează coprocesorul pentru diferitele activități prin 8 instrucțiuni. Procesele `rcv-comm`, `buffer-com` și `exec-unit` memorează, decodifică și execută aceste instrucțiuni.

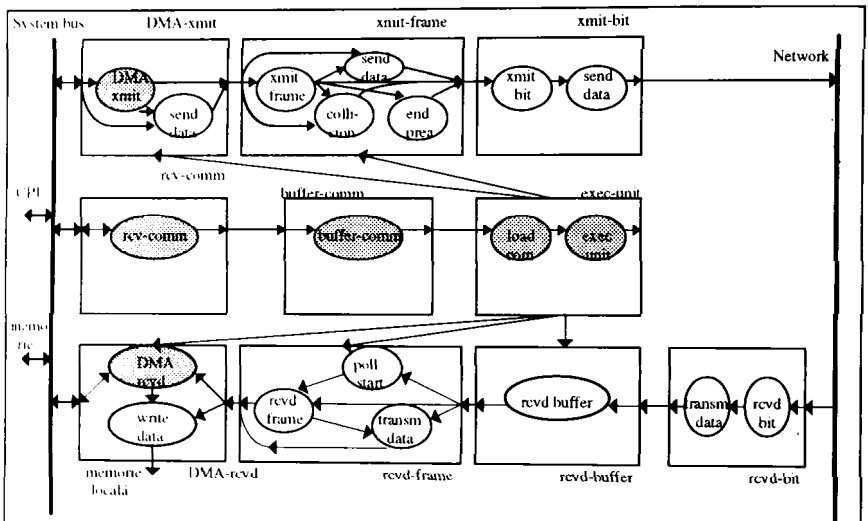


Fig. 2.14 Coprocesorul pentru rețeaua Ethernet

Trei procese cooperează pentru transmiterea datelor spre rețea. Procesul `DMA-xmit` primește o adresă de memorie de la CPU și citește datele, accesând direct memoria. Datele citite sunt transmise următorului proces (`xmit-frame`), unde ele sunt împachetate conform unui standard definit. Pachetele de date sunt trimise sub formă de octeți succesivi către `xmit-bit`, iar `xmit-bit` transmite fiecare octet recepționat, serial spre rețea. Dacă în timpul unei transmisiuni este detectată o situație de *coliziune*, atunci se generează un număr de octeți de jam și pachetul este retransmis după un interval de timp. După transmiterea cu succes a unui pachet de adte, unitatea așteaptă un interval de timp definit, înainte de a transmite următorul pachet de date.

Recepția datelor dinspre rețea spre sistemul gazdă este asigurată de alte patru procese. Procesul `rcvd-bit` recepționează biții transmiși serial în rețea și îi grupează în octeți, pe care îi trimite succesiv procesului `rcvd-buffer`. `rcvd-buffer` are rolul unui buffer pentru memorarea octeților primiți, până când aceștia sunt trimiși mai departe către `xmit-frame`. `rcvd-frame` selectează acele pachete ce sunt destinate sistemului gazdă. După ce a recepționat o secvență de octeți îndicând începutul unui pachet, `rcvd-frame` compară următorii doi octeți cu adresa sistemului gazdă. Când cele două valori sunt egale, `rcvd-frame` trimite următorii octeți ai pachetului spre procesul `DMA-rcvd`, a cărui rol este de a-i scrie în memoria locală.

Primul pas al algoritmului de partiționare extrage în procese noi regiunile intens computaționale. Aceasta a dus la un model cu un număr total de 20 de procese. Fig. 2.14 indică prin ovale procesele specificării rafinate. Apoi, pentru noul modelul este

generat graful proceselor. Partiționând acest graf rezultată 14 procese în hardware și 6 în software. Cele destinate componetei software apar haşurate în Fig. 2.14. Procesele pentru transmisia și recepția de date formează partea intens computațională a coprocesorului, motiv pentru care ele sunt plasate în hardware. Procesele puternic conectate cu acestea, sunt mutate și ele în hardware. Rezultatele noastre sunt în concordanță cu cele raportate de [GM92] și [GU95], care însă au folosit alte abordări.

Al doilea exemplu folosit pentru validarea algoritmului nostru de partiționare corespunde la *blocul pentru realizarea serviciilor OAM (Operation and Maintenance) ale nivelului F4* [PR93]. Acest bloc este folosit în implementarea protocolului de comunicare ATM. Serviciile OAM oferite se referă la gestiunea erorilor (*fault management*), monitorizarea performanțelor (*performance monitoring*), localizarea erorilor (*fault localization*) și funcționalitatea pentru activarea/dezactivarea monitorizării performanțelor (*activation/deactivation*).

Protocolul ATM (*asynchronous transfer mode*) folosește în comunicare celule cu lungime fixă, în a căror header este păstrată informația de routare. Traficul ATM este format din celule succesice, care pot fi destinate la conexiuni diferite. Celula ATM este compusă dintr-un header lung de 5 octeți și un câmp de informație cu lungimea de 48 de octeți. Headerul conține informații referitoare la routare, tipul celulei și pentru controlul corectitudinii transmisiei de informație. Identificarea celulelor ce aparțin unei anumite conexiuni se face folosind două câmpuri din header: VPI (*virtual path identifier*) și VCI (*virtual channel identifier*).

Serviciile OAM ale rețelei sunt executate pentru cele 5 nivele ierarhice asociate prin standard [BE93] protocolului ATM și nivelul fizic (PhL - *Physical Layer*). F4 și F5, care sunt nivelele situate în ierarhie cel mai sus, aparțin modelului ATM.

F4 acoperă următoarea funcționalitate referitoare la VP (*virtual path*) [BE93]:

- *Managementul erorilor*: Celule speciale OAM sunt generate și transmise pe conexiunile pe care au fost detectate erori. Sistemul pentru management este semnalizat atunci când aceste erori persistă.
- *Monitorizarea performanțelor*: Funcționarea normală a rețelei este monitorizată prin verificarea continuă sau periodică a fluxului de celule prin rețea.
- *Localizarea erorilor*: O eroare este localizată prin folosirea unor celule OAM speciale denumite celule loop back.
- *Activarea/dezactivarea*: Această funcționalitate se referă la protocolul special pentru activarea sau dezactivarea acelor funcții OAM, ce implică participarea activă a mai multor blocuri F4 (de exmplu, monitorizarea performanțelor).

Funcțiile enumerate anterior folosesc celule ATM speciale, denumite și celule OAM. Celulele OAM se deosebesc, după funcționalitatea căreia îi sunt destinate, în: celule pentru activare/dezactivare, celule pentru monitorizarea performanței și celule pentru managementul erorilor (FMC). În plus față de celulele OAM, traficul cuprinde și *celule utilizator*.

Modelul ce specifică funcționalitatea blocului F4 al switch-ului ATM este realizat în VHDL (1321 linii ce cod sursă) și este compus din 19 procese ce intecționează între

ele. Figura 2.15 prezintă aceste procese ca și dreptunghiuri. În urma pasului care a extras regiunile de bază ale modelului, numărul proceselor din model a devenit 27, iar noua configurație este prezentată prin ovale în figura 2.15. În urma partiționării hardware/software, 14 procese au fost plasate în hardware, iar 13 în componenta software. Procesele destinate a fi executate în software apar haşurate în figură. În hardware au fost plasate procesele ce filtrează celulele de intrare și tratează celulele utilizator, aceasta pentru că majoritatea covârșitoare a celulelor transmise sunt celule utilizator. În software au fost plasate procesele ce procesează doar celulele OAM (acestea sosesc cu rată foarte scăzută), precum și cele pentru funcționalități executate rar și fără constrângeri de timp (de exemplu procesele `inspect-table` și `clear-error-status`).

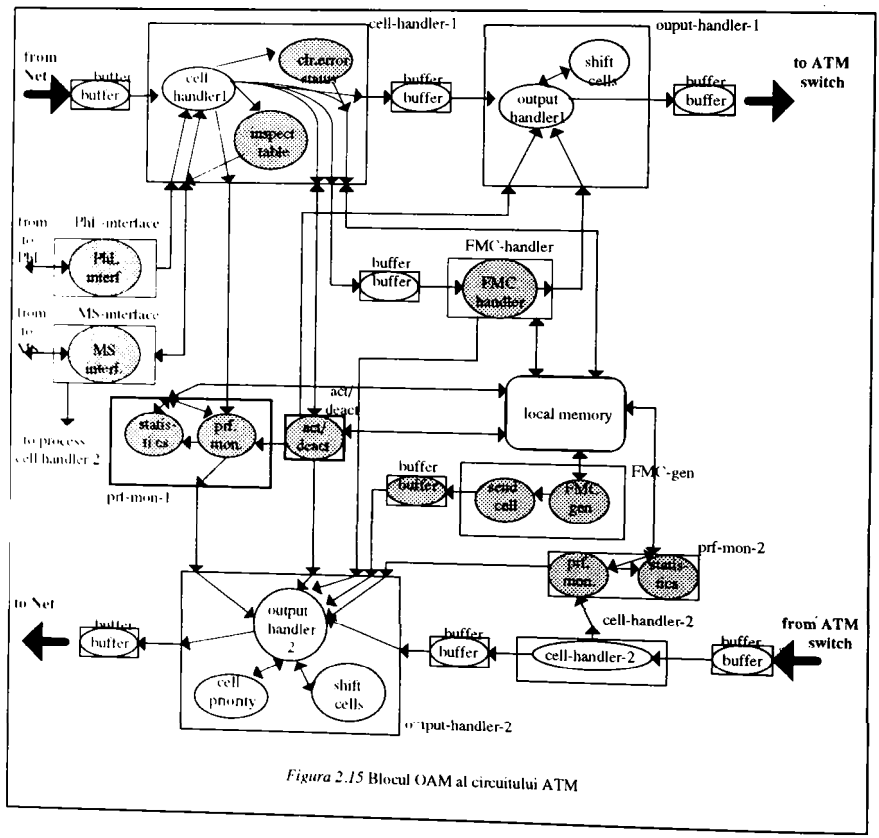


Figura 2.15 Blocul OAM al circuitului ATM

Experimentele noastre cu coprocesorul Ethernet și cu blocul OAM confirmă concluziile obținute în urma experimentelor cu grafuri aleatoare și geometrice. Atât euristica cu SA, cât și cea bazată pe TS au găsit soluțiile optime atunci când au fost executate cu parametrii din tabelele 2.2 și 2.4. Parametrii folosiți pentru ambele exemple au fost:

pentru SA: $TL = 75$, $TI = 400$, $\alpha = 0.95$ și
pentru TS: $\tau = 7$, $Nr\_f\_b = 30$, $Nr\_r = 0$.

Timpii de partiţionare rezultaţi sunt (în secunde):
pentru SA: 0.25 (pentru coprocesorul Ethernet) şi 0.60 (pentru blocul OAM) şi
pentru TS: 0.011 (pentru coprocesorul Ethernet) şi 0.02 (pentru blocul OAM).

Acest rezultat este foarte important din punct de vedere practic, pentru că el certifică faptul că parametrii obţinuţi în urma experimentelor pot fi folosiţi cu succes la partiţionarea a diferite aplicaţii. Astfel poate fi redus sau chiar eliminat timpul de adaptare al euristicilor. Parametrii şi implicit timpii de partiţionare pot fi reduşi la valorile indicate în tabelelul 2.6, pentru că structura grafurilor pentru cele două exemple este mai simplă de cât cea a grafurilor construite artificial. Partiţionarea celor două exemple din industrie a confirmat concluzia noastră că partiţionarea cu TS are un timp de execuţie cu un ordin de mărime mai mic ca şi cea cu SA.

Dacă analizăm toţi cei patru paşi ai algoritmului de partiţionare (Fig. 2.2), putem afirma că etapa partiţionării grafului proceselor este cea mai lungă în timp. Lucrul acesta este absolut normal, din moment ce ceilalţi paşi au complexitate polinomială. Partea cea mai complexă a algoritmului de extragere a regiunilor de bază este cea a analizei necesară pentru generarea intefeţelor cu procesele nou generate [EP94]. De aceea, munca noastră s-a concentrat în special pe găsirea de euristici rapide pentru partiţionarea grafurilor. Pentru coprocesorul Ethernet şi pentru blocul OAM, timpul necesar celor trei etape polinomiale ale partiţionării sunt de acelaşi ordin de mărime cu cel necesar partiţionării grafurilor lor. Dominanţa timpului de partiţionare creşte în raport cu timpii celorlalte etape, cu cât creşte numărul nodurilor în graf.

2.8 Concluzii cu privire la partitionarea hardware-software

Capitolul 2 prezintă abordarea noastră pentru partiţionarea automată, hardware/software a specificărilor de sistem, independente de implementare. Granularitatea la care se face partiţionarea este la nivelul blocurilor de instrucţiuni, ciclurilor, subprogramelor şi proceselor. Implementarea produsă oferă performanţe optime atunci când costurile hardware şi software sunt păstrate sub o anumită limită. Din punctul nostru de vedere, factorii de bază în obţinerea unei performanţe ridicate a implemetării finale sunt minimizarea comunicărilor între componenta hardware şi cea software şi creşterea gradului de paralelism în sistem.

Partiţionarea foloseşte metrici obţinute prin simulare, analiza statică a specificării şi estimarea costurilor hardware şi software. Ponderile asociate nodurilor şi arcelor din graf vor reflecta aceste mărimi. Controlarea procesului de partiţionare are loc printr-o funcţie de cost, care înglobează ponderile nodurilor şi arcelor.

Noi am definit problema partiţionării hardware/software ca problema partiţionării unui graf. Aceasta din urmă am rezolvat-o implementând două euristici, una bazată pe simulated annealing, iar a doua pe tabu search. Euristicile au fost evaluate folosind atât grafuri aleatoare şi geometrice, cât şi exemple din industrie. Ambele euristici produc soluţii de calitate ridicată. Concluzia noastră este că performanţele partiţionării prin TS, care până acuma a fost neglijată în contextul partiţionării hardware/software, sunt superioare celei prin SA (chiar folosind mişcări îmbunătăţite) sau KL. Acest lucru este

foarte important, deoarece pentru grafuri cu număr mare de noduri, timpul de partiționare prin SA sau KL este foarte lung, ceea ce împiedică explorarea eficientă a unui număr cât mai mare de alternative de implementare.

Algoritmii dezvoltați de noi pot fi folosiți și pentru alte probleme de partiționare, de cât partiționarea hardware/software. De exemplu, ei pot fi extinși și folosiți cu același succes în partiționarea la nivele cu granularitate mai coborâtă.

Capitolul 3

Planificarea grafurilor cu dependențe de date

Rezumat

Capitolul 3 adresează problemele aproximării timpului de execuție și respectiv planificării grafurilor legate. Deși problema planificării este una dintre cele mai discutate de literatura de specialitate, există (cel puțin) două motive care justifică cercetarea noastră: spre deosebire de alte abordări, planificarea noastră se referă la grafuri cu nodurile legate resurselor dintr-o arhitectură și după cum justifică capitolul 5, planificarea grafurilor cu dependențe de date este punctul de plecare pentru dezvoltarea euristicilor noastre de planificare a grafurilor cu dependențe de date și control

Mărimile (ASAP, ALAP și drumul critic) folosite cu consecventă în proiectarea euristicilor de planificare primesc o nouă semnificație în cazul grafurilor legate. Acest lucru este motivat de necesitatea de a serializa execuția activităților legate aceloraşi resurse. Noi ținem cont de acest aspect în noile formule pe care le propunem pentru aceste mărimi. Formulele actualizate pentru ASAP, ALAP și drumul critic sunt utilizate în stabilirea unei limite inferioare a timpului necesar pentru execuția unui graf legat. Această limită poate fi folosită ca și criteriu rapid de estimare a timpului de execuție, fără a planifica explicit nodurile grafului. Capitolul schițează un algoritm de planificare bazat pe *branch-and-bound* și care foloseşte limita propusă de noi. Deasemenea, prezentăm experiența noastră legată de planificarea prin metoda programării liniare. Metoda este simplu de formulat și aplicat, este optimă, dar are un timp de execuție prohibitiv de mare, chiar și pentru grafuri cu câteva zeci de noduri. Pentru situațiile în care este acceptabilă o soluție ne-optimă, dar care să fie obținută într-un timp scurt, am propus o euristică de planificare bazată pe liste. Prioritățile folosite reinterpretează noțiunea de drum critic pentru grafurile legate, astfel încât să încurajeze ca timpul total de execuție al grafului să fie cât mai mic. Euristica este rapidă, iar planificările sale sunt apropiate de optimele corespunzătoare.

3.1 Introducere

În această teză, prin *planificare* înțelegem stabilirea momentelor de timp, la care activitățile unui sistem sunt lansate în execuție. După cum reiese din metodologia noastră de proiectare hardware software, detaliată în capitolul 1, planificarea are un impact major asupra structurii componentei software a implementării finale și implicit

asupra timpului total de execuție al sistemului. Folosirea unui algoritm inteligent de planificare conduce la utilizarea intensivă a resurselor arhitecturii și reduce timpul total de execuție al unei implementări. Aceasta permite obținerea unei arhitecturi cu mai puține resurse sau resurse mai ieftine (și probabil cu performanțe mai mici) și implicit, scăderea costului total al implementării.

Problema planificării optime a activităților între care există dependențe este NP-hard [GJ79]. Consecința practică a acestui fapt este că nu se poate dezvolta un algoritm cu timp de execuție polinomial, care să rezolve optim problema pentru cazul ei general. Găsirea optimului înseamnă parcurgerea și analiza întregului spațiu al soluțiilor (cu dimensiune exponențială), ceea ce se poate face într-un timp exponențial în raport cu dimensiunea grafurilor. Ca atare, cercetările în domeniu s-au axat pe trei direcții diferite:

- Introducerea de ipoteze simplificatoare menite să descrie cazuri particulare, ce se pot rezolva exact în timp polinomial de ordin mic. Cercetările în domeniu [SE76][HU61] au identificat două astfel de cazuri. Primul corespunde la planificarea optimă, nepreemptivă a proceselor cu timpi de execuție identici și destinate execuției pe două procesoare. A doilea caz reprezintă planificarea nepreemptivă a arborilor de procese, având timp identici de execuție și executate pe un număr arbitrar de procesoare.
- Perfecționarea metodelor matematice pentru rezolvarea problemelor NP-hard. Această direcție nu elimină caracterul NP-hard al problemei, dar apelând la algoritmi de rezolvare mai eficienți și la calculatoare mai puternice, ea poate extinde domeniul (dimensiunea problemei) pentru care algoritmii de planificare sunt tractabili.
- Dezvoltarea de euristici menite să găsească pentru cazul general soluții apropiate optimului, într-un timp tractabil. Euristicile realizează un compromis între "inteligența" încorporată în ele și timpul lor de execuție. Cu cât construirea soluției folosește informație mai complexă, cu atât complexitatea în timp a algoritmului crește. Euristicile primare folosesc atribute simple, ușor de calculat, dar oferă soluții slabe, chiar și pentru situații nu foarte complicate. Lucrurile sunt complicate și pentru că problema planificării poate rămâne NP-hard, chiar și în condițiile în care se introduc ipoteze simplificatoare privind structura grafului. O asemenea situație este cea în care comportarea unui proces nu depinde de datele de intrare, astfel că între activitățile grafului există doar dependențe de date. Aceste cazuri sunt modelate prin grafuri aciclice, unde arcele între procese indică dependențele de date.

Acest capitol prezintă planificarea activităților într-un graf, între care există dependențe de date. Deși scopul cercetării noastre este de a studia și dezvolta algoritmi de planificare pentru activități între care sunt dependențe de date și *de control*, totuși există două motive pentru separarea și concentrarea planificării cu dependențe de date într-un capitol separat:

- Timpul de execuție al unei activități depinde în mod esențial de tipul resursei căreia îi este alocat. De aceea, în momentul în care algoritmul de planificare este executat trebuie ca toate activitățile să fi fost deja legate resurselor din arhitectură. Acest fapt este fundamental diferit de alte metode de planificare [KN84] [KA96] [BE96] [MB95] etc., care rezolvă în același pas atât planificarea în timp a activităților, cât și maparea lor pe resurse.

- După cum se va detalia în capitolul 5, dezvoltarea euristicilor noastre pentru planificarea activităților cu dependențe de date și de control depinde de existența unor algoritmi buni pentru planificarea cu dependențe de date. Ambele metode de planificare detaliate în capitolul 5 sunt bazate și extind soluții disponibile pentru planificarea activităților legate de resurse, activități între care apar dependențe de date.

Capitolul curent are următoarea structură. Următorul paragraf definește modelul formal pentru reprezentarea unui sistem și care este folosit pentru definirea algoritmilor de planificare. Paragraful trei introduce pentru un graf legat noțiunile de ASAP, ALAP și drum critic. Deasemenea, în acest paragraf se instanțiază formula lui Fernandez a timpului minim de execuție pentru grafuri legate. Paragrafele patru și cinci prezintă metode exacte pentru planificare. Astfel, în patru se indică folosirea metodei branch-and-bound pentru planificare, iar în cinci planificarea prin programare liniară. Paragraful șase este dedicat metodelor euristice. El începe cu o scurtă trecere în revistă a euristicilor existente, după care detaliază algoritmul nostru de planificare a grafurilor legate. Următorul paragraf introduce setul de experimente pe care le-am definit pentru studiul diferitelor euristici, precum și evaluarea statistică a rezultatelor obținute. Capitolul se încheie cu paragraful pentru detalierea concluziilor noastre și indicarea unor direcții de cercetare viitoară.

3.2 Reprezentarea sistemului în vederea planificării

Înainte de a propune o reprezentare a sistemului menită să fie folosită la proiectarea și implemetarea algoritmilor pentru planificare, trebuie să discutăm care este *granularitatea de lucru*. După cum am indicat în capitolul 1, granularitatea de lucru poate fi unul din parametrii de stabilit în timpul ciclului de co-sinteză. Dacă am readapta granularitatea pe parcursul proiectării, atunci am putea crește potențialul de optimizare al proiectării. În acest moment însă, considerăm că granularitatea este fixată de proiectant, implicit prin felul în care el a descris sistemul. Procesul este elementul atomic pentru algoritmii de proiectare (legare, planificare, etc.), în sensul că procesele sunt cele mai mici entități, tratate separat pe parcursul pașilor de proiectare.

Procesele din specificare comunică între ele folosind mecanismul bazat pe mesaje, astfel că datele calculate local de un proces sursă sunt făcute cunoscute procesului țintă prin primitivele *send-receive*. Între cele două există o *dependență de date*, ceea ce impune ca ținta să nu poată fi executată de cât după ce sursa a transmis mesajul. Funcție de modul de mapare al sursei și al țintei pe procesoarele arhitecturii, dependențele de date se particularizează în două categorii:

- Dacă ambele procese sunt plasate *aceluiași* procesor, atunci în momentul în care sursa execută *send*, datele sunt disponibile și către țintă. Astfel, ținta poate să-și înceapă execuția. În acest caz, dependența de date se reduce la o relație de precedență între cele două procese.
- Dacă procesele implicate în comunicare sunt legate la procesoare *diferite*, atunci datele calculate de un proces trebuie comunicate celuilalt, aflat pe o altă resursă. În această situație, dependența de date implică pe lângă o relație de precedență și o

activitate de transmisie de date între două procesoare din sistem. Transmisia foloseşte resursele din arhitectură pe durata ei de execuţie, este nepreemptibilă şi trebuie planificată asemeni proceselor din specificare.

În vederea co-sintezei hardware/software, specificarea sistemului este modelată printr-un graf orientat. Nodurile grafului indică activităţile din sistem, respectiv procesele şi comunicările între procese alocate la resurse distincte. Arcele orientate reprezintă relaţiile de precedenţă între perechi de noduri, astfel că ţinta unui arc este executată doar după ce sursa arcului s-a încheiat. În continuare, un sistem este reprezentat prin graful

$$\mathbf{G = (V, E)}$$

unde **V** este mulţimea nodurilor, iar **E** mulţimea arcelor.

Fiecare activitate a specificării este legată unei resurse, ceea ce implică maparea nodurilor din graf resurselor din arhitectură. Nodurile pentru procese sunt legate de procesoare, iar cele pentru comunicări de magistrale. Funcţia **resouces** modelează această relaţie

resouces: $\mathbf{V} \rightarrow$ {mulţimea resurselor din arhitectură}

resouces(n_i) = resursa de care este legat n_i.

Fiecare nod i din graf are asociat atributul $w(i)$, ce indică timpul de execuţie al nodului pe resursa căreia i-a fost mapat. Acest timp depinde de factori cum sunt: datele de intrare, activitatea reprezentată prin nod, resursa care execută nodul, etc. Aceşti factori fac ca timpul de execuţie să fie mai degrabă modelat printr-un interval de execuţie de cât o valoare unică [YW96]. Capitolul 5 al tezei tratează planificarea grafurilor cu dependenţe de date şi de control, în care relaţiile de control (una din cauzele intervalelor de execuţie) sunt "scoase" în afara nodurilor, iar activităţile dependente de ele sunt reprezentate prin noduri separate. Prin urmare, împreună cu abordarea din capitolul 5, putem considera că w este unic pentru fiecare nod.

Fără a restrânge generalitatea problemei, considerăm că graful are o singură rădăcină şi un singur nod sfârşit. Dacă graful iniţial nu respectă această cerinţă el este transformat adăugându-i-se un nod rădăcină fictiv şi un nod final fictiv, precum şi legăturile necesare cu restul nodurilor din graf. Cele două noduri fictive au timpi de execuţie 0 şi nu sunt legate nici uneia din resursele arhitecturii. A doua proprietate a grafului folosit pentru planificare arată că el este un graf orientat aciclic (GOA), consecinţă a cuprinderii eventualelor cicluri în procese.

Obiectivul urmărit de noi prin planificare este minimizarea timpului de execuţie al sistemului, sau exprimat în termenii modelului pentru planificare, minimizarea timpului de lansare în execuţie a nodului sfârşit.

Cercetările în domeniul co-sintezei hardware/software, dar mai ales al planificării [YW96][GM93][CB94], folosesc cu consecvenţă următoarele mărimi referitoare la graf şi la nodurile grafului:

- $ASAP_i$ (*as soon as possible*) indică pentru nodul i care este cel mai devreme moment când execuţia lui poate începe. $ASAP_i$ este egal cu cel mai lung drum de la rădăcină la nodul i. Lungimea unui drum este suma timpilor de execuţie pentru nodurile sale.

- Drumul Critic (*critical path*) este cel mai lung drum în graf de la nodul rădăcină la nodul sfârşit. Drumul Critic este ASAP-ul nodului sfârşit.
- $ALAP_i$ (*as late as possible*) indică pentru nodul *i* cel mai târziu moment când el îşi poate începe execuţia fără a lungi DC. $ALAP_i$ este diferenţa între DC şi cel mai lung drum de la *i* la nodul sfârşit.

3.3. Definirea ASAP, ALAP şi DC pentru grafuri legate. Limita inferioară a timpului de execuţie pentru un graf legat

Paragraful curent particularizează valorile lui ASAP, ALAP şi DC pentru grafuri legate. Noile formule pentru aceste mărimi sunt mai exacte de cât cele stabilite doar pe baza informaţiilor din graf, deoarece ele cuprind şi modul de legare a nodurilor pe resurse. Pornind de la aceste mărimi, introducem formula pentru calculul limitei inferioare a timpului de execuţie pentru un graf parţial planificat (în cazul limită nici unul din nodurile grafului nu sunt planificate). Aspectele următoare disting studiul nostru de cel din [FB73]: graful nostru al proceselor modelează şi comunicările între procese, procesele şi comunicările sunt legate de resurse, iar arhitectura ţintă este heterogenă, cuprinzând procesoare de diferite tipuri şi magistrale pentru comunicări. Această formulă este un criteriu de estimare a timpului minim de execuţie al unui graf legat. Deasemenea, ea poate fi folosită ca şi limită inferioară pentru proiectarea unui algoritm de planificare bazat pe branch-and-bound (vezi paragraful 3.4). Totodată, concluziile desprinse din această relaţie, pot fi folosite mai departe pentru definirea de euristici de planificare.

[FB73] stabileşte limita inferioară a timpului de execuţie pentru un sistem multiprocesor omogen pornind de la funcţia densităţii de încărcare (*load density function*). Densitatea de încărcare caracterizează activitatea grafului în timp:

$$F(\tau, t) = \Sigma f(t_i, t),$$

unde $f(t_i, t)$ este activitatea nodului *i* din graf. Activitatea nodului este definită de relaţia:

$$f(t_i, t) = 1, \text{ pentru } t \in [t_i, t_i + w(i)], \text{ şi}$$
$$f(t_i, t) = 0, \text{ în rest.}$$

unde t_i este momentul stabilit pentru începerea execuţiei lui *i*. Dacă dorim ca drumul critic (DC) al grafului să nu crească, atunci în mod necesar $t_i \in [ASAP_i, ALAP_i]$. Activitatea globală din graf este caracterizată de formula $\int_0^{DC} F(\tau, t)\, dt$, iar activitatea pentru un interval $[\theta_1, \theta_2] \subset [0, DC]$ (numită şi încărcarea intervalului - *load of segments*) este dată de $\int_{\theta 1}^{\theta 2} F(\tau, t)\, dt$.

Pornind de la faptul că execuţia nodului *i* poate începe oricând în intervalul $[ASAP_i, ALAP_i]$ fără a creşte DC, Fernadez defineşte prin $R(\theta_1, \theta_2, t)$ acele activităţi care trebuie neapărat să se desfăşoare în intervalul $[\theta_1, \theta_2]$. Limita inferioră pe care el o propune pentru timpul total de execuţie al grafului este:

$$T_{carliest} = DC + \max\lceil -(\theta_2 - \theta_1) + 1/m \int_{\theta 1}^{\theta 2} R(\theta_1, \theta_2, t)\, dt \rceil,$$

pentru $\forall [\theta_1, \theta_2] \subseteq [0, DC]$ şi *m* numărul procesoarelor din arhitectură.

3.3.1 Definirea ASAP şi a drumului critic într-un graf legat

Pornind de la faptul că în cazul nostru nodurile grafului sunt legate resurselor, putem găsi estimări şi mai strânse pentru ASAP-urile nodurilor, şi implicit pentru DC şi ALAP-uri. Apoi, reluând raţionamentul parcurs la stabilirea formulei lui Fernadez, estimăm mai exact timpul total de execuţie al grafului. Această estimare poate fi folosită atât pentru definirea lui L pentru planificarea prin BB (vezi paragraful 3.4), dar şi pentru proiectarea unei euristici noi de planificare.

Calcularea ASAP-ului pentru un nod în graf ia în considerare doar caracteristicile grafului (timpii de execuţie ai nodurilor şi dependenţele între noduri), dar nu include nici un fel de informaţii referitoare la modul de mapare a nodurilor pe resurse. Acest lucru este absolut firesc, din moment ce studiile în domeniu nu consideră grafuri cu

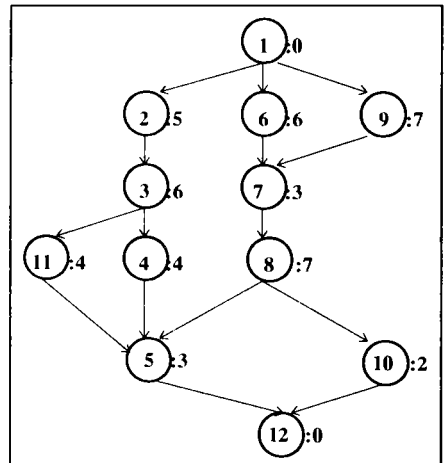


Figura 3.1 Exemplu de graf cu dependenţe de date

nodurile legate de resurse. Însă pentru grafuri legate, deoarece nodurile executate pe procesoarele software sau magistrale trebuie serializate, se pot folosi aceste informaţii suplimentare în determinarea unor estimări mai strânse ale timpilor, când nodurile pot să-şi înceapă activitatea.

Exemplul din figura 3.1 ilustrează o situaţie pentru care, folosind detaliile legate de maparea proceselor pe resurse, obţinem o estimare mai precisă a ASAP. Deasemenea, exemplul este un suport pentru a detalia metoda noastră pentru stabilirea ASAP-urilor nodurilor din graf.

Figura 3.1 indică structura grafului, pentru fiecare nod fiind ataşat (precedat de :) timpul său de execuţie. Dorim să estimăm cel mai devreme moment când poate începe execuţia nodului 5. Această estimare o notăm cu $ASAP_5$. Dacă estimăm $ASAP_5$

folosind doar graful, ea va fi egală cu cel mai lung drum de la 1 la 5 şi ea nu va depinde de modul de mapare a nodurilor pe resurse. Deci, $ASAP_5 = \max(15, 15, 16, 17) = 17$.

În continuare, estimăm valoarea lui $ASAP_5$ şi pe baza modului de legare a nodurilor pe elementele de procesare. De exemplu, considerăm pentru întreaga discuţie că 3 şi 7 sunt legate aceleiaşi resurse, care este diferită de procesorul hardware.

În *primul caz*, presupunem că 6 şi 9 sunt alocate la resurse diferite. Atunci pentru 7, $ASAP_7$ este 7, iar pentru 3 avem $ASAP_3 = 5$. Fiind alocate aceleiaşi resurse, execuţia lui 3 şi 7 trebuie secvenţializată. Un nod din grupul 3 şi 7 poate începe cel mai devreme la minimul între $ASAP_3$ şi $ASAP_5$. Prin urmare, cel mai devreme moment când execuţia ambelor noduri 3 şi 7 poate să se încheie este $min(ASAP_3, ASAP_5) + w(3) + w(7) = 5 + 6 + 3 = 14$. Pentru grupul de noduri 3 şi 7 (grupul corespunde nodurilor alocate aceleiaşi resurse) cel mai scurt drum la nodul 5 are lungimea 4. Astfel, dacă grupul nodurilor 3 şi 7 se încheie la 14, 5 poate începe cel mai devreme la $14 + 4 = 18$. Oricum, 18 este o estimare mai strânsă de cât 17, valoarea obţinută anterior. Acest exemplu arată că pentru anumite grafuri şi legare a nodurilor lor pe resurse, ASAP-urile calculate considerând şi informaţia de legare este mai strânsă de cât cea rezultată folosind doar timpii şi relaţiile între nodurile grafului.

Al *doilea caz* consideră 6 şi 9 plasate pe o aceeaşi resursă, diferită de procesorul hardware. Prin urmare execuţia lor trebuie serializată, astfel că 7 poate începe cel mai devreme la 13. Dacă estimăm momentul când grupul lui 3 şi 7 se încheie folosind relaţia precedentă, adică $ASAP_3 + w(3) + w(7)$, este ca şi cum nu am considera relaţiile de precedenţă pentru 7. Deşi 3 îşi încheie execuţia la 11, datorită predecesorilor, 7 nu începe de cât cel mai devreme la 13, astfel că grupul proceselor 3 şi 7 se încheie cel mai devreme la 16. Continuând analiza pentru acest caz, 5 poate începe cel mai devreme la 20. Cazul acesta surpinde următorul aspect interesant: atunci când stabilim cel mai timpuriu moment când poate să se încheie execuţia unui grup de noduri legate aceleiaşi resurse, trebuie să considerăm atât influenţa predecesorilor, cât şi necesitatea de a serializa execuţia grupului.

În al *treilea caz*, considerăm situaţia din cazul 1, cu diferenţa că procesul 4 are timpul de execuţie 1. Repetând raţionamentul, deducem că grupul lui 3 şi 7 se încheie cel mai devreme la 14, şi adunând la această valoare cel mai scurt drum de la 3 sau 7 la 5 (adică 1), obţinem $ASAP_5 = 15$. Această estimare este mai slabă de cât cea rezultată folosind doar graful. Motivul acestei situaţii este că deşi cel mai devreme moment când grupul 3 şi 7 se încheie corespunde scenariului de planificare 3 urmat de 7, nu putem trage concluzia că cel mai devreme moment când 5 ar putea începe este 21 (adică, 14 + drumul de la 7 la 5). Altfel spus, cel mai mic $ASAP_5$ nu apare pentru scenariul de execuţie, care duce la cel mai timpuriu moment de terminare pentru nodurile grupului 3 şi 7. Un contra-exemplu în acest sens este cel dacă la momentul 5 nu începem execuţia lui 3, lăsând procesorul liber pentru 7. La 7 începe 7 şi dureaza până la 10, la 10 este planificat 3, care durează până la 16, iar astfel execuţia lui 5 poate începe la 20. Trebuie însă să observăm că în acest contra-exemplu grupul 3 şi 7 se încheie la momentul 16, adică mai târziu de cât cel mai devreme moment când execuţia grupului s-ar fi putut încheia. Acest caz evidenţiază două aspecte importante: primul, că **nu** în toate cazurile estimările folosind şi informaţia de legare sunt mai strânse de cât cele folosind doar graful, iar al doilea indică necesitatea unei estimări cât mai precise a celui

mai scurt drum la nodul studiat. Insistăm asupra celui de-al doilea aspect prin următorul caz.

În *cazul patru*, păstrăm situația din cazul unu, dar suplimentar considerăm că 11 și 4 sunt repartizate aceleiași resurse diferită de procesorul hardware. Prin urmare, execuția lui 4 și 11 trebuie serializată, ceea ce arată că timpul minim între terminarea lui 3 și începutul lui 5, este 8 (suma timpilor lui 4 și 11). Folosind informația suplimentară legată de legare, rezultă că cel mai scurt drum între grupul nodurilor 3, 7 și nodul 5 este 7. Deci, 5 nu poate începe înainte de 14 + 7 = 21, față de estimarea 18, cât s-a obținut în cazul unu și 17, valoarea lui $ASAP_5$ calculată doar pe baza grafului. Acest caz arată că lungimile drumurilor de la nodurile legate aceleiași resurse la nodul în cauză, trebuie calculate considerând influențele reciproce ce apar între ele, datorită modului lor de legare pe resurse.

În continuare, prezentăm formula noastră pentru estimarea celui mai devreme moment, când nodul i poate să-și înceapă execuția ($ASAP_i$). Notăm cu M_i mulțimea tuturor submulțimilor de noduri din graf cu următoarea proprietate: o submulțime din M_i cuprinde toate nodurile grafului, predecesoare lui i și legate aceleiași resurse, diferită de procesorul hardware. Fie M_{ik} o submulțime din M_i, $M_{ik} = \{n_{k1}, n_{k2}, \ldots\}$. Pentru simplitatea prezentării considerăm că nodurile în M_{ik} apar în ordinea crescătoare a ASAP-urilor lor, deci:

$$ASAP_{nk1} \leq ASAP_{nk2} \leq$$

Pentru a estima cel mai devreme moment când execuția nodurilor din M_{ik} s-ar putea încheia facem următorul raționament. n_{k1} pornește la momentul $ASAP_{nk1}$ și durează până la $ASAP_{nk1} + w(n_{k1})$. Dacă $ASAP_{nk2} \geq ASAP_{nk1} + w(n_{k1})$, atunci n_{k2} pornește la $ASAP_{nk2}$, altfel, pentru a fi executat, el trebuie să aștepte terminarea lui n_{k1}. Raționametul continuă pentru toate nodurile din M_{ik} până la ultimul, și notăm $term_{pen}$ timpul când se încheie penultimul nod. Dacă $ASAP_{ultim} \geq term_{pen}$ atunci ultimul nod pornește la $ASAP_{ultim}$ și cel mai devreme moment pentru terminarea nodurilor din M_{ik} este $ASAP_{ultim} + w(ultim)$, altfel el trebuie să aștepte terminarea penutimului nod, deci cel mai devreme moment pentru terminarea nodurilor din M_{ik} este $term_{pen} + w(ultim)$.

Lema 3.1. Regula precedentă calculează cel mai devreme moment pentru terminarea execuției nodurilor lui M_{ik} și notăm această valoare cu $earliest(M_{ik})$.
Demonstrație: Demonstrăm această lemă prin inducție matematică.
Demonstrăm lema pentru cazul când M_{ik} are două elemente, $M_{ik} = \{n_{k1}, n_{k2}\}$, și $ASAP_{nk1} \leq ASAP_{nk2}$. Dacă am executa n_{k1} începând cu $ASAP_{nk1}$, el durează până la $ASAP_{nk1} + w(n_{k1})$. Dacă $ASAP_{nk2} > ASAP_{nk1} + w(n_{k1})$, atunci evident că M_{ik} nu se poate încheia înainte de $ASAP_{nk2} + w(n_{k2})$. Dacă $ASAP_{nk2} \leq ASAP_{nk1} + w(n_{k1})$, atunci analizăm două variante de execuție: n_{k1} urmat de n_{k2}, iar timpul total de execuție pentru M_{ik} este egal cu $ASAP_{nk1} + w(n_{k1}) + w(n_{k2})$ și n_{k2} urmat de n_{k1}, când obținem timpul total $ASAP_{nk2} + w(n_{k1}) + w(n_{k2})$. Evident că timpul în prima situație este mai scurt, deci regula noastră este corectă pentru M_{ik} având două elemente.
Presupunem că regula este satisfăcută pentru M_{ik} cu p elemente, demonstăm că ea este adevărată și pentru M_{ik} cu $p + 1$ elemente. Dacă estimăm timpul de execuție pentru submulțimea primelor p elemente din M_{ik} cu regula noastră, atunci acesta este minim și îl notăm cu t (pentru că așa am presupus). Când $t \leq ASAP_{np+1}$ atunci în mod evident

execuția nodurilor lui M_{ik} nu se poate încheia înainte de $ASAP_{np+1} + w(n_{p+1})$. Dacă $t > ASAP_{np+1}$, atunci considerăm submulțimea $S = \{n_{k1}, n_{k2}, ..., n_k\}$ pentru care timpul t' de execuție a nodurilor este mai mic ca și $ASAP_{np+1}$ și S este maximă. Un astfel de t' există, iar la limită el este 0. Executând nodurile $n_{k1}, n_{k2}, ..., n_k$ după regula noastră, t' este minim. Acest lucru este justificat de ipoteza inducției matematice și de faptul că această submulțime are cel mult p elemente. Dacă există noduri cu ASAP-ul mai mic ca și t', atunci timpul total pentru M_{ik} este $t' + \sum_{n_i \in M_{ik}-S} w(n_i)$, indiferent de ordinea în care le executăm. Momentul în care am executat toate nodurile cu ASAP-ul mai mic ca și t' este mai mare ca și $ASAP_{np+1}$ (din ipoteza făcută), astfel că toate nodurile din M_{ik} pot fi executate, fără a lăsa procesorul neutilizat. Valoarea lui $t' + \Sigma w$ coincide cu cea calculată aplicând lema 3.1. Dacă nu există nici un nod cu $ASAP < t'$, atunci procesorul rămâne liber până la nodul cu primul ASAP Dacă îl executăm pe acesta, atunci la terminarea sa toate nodurile rămase devin disponibile pentru execuție (ipoteza făcută), iar timpul care rezultă este cel din lema 3.1. Orice alt scenariu de execuție duce către un timp mai lung, din cauză că procesorul rămâne mai mult timp neocupat. Prin urmare, regula noastră este adevărată și pentru cazul $p + 1$, ceea ce încheie demonstrația.

Observație: Această estimare pentru cel mai devreme moment de terminare a nodurilor din M_{ik} este mai strânsă de cât cea dată de formula $ASAP_{nk1} + \Sigma w(n_j)$, pentru $n_j \in \{n_{k1}, n_{k2}, ...\}$, deoarece aceasta din urmă nu consideră relațiile de precedență pentru nodurile din M_{ik}.

Lema 3.2: Dacă un nod $n_j \in M_{ik}$ este legat de nodul n_i prin cel puțin un drum, atunci n_i nu poate începe de cât cel mai devreme la Δ unități de timp după ce n_j se încheie, unde Δ este lungimea celui mai lung drum de la n_j la n_i.
Demonstrație: Demonstrația lemei este implicită din relațiile de precedență între n_j și n_i.

Lema 3.3: Cel mai devreme moment, când nodul n_i poate să-și înceapă execuția este mai mare cel mult egal cu suma dintre cel mai devreme moment pentru terminarea nodurilor din M_{ik} (definit după regula de mai sus) și lungimea Δ_{ik} a celui mai scurt Δ de la un nod din M_{ik} la n_i, oricare ar fi mulțimea M_{ik} a lui n_i. Notăm valoarea limitei ce rezultă din această lemă cu $ASAP^{aloc}_i$.

$$ASAP^{aloc}_i = max(earliest(M_{ik}) + \Delta_{ik}), \forall M_{ik}.$$

Demonstrație: Rezultă implicit pe baza ultimelor două leme, respectiv a noțiunii de drum minim între două noduri.

După cum a rezultat în situația patru, valoarea lui $ASAP_i$ este îmbunătățită dacă la calcularea lungimii Δ a drumului de la nodul $n_j \in M_{ik}$ la nodul în cauză, se iau în considerare simultan toate drumurile de la n_j la n_i și nu fiecare drum separat. Acest lucru este benefic pentru că pot fi surprinse eventualele alungiri datorită modului de mapare a nodurilor pe resurse. Dacă rutina ce calculează timpul minim de execuție pentru un graf se numește `drum_minim`, atunci apelăm `drum_minim` pentru subgraful cu rădăcina în n_j, nodul final în n_i și conținând toate drumurile de la n_j la n_i.

Rezultatele anterioare conduc la următoarea regulă pentru calcularea ASAP-urilor pentru nodurile unui graf. Dacă nodul i are $ASAP^{graf}_i$, ASAP-ul calculat doar pe baza grafului, și $ASAP^{aloc}_i$ cel calculat folosind lema 3.3, atunci el nu poate începe mai devreme de

$$max(ASAP^{graf}_i, ASAP^{aloc}_i).$$

Particularizând această relație pentru nodul final al grafului, obținem o formulă nouă pentru *drumul critic*

$$DC = max(ASAP^{graf}_{nod\ final}, ASAP^{aloc}_{nod\ final}).$$

Această valoare poate fi interpretată și ca o limită inferioară pentru timpul de execuție al întregului graf.

Comentarii cu privire la regula propusă pentru calculul lui ASAP.

În această secțiune facem o paralelă între formula propusă de noi și metoda prezentată în [NM96] pentru "strângerea" ASAP-urilor pentru nodurile din grafuri legate. Deasemenea vom arăta că această metodă poate fi considerată o simplificare a regulii formulate de noi. După cunoștințele noastre, [NM96] este singura ce folosește legarea nodurilor pentru găsirea unor valori mai strânse pentru ASAP-uri.

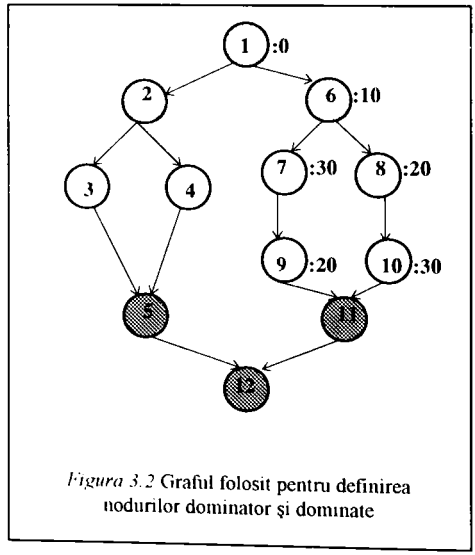


Figura 3.2 Graful folosit pentru definirea nodurilor dominator și dominate

Pentru prezentarea metodei lui Niemann folosim figura 3.2. Ea folosește noțiunile de nod *dominat* și nod *dominator*. Pentru figura următoare 1, 2 și 6 sunt noduri dominator, iar 5, 11 și 12 sunt nodurile dominate. Presupunem că 9 și 10 sunt alocate aceleiași resurse, iar 7 și 8 la resurse diferite. Niemann consideră că un nod dominat nu poate să-și înceapă execuția mai repede de cât valoarea dată de momentul când dominatorul său se încheie adunat cu cea mai mare dintre sumele timpilor nodurilor situate între dominator și dominat și legate aceleiași resurse.

Pentru acest exemplu, 11 nu poate începe înainte de 10 (când se termină dominatorul său 6) + $w(9) + w(10) = 10 + 20 + 30 = 60$. Această valoare nu este o îmbunătățire față de cea calculată considerând graful nealocat. Pentru acest caz, formula noastră este mai exactă: $ASAP_9 = 40$, $ASAP_{10} = 30$, prin urmare $ASAP^{aloc}_{11} = \min(ASAP_9, ASAP_{10}) + w(9) + w(10) = 30 + 20 + 30 = 80$. Prin urmare, $ASAP_{11} = \max(ASAP^{aloc}_{11}, ASAP^{graf}_{11}) = \max(50, 70) = 80$.

Regula formulată de Niemann se referă doar la nodurile dominate şi se obține aplicând acestora regula noastră, cu următoarele particularizări: în calcule se folosesc doar subgrafurile cuprinse între dominatori şi nodurile dominate, considerăm că $ASAP_{nk1} = ASAP_{nk2} = \ldots$ = timpul de terminare al dominatorului pentru nodurile oricărei mulțimi $M_{dominat\ k}$, şi nu luăm în considerare cel mai scurt drum de la nodurile din $M_{dominat\ k}$ la nodul dominat.

Algoritmul pentru calcularea ASAP într-un graf alocat

```
void calculează_ASAP (nodul i, graf G) {
      if (∃ predecesori direcți ai lui i pentru care ASAP nu
      a fost calculat)
            return;
      else {
            ASAP^graf_i = -∞;
            for (∀ j, predecesor direct al lui i în graful G)
                  if (ASAP_j + w(j) > ASAP^graf_i)
                        ASAP^graf_i = ASAP_j + w(j);
            ASAP^aloc_i = -∞;
            for (∀ mulțime M_ik a lui i) {
                  t^finish_earliest = calculează cel mai timpuriu
                  moment când nodurile din M_ik se termină;
                  for (∀ nod j ∈ M_ik)
                        Δ_j = drum_minim (subgraful
                        drumurilor de la j la i);
                  Δ = min(Δ_j), pentru ∀ j ∈ M_ik;
                  if (ASAP^aloc_i < t^finish_earliest + Δ)
                        ASAP^aloc_i = t^finish_earliest + Δ;
            }
            ASAP_i = max(ASAP^graf_i, ASAP^aloc_i);
            for (∀ j, succesor direct al lui i în graful G)
                  calculează_ASAP (j, G);
      }
}
```

calculează_ASAP parcurge graful *G*, şi calculează ASAP-ul unui nod în momentul în care ASAP-urile tuturor predecesorilor săi sunt calculate. În prima etapă, rutina stabileşte $ASAP^{graf}$, adică momentul când ultimul dintre predecesorii săi direcți se termină. Apoi pentru fiecare din mulțimile M_{ik} ale nodului, algoritmul calculează cel mai devreme moment când toate nodurile din M_{ik} sunt executate ($t^{finish}_{earliest}$). La această valoare se adaugă cel mai scurt Δ_j pentru $j \in M_{ik}$. Maximul acestor sume este $ASAP^{aloc}$. Maximul dintre $ASAP^{graf}$ şi $ASAP^{aloc}$ este ASAP-ul nodului.

3.3.2 Calculul ALAP într-un graf legat

După ce s-a calculat DC al unui graf, pentru fiecare nod stabilim cel mai târziu moment când acesta își poate începe execuția, fără a mări DC (acest moment se numește ALAP). ALAP trebuie să permită tuturor succesorilor unui nod să fie executați fără a mări DC. Teoria [GD92] identifică timpul necesar execuției succesorilor cu cel mai lung drum al nodului la nodul final. Dacă notăm acest drum cu L_{max}, rezultă că

$$ALAP = DC - L_{max}.$$

În cazul nostru, pentru că nodurile sunt legate, L_{max} poate fi estimat mai strâns luând în considerare întârzierile produse de nodurile altor drumuri, dar care sunt alocate aceleiași resurse. Prin urmare, pentru a stabili timpul necesar execuției succesorilor unui nod, considerăm subgraful cu rădăcina în nodul în cauză, nodul final în nodul final al grafului și conținînd toate drumurile de la nodul în cauză la nodul final. Aplicând rutina *drum_minim*, menționată în paragraful 3.3.3, calculăm timpul minim necesar pentru execuția subgrafului (acest timp îl notăm cu t_{minim}). Atunci ALAP-ul nodului este

$$ALAP = DC - t_{minim}.$$

```
void calculează_ALAP (graf G, DC) {
      for (∀ nod i ∈ G) {
            SG = subgraful care are pe i rădăcină, nodul final în
            nodul final al lui G și conține toate drumurile de la
            i la nodul final al lui G;
            t      = drum_minim (SG);
            ALAP_i = DC - t_minim;
```

3.3.3 Estimarea timpului de execuție a grafului

Pentru estimarea timpului total de execuție al grafului repetăm raționamentul prezentat în [FB73] și care a fost detaliat pe scurt la începutul paragrafului. Particularizăm raționamentul său pentru grafuri cu nodurile legate resurselor și folosim îmbunătățirea din [MO90] pentru stabilirea intervalelor $[\theta_1, \theta_2]$. Menționăm că în momentul în care se calculează limita inferioară pentru timpul de execuție al grafului, fiecare nod are calculate ASAP și ALAP, și deasemenea este disponibil DC al grafului.

Asemănător cu [MO90], pentru fiecare nod i din graf considerăm intervalul $[ASAP_i, ALAP_i + w(i)]$, care va cuprinde cu siguranță execuția lui i, pentru ca DC să nu crească. Restul proceselor alocate aceleiași resurse cu i (resursă diferită de procesorul hardware) sunt deplasate în intervalele lor $[ASAP, ALAP + w]$, astfel încât să aibă suprapunere minimă cu $[ASAP_i, ALAP_i + w(i)]$. Dacă notăm cu S suma suprapunerilor astfel rezultate, alungirea pentru intervalul $[ASAP_i, ALAP_i + w(i)]$ este

$$\Delta t_i = \max (0, S - (ASAP_i - ALAP_i)).$$

Am considerat maximul cu 0 pentru a elimina eventualele valori negative ale lui $S - (ASAP_i - ALAP_i)$.

Limita inferioară a timpului de execuție al grafului rezultă particularizând relația lui Fernandez pentru fiecare resursă din arhitectură. Pentru fiecare resursă diferită de procesorul hardware şi pentru fiecare nod alocat ei, calculăm Δt_i lui. Maximul tuturor Δt_i -urilor pentru nodurile alocate resursei r este alungirea timpului de execuție pentru resursa respectivă (notăm valoarea cu q_r). Alungirea timpului total de execuție al grafului este maximul alungirilor pentru toate resursele din arhitectură.

Prin urmare, limita inferioară pentru timpul de execuție al grafului este:

$T_{limită}$ = DC + $max(q_r)$, pentru $r \in$ resursele din arhitectură - procesorul hardware.

Algoritmul pentru stabilirea limitei inferioare pentru timpul de execuție al unui graf este:

```
void drum_minim (Graf G, Arhitectură A, legarea M) {
      calculează_ASAP (rădăcina lui G, G);
      DC = ASAP_nodul_final;
      calculează_ALAP (G, DC);
      R_max = 0;
      for (∀ resursă k ∈ A, k ≠ procesorul hardware)
            for (∀ nod i în graf, legat lui k) {
                  S = 0;
                  for (∀ nod j în graf, legat lui k, i ≠ j) {
                        mută T_j^S, începetul execuției lui j, în
                        intervalul [ASAP_j, ALAP_j] a.î.
                        d_ij = [T_j^S, T_j^S + w(j)] ∩ [ASAP_i, ALAP_i + w(i)]
                        să fie minimă;
                        S = S + d_ij;
                  }
            S = S - (ALAP_i - ASAP_i);
            if (R_max < S)
                  R_max = S;
            }
      timpul_minim = DC + R_max;
}
```

Algoritmul pentru stabilirea timpului minim de execuție a grafului G, cu arhitectura A şi legarea M începe prin calcularea valorilor ASAP şi ALAP pentru noduri şi DC-ul grafului. Apoi, pentru fiecare resursă din graf diferită de procesorul hardware şi pentru fiecare nod alocat resursei se calculează valoarea suprapunerilor (variabile S) din partea altor noduri legate aceleaşi resurse. R_{max} este maximul acestor suprapuneri şi stabileşte valoarea cu care creşte DC al grafului.

3.4 Planificarea activităţilor folosind Branch-and-Bound

Planificarea activităţilor într-un graf prin metoda *branch-and-bound* (pe care o notăm în continuare prescurtat BB) [KO74] [KS75] găseşte secvenţa optimă de execuţie a activităţilor pentru un anumit criteriu de optimizat. Principial, BB parcurge întregul spaţiu al soluţiilor şi memorează optimul, dar timpul său de căutare este redus prin eliminarea soluţiilor parţiale, care nu pot conduce către o soluţie superioară celei cunoscute la momentul respectiv. Cu cât numărul soluţiilor parţiale eliminate este mai mare şi cu cât eliminarea din procesul de căutare se produce mai repede, cu atât viteza lui BB creşte. Prin urmare, performanţa (exprimată ca şi timpul de căutare) unui algoritm de planificare bazat pe BB creşte, cu cât criteriul după care se face eliminarea soluţiilor parţiale este mai strâns. Astfel, putem afirma fără rezerve că "nodul gordian" al acestei metode stă în găsirea unor reguli eficiente pentru trierea soluţiilor intermediare. Totuşi, trebuie să avem în vedere că, indiferent de calitatea acestor reguli de eliminare, nu putem evita caracterul NP-hard al problemei planificării, deci nu ne putem aştepta să rezolvăm optim problema pentru grafuri cu orice dimensiune. Un criteriu strâns de evaluare a unei planificări parţiale creşte domeniul problemelor (exprimat ca număr de noduri în graf) pentru care putem obţine optime într-un timp tractabil.

În continuare, detaliem elementele de bază ale metodei BB, după cum sunt ele prezentate în [KO74][KS75][KN84]. Algoritmul este format din următoarele componente:

- Bp (*branching rule*) este regula după care problema planificării este descompusă în subprobleme. Paşii parcurşi de algoritm corespund momentelor de timp când taskurile de pe unul sau mai multe procesoare îşi termină execuţia, iar succesorii lor devin pregătiţi pentru execuţie. Notăm cu m numărul resurselor disponibile, iar cu k numărul taskurilor pregătite. Deciziile de planificare ce pot fi luate la fiecare pas, corespund situaţiilor când unul din cele k taskuri ready este executat pe unul din cele m procesoare disponibile şi respectiv variantelor când una sau mai multe resurse sunt lăsate libere (notăm cu n_{idle} numărul variantelor "libere"). Strategia de a lăsa unele resurse libere este benefică, dacă în viitorul apropiat pe acele resurse devin ready taskuri de prioritate mare şi care nu trebuie întârziate. Numărul variantelor de planificare ce trebuie analizate la fiecare pas, este $n_{branch} = C^{m}_{(k+nidle)}$.
- S (*selection rule*) defineşte ordinea în care sunt vizitate deciziile de planificare. Această regulă reprezintă o euristică pentru ordonarea spaţiului de căutare, dar ea poate mări considerabil viteza de căutare, favorizând eliminarea unui număr mare de cazuri intermediare.
- L (*lower bound function*) stabileşte limita superioară a calităţii soluţiei finale, ce poate rezulta pentru o anumită configuraţie intermediară. Dacă această calitate se referă la timpul total de execuţie a grafului, atunci avem certitudinea că L(soluţie intermediară) ≤ timpul de execuţie pentru cea mai bună soluţia finală, ce rezultă pentru soluţia intermediară curentă.
- U (*upper bound function*) este o soluţie a problemei care se cunoaşte *apriori* planificării. Valoarea lui U este reactualizată ori de câte ori găsim o soluţie mai bună ca şi U. Este cunoscut că cu cât U este mai apropiat optimului, cu atât spaţiul de căutare este mai mic (un număr mare de planificări intermediare sunt eliminate),

astfel că viteza lui BB este mai mare. Prin urmare, este benefică şi indicată cunoaşterea unei soluţii de start (obţinută eventual printr-o euristică) cât mai aproape de optim.

- E (*elimination rule*) este regula după care se face eliminarea soluţiilor intermediare, care nu pot conduce spre soluţii finale superioare celei deja cunoscute. BB elimină planificările parţiale pentru care L(planificare parţială) ≥ timpul de execuţie al optimului parţial găsit.
- ε (*acceptable approximation error*) reprezintă eroarea soluţiei găsite de BB faţă de cea optimă. ε este util pentru situaţiile când cunoşterea unei soluţii aproximative este suficientă. Experimentele din [KN84] indică faptul că pentru ε = 0.1 sau 0.05 s-au obţinut soluţii aproximative într-un timp tractabil (de ordinul secundelor), pentru grafuri cu până la 200 de noduri.

Din cele şase componente, trei (Bp, E şi ε) depind de natura căutării şi putem să le considerăm invariante, dar S, L şi U constituie subiecte de studiu. Discuţia noastră asupra planificării cu BB prezintă, în continuare, o metodă din literatura de specialitate, iar apoi schiţează metoda noastră de planificare a grafurilor cu nodurile legate resurselor.

DF/IHS [KN84] este o combinaţie între o euristică de planificare bazată pe liste şi planificarea folosind BB. DF/IHS planifică static, nepreemptiv nodurile unui graf, între care există dependenţe de date. Criteriul optimizat în planificare este minimizarea timpului total de execuţie al grafului. Nodurile grafului nu sunt legate de resurse şi se cunoaşte arhitectura ţintă (numărul de procesoare şi felul lor). Menţionăm că arhitectura este omogenă, deci toate procesoarele sunt identice şi nu sunt făcute referiri privind topologia lor de interconectare. Rezultatul oferit de DF/IHS cuprinde atât procesoarele unde sunt executate, cât şi secvenţa de execuţie pentru taskuri.

În continuare detaliem felul în care DF/IHS instanţiază S, E şi U. După cum am menţionat deja, DF/IHS combină BB cu CP/MISF [KN84], o euristică de planificare bazată pe liste. Din experienţa autorilor, CP/MISF se comportă acceptabil în majoritatea situaţiilor (găseşte optimul în aproximativ 67%), fiind în 98.5% din cazuri la mai puţin de 10% de optim. În paragraful rezervat euristicilor de planificare bazate pe liste, vom detalia modul în care CP/MISF calculează priorităţile taskurilor. Într-o etapă de preprocesare, fiecărui task i se asociază prioritatea dată de CP/MISF. Această prioritate este folosită pentru definirea lui S, deoarece taskurile ready sunt vizitate în ordinea priorităţii lor. Ca urmare a acestei reguli, prima soluţie finală stabilită de DF/IHS este chiar cea ce se obţine cu CP/MISF (deci U este obţinut tot prin CP/MISF). Pentru L, autorii folosesc relaţiile prezentate în [FB73] şi care extind formulele din [HU61]. În [FB73] se defineşte limita inferioară a timpului de execuţie a unui set de taskuri cu dependenţe pentru un număr cunoscut de procesoare identice. Experimentele autorilor arată că DF/IHS se comportă rezonabil pentru grafuri cu până la 200 de taskuri. Astfel, în 75% din cazuri se găsesc optimele în mai puţin de 180 de secunde, 92% din cazuri sunt rezolvate cu o eroare de mai puţin de 5% şi toate situaţiile sunt rezolvate cu eroare de până la 10% (în timpul de 180 de secunde).

Întrebarea următoare este: prin ce se aseamănă abordarea din [KN84] de problema noastră, şi bineînţeles, unde apar deosebirile? DF/IHS este destinat planificării statice,

nepreemptive a taskurilor între care există dependenţe de date. Timpul de execuţie al taskurilor este modelat prin numere întregi. În fine, criteriul optimizat prin planificare este minimizarea timpului total de execuţie al grafului. La aceste asemănări cu problema noastră trebuie să adăugăm următoarele diferenţe. DF/IHS foloseşte o arhitectura ţintă omogenă, unde fiecare procesor poate executa pe oricare din taskurile grafului, iar timpul de execuţie al taskurilor nu depinde de procesor. Aceasta are ca şi consecinţă faptul, că deşi sunt considerate dependenţele de date între procese, nu sunt modelate comunicările între procese mapate la procesoare diferite (sau formal, timpul lor de comunicare este 0). Deasemenea, într-o arhitectură heterogenă, timpul de execuţie al unui task depinde de felul procesorului care îl execută (cu remarcă specială pentru taskurile implementate în hardware, pentru că acestea pot fi executate în paralel) În fine, DF/IHS rezolvă legarea şi alocarea taskurilor în acelaşi timp, în timp ce în cazul nostru, procesele şi comunicările se consideră deja mapate resurselor din arhitectură.

În continuare, instanţiem S, L şi U pentru planificarea proceselor şi a comunicărilor legate de resurse. Ca şi regulă L folosim formula noastră prezentată în paragraful 3.3 pentru estimarea limitei inferioare a timpului de execuţie pentru un graf legat. Dacă graful este parţial planificat, atunci ASAP-urile nodurilor planificate sunt egale cu timpul la care ele urmează să fie executate, iar pentru restul nodurilor, aceasta se calculează cu algoritmul `calculează_ASAP`. Estimarea timpului total de execuţie îl obţinem apelând `drum_critic` şi dacă acesta este mai mare ca şi cea mai bună soluţie cunoscută până în acel moment, atunci planificarea parţială este eliminată din căutare, întru-cât nu conduce spre soluţii superioare. Ca şi regulă de parcurgere a alternativelor (S) şi ca soluţie de start (U) folosim cele date de cea mai bună euristică bazată pe liste experimentată de noi. În urma rezultatelor experimentale, a reieşit că CP/MISF-ML are cel mai bun comportament pentru grafurile legate. Prin urmare, priorităţile ataşate nodurilor sunt cele corespunzătoare lui CP/MISF-ML.

3.5 Planificarea activităţilor dintr-un graf legat folosind Programarea Liniară

Folosirea metodei programării liniare [TH88] implică modelarea problemei planificării printr-un set de ecuaţii şi inecuaţii, ce redau constrângerile de planificare şi respectiv o inecuaţie pentru descrierea criteriului de optimizat prin planificare. Programarea liniară găseşte soluţia, ce satisface setul de ecuaţii şi inecuaţii şi optimizează (maximizează sau minimizează) criteriul modelului. După cum soluţiile problemei sunt numere întregi, numere reale sau întregi şi reale, avem de-a face cu *programare liniară întreagă* (integer linear programming sau ILP), *programare liniară* (linear programming - LP) sau *programare liniară, mixtă* (mixed integer linear programming - MILP).

Obiectivul planificării noastre este minimizarea timpului de execuţie pentru un graf cu nodurile legate şi el este descris ca atare prin criteriul de optimizat al modelului. Restul relaţiilor din model se referă la următoarele tipuri de constrângeri: respectarea relaţiilor de precedenţă - un nod nu poate începe de cât după ce toţi predecesorii săi au fost executaţi şi respectiv modelarea conflictelor privind utilizarea în comun a resurselor din arhitectură - două noduri legate aceluiaşi procesor software sau magistrală nu pot fi

executate simultan. Ultima restricție cade pentru nodurile legate procesorului hardware din cauza capacității sale de a le executa concurent.

Avantajul major al planificării bazate pe programare liniară provine din faptul că este o metodă exactă, astfel că rezultatul găsit este întotdeauna cel optim. Suplimentar, metoda este flexibilă şi uşor de utilizat, pentru că construirea modelului pentru o aplicație particulară este simplă şi poate fi uşor automatizată. În acelaşi timp, acest model este o descriere formală a problemei planificării, oferind toate avantajele formalismelor privind claritatea, precizia şi corectitudinea descrierii. Dezavantajul major al planificării cu LP provine din faptul că soluționarea oricărei probleme de programare liniară este NP-hard [SW88], ceea ce face ca ea să fie aplicabilă practic doar pentru grafuri cu dimensiuni mici. Ca urmare a stadiului actual privind folosirea LP pentru planificare, cercetările în domeniu sunt axate pe două direcții: prima cuprinde dezvoltarea de metode noi, eficiente pentru programare liniară [MB95][AC95], iar a doua, foloseşte cunoaşterea dobândită prin această metodă pentru proiectarea unor euristici eficiente.

În continuare, descriem metoda noastră de modelare folosită pentru planificarea prin LP a grafurilor legate. Pentru un nod i sunt valabile următoarele notații: $w(i)$ este timpul de execuție al activității modelate prin nod, T_i^s timpul când execuția lui i începe şi T_i^f cel când ea se încheie.

Ecuațiile şi inecuațiile modelului sunt grupate în următoarele categorii:

Ecuatii şi inecuatii pentru descrierea relațiilor de precedență între noduri

În mod evident, timpul când se termină execuția lui i este:

$$T_i^f = T_i^s + w(i)$$

Un nod nu îşi poate începe execuția până când toți predecesorii săi direcți sunt executați, ceea ce se traduce pentru nodul i în următoarea inecuație:

$$T_i^s \geq max(T_j^s + w(j)),\ \forall\ j \text{ un predecesor direct al lui } i.$$

Astfel de ecuații şi inecuații sunt scrise pentru toate nodurile din graf.

Inecuatii pentru modelarea conflictelor pe resurse

A doua categorie de restricții se referă la serializarea execuției nodurilor legate aceluiaşi procesor software, magistralelor sau ele sunt în hardware, dar utilizate în comun (*shared*). Dacă nodurile i şi j aparțin aceleiaşi resurse şi între ele nu există nici o relație de precedență (nici unul nu este predecesorul celuilalt), atunci trebuie descrisă serializarea execuției lor. Pentru aceasta se definesc următoarele două inecuații:

$$T_i^s \geq T_j^s + w(j) - \infty * b_{ij}$$
$$T_j^s \geq T_i^s + w(i) - \infty * b_{ji}$$

unde variabilele întregi b_{ij}, $b_{ji} \in \{0, 1\}$ şi au următoarea semnificaţie: dacă i este planificat după terminarea lui j, atunci $b_{ij} = 0$ şi $b_{ji} = 1$, iar dacă i este planificat înaintea lui j, atunci $b_{ij} = 1$ şi $b_{ji} = 0$. Prin urmare, b_{ij} şi b_{ji} caracterizează ordinea de execuţie a lui i relativ la j şi ele satisfac **întotdeauna** următoarea egalitate:

$$b_{ij} + b_{ji} = 1$$

Pe baza ultimei egalităţi, deducem că $b_{ji} = 1 - b_{ij}$, iar setul de restricţii pentru serializarea execuţiei lui i şi j devine:

$$T_i^s \geq T_j^s + w(j) - \infty * b_{ij}$$
$$T_j^s \geq T_i^s + w(i) - \infty * (1 - b_{ij})$$

Acest set de inegalităţi este echivalent cu cel iniţial, dar economiseşte o variabilă (b_{ji}) şi o ecuaţie ($b_{ij} + b_{ji} = 1$), ceea ce micşorează atât dimensiunea modelului, precum şi timpul său de rezolvare.

Ecuaţiilor şi inecuaţiilor ce definesc problema planificării unui graf cu nodurile legate, li se adaugă **criteriul de optimizat** de către planificare şi anume ca timpul total de execuţie al grafului să fi minim. Acest fapt se exprimă prin:

$$Minimizează\ T^s_{nod\ final}$$

Inecuatii pentru uşurarea rezolvării modelului

Aceste inecuaţii nu ţin de modelarea planificării nodurilor, ci ele au menirea de a reduce spaţiul soluţiilor explorat. Prima categorie de constângeri redundante sunt introduse de ASAP şi ALAP. Pentru nodul i din graf, el nu îşi poate începe execuţia mai repede de $ASAP_i$, din cauza dependenţelor sale faţă de predecesorii săi. Această constângere este indicată prin:

$$ASAP_i \leq T_i^s$$

De regulă, specificarea sistemului impune ca execuţia sa să se încheie înainte de o limită t_{cr}. Pentru fiecare nod în graf se stabileşte valoarea ALAP, care indică cel mai târziu moment când nodul poate începe, fără ca timpul total de execuţie al grafului să depăşească t_{cr}. $ALAP_i$-ul pentru nodul i trebuie să fie suficient de strâns, astfel încât inclusiv cel mai lung drum de la i la nodul final în graf, să poată fi executat fără a depăşi t_{cr}. Prin urmare, i trebuie să înceapă cel târziu la $ALAP_i$ pentru a nu creşte t_{cr}:

$$T_i^s \leq ALAP_i$$

Relaţiile între T^s, ASAP şi ALAP sunt formulate pentru toate nodurile în graf.

În afară de a limita intervalul de apartenenţă pentru T^s, ASAP şi ALAP pot fi folosite pentru a evita generarea inutilă a ecuaţiilor pentru secvenţierea nodurilor. Acest fapt este perfect justificat, deoarece experimentele noastre au indicat că inclusiv pentru exemple nu foarte mari, majoritatea inecuaţiilor sunt pentru descrierea constrângerile de secvenţiere. Fiecare pereche de noduri legate aceleiaşi resurse introduce o variabilă şi trei relaţii, astfel că pentru k procese independente alocate aceleiaşi resurse, numărul variabilelor necesare pentru modelare secvenţierii este $\mathbf{C}^2_k$, iar numărul relaţiilor este $3\mathbf{C}^2_k$.

Revenind la folosirea lui ASAP şi ALAP în reducerea modelului, folosim observaţia că relaţiile de secvenţiere trebuie scrise doar pentru procesele ale căror intervale de execuţie se suprapun. Intervalele de execuţie pentru i şi j nu se suprapun, dacă una din următoarele două relaţii este satisfăcută:

$$ALAP_i + w(i) \leq ASAP_j$$
$$ALAP_j + w(j) \leq ASAP_i$$

A treia categorie de restricţii ajutătoare sunt prezentate în [NM96] şi se bazează pe noţiunile de nod dominator şi nod dominat, prezentate deja în paragraful 3.3 al acestui capitol. Pentru fiecare nod se identifică dominatorul său (dacă există unul) şi se impune restricţia ca timpul său de start să fie mai mare cel mult egal cu timpul de terminare al dominatorului la care se adaugă maximul sumelor timpilor de execuţie pentru nodurile cuprinse între dominator şi nod, care sunt legate aceleiaşi resurse.

$$T_i^s \geq T^s_{dominator} + w(\text{dominator}) + max(\Sigma\, w(j)),$$

unde $j \in$ unui drum de la dominator la i şi toate j-urile unei sume sunt legate de acelaşi procesor.

Deşi restricţia este foarte interesantă şi poate fi punctul de pornire pentru re-estimarea mai strânsă a ASAP-urilor pentru grafurile legate (fapt prezentat în paragraful 3.3 al capitolului), felul în care ea a fost formulată în [NM96] nu este de mare folos practic. Experimental am constat că aceste restricţii sunt de regulă mai slabe de cât ASAP, din cauză că nu se consideră drumul de la nodul dominator la setul nodurilor legate aceluiaşi procesor, respectiv drumul de la acest set la nodul dominat.

A patra categorie de restricţii ne aparţine şi se referă tot la variabilele pentru secvenţiere. Considerăm figura următoare:

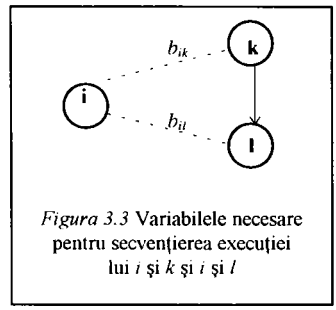


Figura 3.3 Variabilele necesare pentru secvenţierea execuţiei lui i şi k şi i şi l

Pentru exprimarea secvenţierii perechilor de noduri i şi k şi i şi l sunt necesare două variabile b_{ik} şi b_{il} (vezi figura 3.3). Variabilelor b_{ik} şi b_{il} le corespund patru combinaţii posibile de valori:

b_{ik}	b_{il}
0	0
0	1
1	0
1	1

Combinația haşurată implică faptul că i este executat după l, dar înaintea lui k. Aceast scenariu de execuție este însă invalid, din cauza dependenței între k şi l. Prin urmare pentru a elimina combinația de valori invalidă, impunem constrângerea ca:

$$b_{ik} \geq b_{il}$$

Pentru secvențe independente de procese mapate aceleiaşi resurse (figura 3.4), scrierea acestui tip de constrângeri poate fi existinsă. Pentru exemplul din figura 3.4 putem scrie următoarea secvență de inegalități:

$$\begin{array}{cccc} b_{i1j1} \geq & b_{i1j2} \geq & b_{i1j3} \geq & b_{i1j4} \\ \leq & \leq & \leq & \leq \\ b_{i2j1} \geq & b_{i2j2} \geq & b_{i2j3} \geq & b_{i2j4} \\ \leq & \leq & \leq & \leq \\ b_{i3j1} \geq & b_{i3j2} \geq & b_{i3j3} \geq & b_{i3j4} \\ \leq & \leq & \leq & \leq \\ b_{i4j1} \geq & b_{i4j2} \geq & b_{i4j3} \geq & b_{i4j4} \end{array}$$

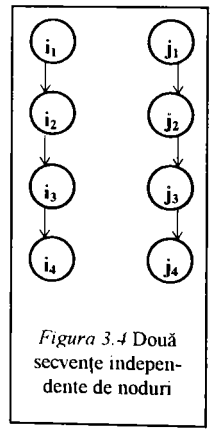


Figura 3.4 Două secvențe independente de noduri

Relațiile pe linii sunt determinate de precedențele între procesele j_k, $k = 1,n$, iar inegalitățile pe coloane apar pentru relațiile de precedență între procesele i_k, $k = 1,n$.

Experimentarea planificării cu ILP

Experimentele noastre pentru folosirea LP în planificarea grafurilor legate au inclus proiectarea şi implementarea unei program care plecând de la graful şi modul său de legare, generează automat modelul ILP şi apoi apelează `lp_solve` [BE92] pentru rezolvarea acestuia.

Pentru a mări viteza de rezolvare a modelului (care constituie aspectul critic al metodei) am folosit câteva din artificiile raportate în literatura de specialitate [AC95][MB95][NM96][BE96]. De exemplu, am verificat dacă este mai bine din punct de vedere al timpului de rezolvare să descriem problema ca şi ILP sau MILP (ambele variante sunt posibile). Deasemenea, am folosit diferite valori pentru infinit şi am considerat diferite secvențe de prelucrare a variabilelor din model.

Oricum rezultatele experimentale nu au fost de loc spectaculoase, timpii de rezolvare ai modelelor fiind mult prea mari pentru scopurile noastre de proiectare. Următorul tabel prezintă experimente efectuate pentru grafuri mici (cu numărul de noduri în jur de 40) folosind o stație SUN10. Pentru grafuri puțin mai mari, timpii de rezolvare au devenit prohibitiv de lungi. În concluzie, aceste experimente ne îndreptățesc să considerăm (cel puțin pentru moment) programarea liniară ca și neadecvată scopurilor noastre.

	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6
Număr de noduri în graf	42	37	41	33	38	37
Număr de variabile	211	202	129	90	110	111
Număr de ecuații	125	118	220	150	185	180
Timp de execuție pentru ILP (s)	12.95	12.0	9.50	9.20	7.80	8.30
∞ "=" 600000	12.80	12.10	9.50	9.40	7 80	8.40
∞ "=" 10000	12.90	12.30	9.40	9.30	7.80	8.30
Noduri dominate	12.95	12.25	9.60	9.35	7.80	8.30
MILP + ASAP + ALAP	12.90	12.10	9.60	9.30	7.80	8.30
Constrângeri pt. var. de secvent.	12.40	11.50	8.30	9.20	7.40	6.50
Achatz + Marwedel	foarte mare	foarte mare	foarte mare	foarte mare	foarte mare	foarte mare

Coloanele tabelului corespund la cele șase grafuri experimentate. Liniile unu, doi și trei indică caracteristicile modelelor ILP generate, adică numărul de noduri în graf, numărul variabilelor folosite în model și numărul ecuațiilor și al inecuațiilor. Timpii de rezolvare pentru ILP sunt prezentați în linia patru. Liniile de la cinci la zece prezintă timpii de rezolvare, când suplimentar s-au folosit anumite recomandări de optimizare prezentate în literatura de specialitate. Liniile cinci și șase sunt pentru situații când ∞ a fost "aproximat" prin valori mari. Linia șapte este pentru modelul ILP la care s-au adăugat constrângerile pentru nodurile dominate. Linia opt rezolvă modelul ca MILP și folosește suplimentar constrângerile pentru ASAP și ALAP. Rândul nouă este pentru modelul cuprinzând constrângerile noastre pentru variabilele de secvențiere. Linia zece folosește recomandările lui Marwedel și Achatz [AC95][MB95].

3.6 Planificarea activităților în graf prin metode euristice

Planificarea bazată pe liste (*list scheduling*) a devenit metoda euristică clasică de planificare nepreemptivă. Nodurile grafului au atașate priorități, care sunt apoi folosite pentru rezolvarea situațiilor de conflict, când la același moment, mai multe procese sunt pregătite pentru execuție pe aceeași resursă. Din setul proceselor pregătite pentru execuție este selectat cel cu prioritatea cea mai mare. Rolul priorităților este de a modela importanța nodului în satisfacerea criteriului de planificare, astfel că nodurile cu prioritatea mare (importanță mare) sunt planificate înaintea celor cu prioritate mică (importanță mică).

"Piatra de încercare" pentru planificarea bazată pe liste o constituie definirea priorităților. Acestea trebuie să realizeze un compromis între simplitate, astfel încât să fie calculate repede, și capacitatea lor de găsi soluții finale cât mai apropiate optimelor, ceea ce implică folosirea a cât mai multă informație.

Prima formă a algoritmului planificării bazate pe liste, numită și planificare *statică*, calculează o singură dată prioritățile, înainte de a începe planificarea nodurilor. Acestea sunt folosite apoi în interiorul unui ciclu pentru a stabili secvența de execuție a nodurilor.

```
void planificare_statică (graf G) {
      for (∀ nod ∈ G)
         calculează prioritatea nodului;
      while (∃ noduri neplanificate ∈ G)
         selectează și planifică nodul ready cu prioritatea
         cea mai mare;
```

Deobicei prioritatea nodurilor este definită pe baza atributului numit *drum critic* (*critical path*), pe care-l notăm cu DC [KA96]. Drumul critic este cel mai lung drum în graf, lungimea sa fiind calculată pe baza timpilor de execuție a nodurilor ce formează drumul. Pentru grafuri nelegate, valoarea DC determină potențial timpul total de execuție al planificării, fiind de fapt limita inferioară a acestuia.

DC nu are un caracter static, pentru că el se modifică pe parcursul planificării [KA96]. Astfel, pentru a obține planificări apropiate optimului se recomandă recalcularea DC-ului și a priorității nodurilor, de fiecare dată, după planificarea unui nod [KA96]. În acest mod, obținem a doua variantă a planificării bazată pe liste, variantă numită *dinamică*.

```
void planificare_dinamică (graf G) {
      while (∃ noduri neplanificate ∈ G) {
         for (∀ nod neplanificat ∈ G)
            calculează prioritatea nodurilor;
         selectează și planifică nodul ready cu prioritatea
         cea mai mare;
      }
```

În continuare, facem o scurtă prezentare a celor mai populare priorităţi menţionate în literatura de specialitate. Menţionăm că aceste priorităţi au fost menite să rezolve împreună atât legarea proceselor, cât şi planificarea lor.

Prioritatea cea mai simplă este cea folosită în *metoda drumul critic* (*critical path method*) [CO76]. Prioritatea unui nod este egală cu lungimea celui mai lung drum al său la nodul sfârşit al grafului. Această lungime este numită şi *nivelul* nodului.

CP/MISF (*critical path/most immediat successor first*) [KN84] este o îmbunătăţire a metodei precedente. Observaţia de la care porneşte CP/MISF este că dacă există mai multe taskuri situate pe acelaşi nivel, atunci prioritatea nu poate fi determinată în mod unic. În aceste cazuri, selecţia nodurilor de egală prioritate este arbitrară, deşi rezultatele planificării depind esenţial de ordinea de execuţie a taskurilor [CO76]. CP/MISF încorporează rezultatele teoretice din [CG72] şi redefineşte prioritatea pentru nodurile aceluiaşi nivel: pentru două taskuri cu acelaşi nivel, acela cu mai mulţi descendenţi direcţi are prioritatea mai mare.

Mobilitatea [WG90] unui task este diferenţa între ALAP-ul şi ASAP-ul său. Cu cât mobilitatea unui task este mai mică, cu atât prioritatea sa este mai mare. Taskurile cu mobilitatea 0 aparţin drumului critic. Autorii indică *mobilitatea relativă* ca şi o altă prioritate egală, cu rezultatul împărţirii mobilităţii unui task la timpul său de execuţie.

Urgenţa [GD92] este calculată în mod dinamic pe parcursul planificării. Dacă pasului curent din algoritmul de planificare îi corespunde momentul t, atunci prioritatea unui task este dieferenţa între ALAP-ul său şi t. Cu cât această diferenţă este mai mică, cu atât prioritatea taskului este mai mare.

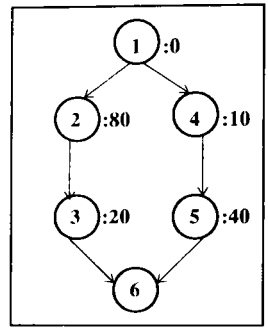
Metoda drumului critic modificat [WG90] (*modified critical path*) asociază fiecărui task o prioritate egală cu mulţimea formată din ALAP-ul său şi cele ale succesorilor săi imediaţi. Priorităţile taskurilor sunt ordonate funcţie de ordinea lexicografică a mulţimilor de ALAP-uri asociate.

Algoritmul *Edge-Zeroing* [SA89] ordonează taskurile funcţie de costul arcelor incidente la ele. La fiecare pas este selectată perechea de taskuri, care sunt interconectate prin arcul cu costul cel mai mare. Dacă lungimea planificării nu creşte, atunci cele două taskuri sunt legate aceluiaşi procesor, iar în caz contrar ele sunt plasate la procesoare diferite. Taskurile legate aceleiaşi resurse sunt planificate funcţie de nivelul lor.

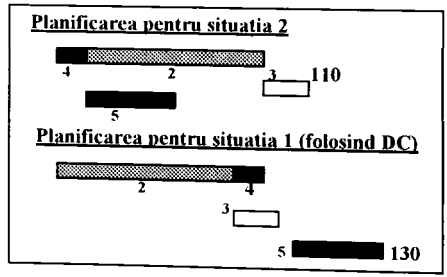
3.6.1 Planificarea grafurilor alocate folosind euristici bazate pe liste

Din prezentarea principalelor priorităţi folosite pentru planificare rezultă că acestea sunt definite pe baza DC. Acest lucru este perfect justificat pentru grafuri nelegate, deoarece DC reprezintă secvenţa cea mai constrânsă a grafului. Raportând nodurile la DC obţinem o măsură a gradului lor de constrângere. Pentru grafurile legate, din cauza secvenţializării execuţiei nodurilor mapate aceleiaşi resurse, DC (în formula în care a

fost definit) nu mai are relevanţă. Următorul exemplu foarte simplu motivează afirmaţia noastră



Timpii de execuţie ai nodurilor sunt indicaţi în figură, iar nodurile 2 şi 4 sunt executate de aceiaşi resursă. Nodurile 3 şi 5 sunt legate la două resurse diferite, ambele diferite de cele pentru 2 şi 4. Drumul critic al grafului este format din nodurile 1, 2, 3 şi 6. Dacă planificăm nodurile folosind priorităţile definite pe baza drumului critic (de exemplu CP/MISF), obţinem următoarea planificare. După 1, nodul 2 este planificat primul, la terminarea sa (momentul 80) sunt lansate nodurile 4 şi 3. 3 se încheie la 80 + 20 = 100, iar 4 la 80 + 10 = 90. La momentul 90 porneşte nodul 5, care se încheie la 130. Prin urmare, planificând nodurile folosind DC obţinem un timp total de execuţie de 130. Dacă am folosi o altă prioritate care să lanseze primul pe 4, apoi pe 2, timpul total de execuţie care se obţine este de 110. Explicaţia acestei diferenţe (după cum se vede şi din figura următoare) este că în primul caz concurenţa sistemului este mică, în timp ce în al doilea, ea este cea maximă permisă pentru această aplicaţie.



Acest mic exemplu sugestionează că pentru grafuri legate noţiunea de drum critic trebuie reinterpretată sau este necesară definirea priorităţilor folosind alte atribute de cât DC. În continuare, argumentaţia noastră este axată pe reinterpretarea noţiunii de DC şi se bazează pe următorul desenului din figura 3.5.

În figura 3.5, prin haşuri am indicat modul de legare a nodurilor din graf pe resurse. Nodurile *P1* şi *P2* sunt unite de rădăcina grafului prin drumuri (indicate prin arce) şi sunt plasate aceluiaşi procesor *M*. Procesorul *M* este diferite de procesorul hardware, astfel că cele două noduri mapate lui trebuie executate secvenţial. Pentru nodul *j*, $Succ_j$ este mulţimea tuturor drumurilor de la *j* la nodul final al grafului. Un drum $path_{jk} \in Succ_j$ este compus din două părţi, Δ'_{jk} formată din succesorii lui *j* situaţi pe aceeaşi

resursă cu j şi Δ_{jk}, care începe de la primul succesor legat altei resurse de cât j. $First_{jk}$ indică primul succesor al lui j, aparţinând lui $path_{jk}$ şi situat pe altă resursă de cât j. Notăm cu $\Delta_j = max(\Delta_{jk})$, pentru toate $path_{jk} \in Succ_j$ şi cu $First_j$ pe $First_{jk}$ al lui $path_{jk}$, care stabileşte Δ_j. Pentru $First_j$ indicăm prin Δ'_j nodurile situate între j şi $First_j$, noduri care sunt legate aceluiaşi procesor ca şi j. T_j^s reprezintă timpul când execuţia lui j începe, iar $w(j)$ este timpul cât durează j.

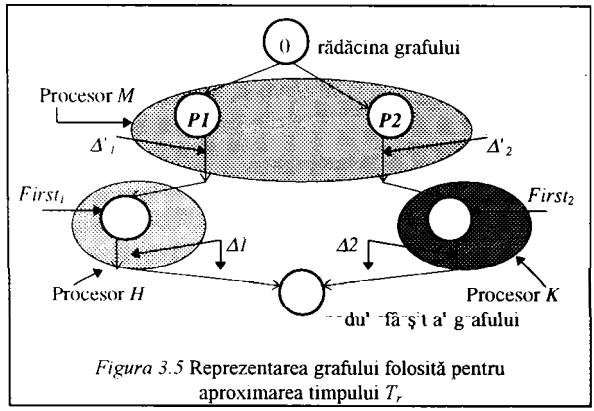


Figura 3.5 Reprezentarea grafului folosită pentru aproximarea timpului T_r

Timpul total de execuţie al grafului este definit ca fiind timpul pentru lansarea în execuţie a nodului sfârşit al grafului. Bineînţeles că acest timp depinde fundamental de secvenţa de planificare a nodurilor. Pentru succesorii lui $P1$, limita inferioară a timpului de execuţie este:

$$T^{succ}_{P1} \geq T_{P1}^{s} + w(P1) + \Delta_1' + \Delta_1$$

şi în mod asemănător, pentru succesorii lui $P2$ limita inferioară este:

$$T^{succ}_{P2} \geq T_{P2}^{s} + w(P2) + \Delta_2' + \Delta_2$$

Considerând doar nodurile $P1$ şi $P2$, execuţia grafului nu se încheie mai repede de

$$T_r = max(T^{succ}_{P1}, T^{succ}_{P2})$$

Dacă nodurile $P1$ şi $P2$ sunt simultan ready, atunci execuţia lor trebuie secvenţializată Folosind formulele stabilite mai sus pentru T^{succ}_{P1} şi T^{succ}_{P2} stabilim ordinea de execuţie ce conduce la un T_r cât mai mic. În primul caz considerăm $P1$ executat înaintea lui $P2$, iar în cazul doi $P2$ este planificat înaintea lui $P1$. Pentru fiecare situaţie calculăm T_r-ul corespunzător, iar în final alegem acea ordine de planificare al lui $P1$ şi $P2$ care stabileşte T_r-ul cel mai mic.

În primul caz, considerăm că $P2$ este executat după $P1$ şi succesorii săi din Δ'_1. Timpii estimaţi pentru execuţia succesorilor lui $P1$ şi $P2$ sunt:

$$T^{succ}_{P1} \geq T_{P1}^{s} + w(P1) + \Delta_1' + \Delta_1$$
$$T^{succ}_{P2} \geq T_{P1}^{s} + w(P1) + \Delta_1' + w(P2) + \Delta_2' + \Delta_2$$

Asemănător, dacă $P2$ şi succesorii săi din Δ'_2 sunt executaţi înaintea lui $P1$, timpii estimaţi pentru execuţia succesorilor lui $P1$ şi $P2$ sunt:

$$T^{succ}_{P1} \geq T_{P2}^{s} + w(P2) + \Delta_2' + w(P1) + \Delta_1' + \Delta_1$$
$$T^{succ}_{P2} \geq T_{P2}^{s} + w(P2) + \Delta_2' + \Delta_2$$

Deoarece am considerat că atât *P1*, cât şi *P2* sunt ready pentru execuţie, rezultă că $T_{P1}^{s} = T_{P2}^{s}$. În mod evident $T^{succ}{}_{P1} \geq T^{succ}{}_{P1}$ şi $T^{succ}{}_{P2} \geq T^{succ}{}_{P2}$, astfel că comparaţia trebuie făcută între $T^{succ}{}_{P1}$ şi $T^{succ}{}_{P2}$. Dacă $\Delta_1 \geq \Delta_2$, atunci $T^{succ}{}_{P1} \geq T^{succ}{}_{P2}$ şi este mai convenabil să planificăm *P1* înaintea lui *P2*, iar dacă $\Delta_1 < \Delta_2$, atunci $T^{succ}{}_{P1} < T^{succ}{}_{P2}$ şi planificăm pe *P2* înaintea lui *P1*.

Această analiză ne duce către următoarea regulă de planificare: dintre două noduri ready la un moment dat pe aceeaşi resursă, primul este planificat acela care are un drum mai lung, calculat începând cu primul nod legat altei resurse de cât nodul. Deoarece această metodă se bazează pe CP/MISF dar reinterpretează noţiunea de lungime a drumului critic am denumit-o CP/MISF-ML (*CP/MISF modified level*).

3.6.2 Exemplu de planificare

În continuare, prezentăm un exemplu ce compară planificările obţinute folosind priorităţile definite prin urgenţă şi CP/MISF-ML. Deşi nu putem afirma superioritatea priorităţii noastre în raport cu urgenţa doar pe baza unui singur exemplu, el indică un caz relativ comun în care folosind CP/MISF-ML obţinem o planificare semnificativ mai scurtă de cât folosind urgenţa.

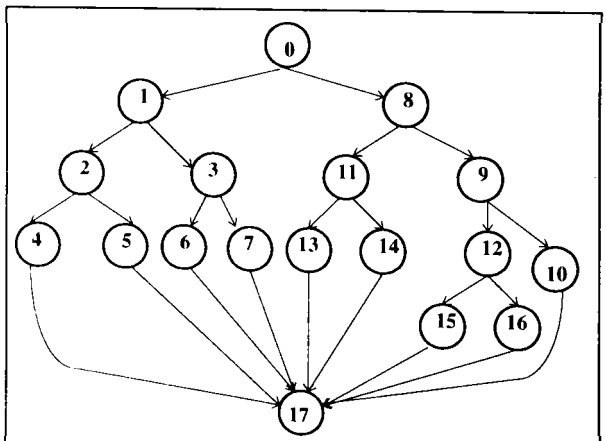


Figura 3.6 Exemplu de graf folosit pentru compararea planificării folosind urgenţa şi CP/MISF-ML

Descriem pe scurt structura de dependenţe a grafului şi timpii de execuţie asociaţi nodurilor. Nodul 0 analizează în timp linear fiecare din cele 810 elemente ale sale, şi transmite 320 de elemente către 1, iar restul către 8. 1 prelucrează în timp liniar elementele sale şi comunică 160 la 2 şi 160 la 3. Fiecare din 2 şi 3 prelucrează în timp liniar elementele lor şi transmit câte 80 de elemente la 4, 5, 6 şi 7. 8 recepţionează 490 de elemente de la 0, le procesează în timp liniar şi comunică 160 de valori lui 11 şi 330 lui 9. 11, după o prelucrare în timp liniar a celor 160 de valori, transmite 80 lui 13 şi 80

lui 14. În mod similar, 9 îşi procesează elementele în timp liniar şi transmite 180 de valori lui 12 şi 150 lui 10. 9 îşi distribuie elementele în mod egal lui 15 şi 16. 4, 5, 6, 7, 13, 14, 15, 16 şi 10 execută prelucrări ce depind cu pătratul numărului de valori recepţionate. Următorul tabel concentrează timpii de execuţie asociaţi proceselor, precum şi modul de legare a nodurilor pe resurse. Arhitectura ce execută graful este formată din trei procesoare software (*S1*, *S2* şi *S3*) şi o magistrală. Toate comunicările durează 10 unităţi de timp, indiferent de cantitatea de informaţie transmisă.

0	810	S1	6	6400	S2	12	180	S3
1	320	S1	7	6400	S2	13	6400	S3
2	160	S2	8	490	S1	14	6400	S3
3	160	S2	9	330	S1	15	8100	S3
4	6400	S2	10	22500	S1	16	8100	S3
5	6400	S2	11	160	S3	17	0	-

Pe baza grafului aplicaţiei şi a modului de legare a nodurilor pe procesoare se construieşte graful folosit pentru planificare (figura 3.7). Acest graf conţine suplimentar nodurile pentru comunicările între noduri mapate la procesoare distincte şi pentru a le evidenţia, le-am reprezentat în desen prin dreptunghiuri.

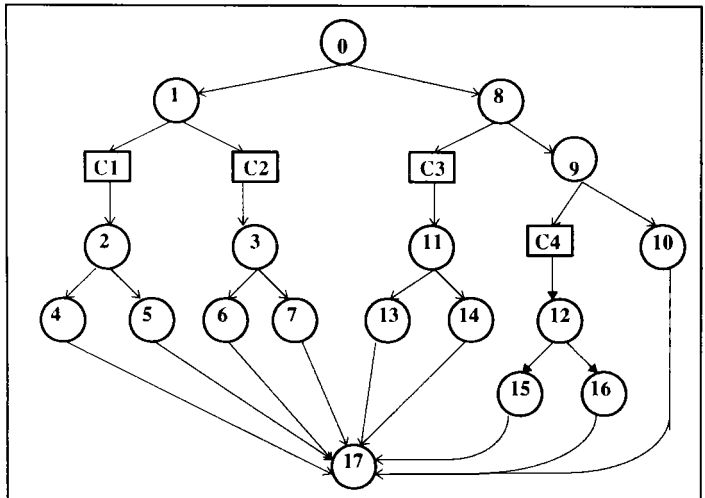


Figura 3.7 Graful folosit pentru planificare

Planificarea folosind urgenţa, planifică întâi nodurile aparţinând drumului critic, iar apoi pe restul. După cum arată diagrama Gantt din figura 3.8, concurenţa execuţiei este mică, iar timpul total de execuţie al grafului este de 50380.

Figura 3.9 este diagrama Gantt pentru planificarea folosind CP/MISF-ML. Concurenţa execuţiei este mult îmbunătăţită, ceea ce reduce semnificativ timpul total de execuţie al grafului care este 30970 (cu 62% mai mic de cât în cazul precedent).

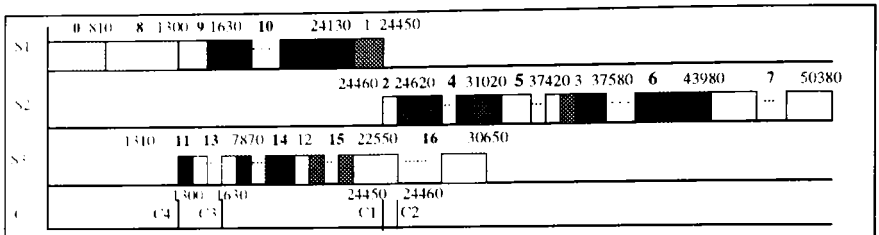


Figura 3.8 Planificarea grafului din figura 3.7 folosind urgenţa

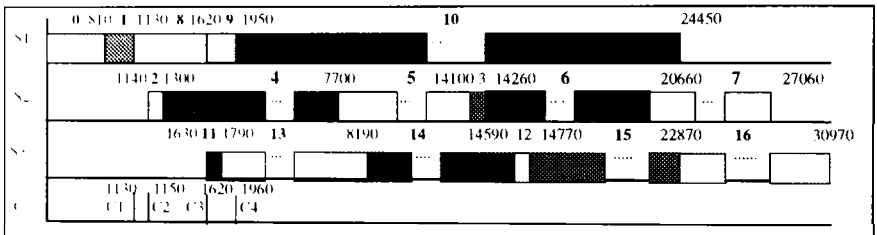


Figura 3.9 Planificarea grafului din figura 3.7 folosind CP/MISF-ML

3.6.3 Experimentarea lui CP/MISF-ML

Experimentarea euristicii CP/MISF-ML a folosit 1250 de grafuri. S-au generat câte 250 de grafuri pentru fiecare din dimensiunile de 20, 40, 75, 130 şi 200 de noduri. Structura grafurilor (modul de interconectare a nodurilor) a fost atât aleatoare, cât şi regulată (arbori sau lanţuri de procese). Structurilor regulate li s-au adăugat un număr de conexiuni aleatoare. Timpii de execuţie ataşaţi nodurilor din graf au avut distribuţie uniformă sau exponenţială. Arhitecturile considerate pentru exeprimente au fost format din 1 ASIC, între 1 şi 10 procesoare şi între 1 şi 3 magistrale. O parte a grafurilor foloseşte o mapare aleatoare a proceselor pe procesoare şi a arcelor pe magistrale. A doua variantă de legare este mai inteligentă, în sensul că secvenţe de procese sunt executate de acelaşi procesor, astfel încât să minimizăm comunicările între procesoarem, iar încărcarea resurselor să fie uniformă. Experimentele acestui capitol s-au făcut folosind staţii SUN SPARC 20.

Prima concluzie se referă la calitatea planificărilor obţinute cu CP/MISF-ML. Am determinat variaţiile relative în procente ale planificărilor cu CP/MISF-ML faţă de cele optime. Planificările optime au fost produse cu BB [EL97]. Când optimele nu au putut fi obţinute în 180 de secunde, atunci referinţa a fost cea mai scurtă planificare obţinută prin una din cele patru metode. Variaţia medie în procente a planificărilor prin CP/MISF-ML faţă de optime, calculată pentru cele 1250 de grafuri, este 2.10%. Ca şi

termen de comparaţie, folosind ca priorite urgenţa a rezultat 2.85% şi 2.82% pentru drumul critic. Deviaţile relative pentru grafuri individuale pentru CP/MISF-ML sunt cuprinse în intervalul 0 şi 21.10%. Următorul tabel sintetizează variaţiile medii relative ale alungirilor faţă de optime pentru CP/MISF-ML, drum critic şi urgenţă.

	40		75		130		200	
	medie	maxim	medie	maxim	medie	maxim	medie	maxim
CP/MISF-ML	2.26	18.18	2.16	19.01	2.20	13.73	1.81	21.10
drum critic	3.59	22.22	2.97	19.01	2.52	13.73	2.22	21.10
urgenţă	3.83	22.22	2.91	19.15	2.51	13.73	2.17	21.10

Figura 3.10 prezintă variaţia timpului mediu de execuţie pentru CP/MISF-ML funcţie de numărul de noduri ale grafului. Figura arată comparativ şi variaţia timpului mediu de execuţie atunci când planificarea foloseşte ca priorităţi drumul critic şi urgenţa. Timpii de execuţie care se obţin pentru CP/MISF-ML sunt foarte mici. De exemplu, ei sunt mai mici de 0.0085 secunde pentru grafuri cu 200 de noduri.

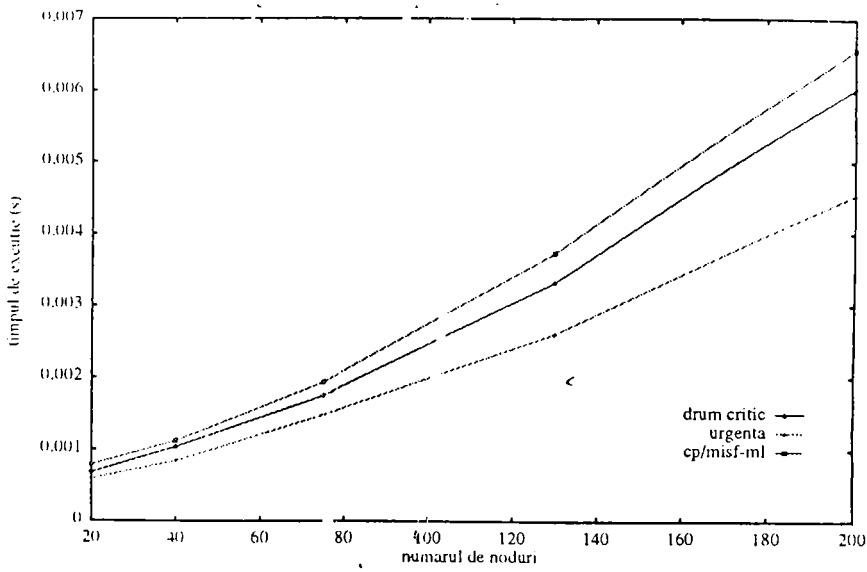


Figura 3.10 Variaţia timpului de execuţie cu numărul de noduri

Încheiem paragraful curent cu prezentarea principalelor cauze ce au determinat alungirea planificărilor obţinute cu CP/MISF-ML faţă de optime. Considerăm această identificare utilă pentru proiectarea viitoare a unor euristici superioare, ce vor trata inteligent aceste situaţii. Cazurile le-am stabilit în urma analizei câtorva zeci de grafuri, pentru care alungirile planificării euristice erau semnificativ mai mari ca şi cele optime.

Examinând situaţiile pentru care CP/MISF-ML găseşte o soluţie mai slabă ca şi optimul, am izolat următoarele cauze

1) *Necesitatea introducerii de intervale "idle"*: pe parcursul planificării CP/MISF-ML se lansează în execuție un nod cu prioritate mică, dar scurt timp după aceasta devine ready un nod cu prioritate mare. Acesta din urmă este întârziat până când se termină nodul mai puțin prioritar. Algoritmul de planificare optimă lasă un interval de timp resursa neocupată, iar nodul prioritar este executat înaintea celui mai puțin prioritar. Figura 3.11 indică o astfel de situație.

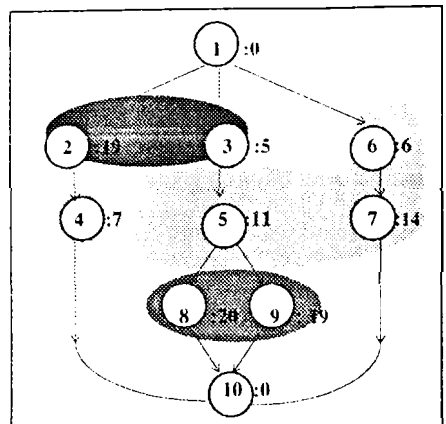


Figura 3.11 Exemplu ce ilustrează importanța timpilor *idle* în obținerea unei planificări scurte

Nodul 7 devine ready cu 1 unitate de timp mai repede ca și 5. 5, deși are prioritate mai mare, trebuie să aștepte 13 unități de timp terminarea lui 7. În mod optim, resursa este lăsată 1 unitate de timp nefolosită, după care este planificat 5, iar apoi 7. În acest mod gradul de concurență al sistemului crește, deoarece 7 este executat concurent cu 8 și 9.

Un caz intersant, în care soluția găsită de CP/MISF-ML este mai slabă de cât optimul, dar și de cât restul euristicilor, este următorul. La un moment dat există mai multe noduri ready pe același resursă, toate având aceeași prioritate și același număr de succesori direcți. În această situație, alegem arbitrar următorul nod de planificat. Dacă la scurt timp pe resursa respectivă devine ready un nod cu prioritate mare, atunci el este întârziat funcție de timpul de execuție al nodului selectat arbitrar. În cazurile în care CP/MISF-ML s-a comportat mai slab de cât restul euristicilor, datorită hazardului, a fost ales nodul mai puțin favorabil, în sensul că întârzia mai mult nodul cu prioritate mai mare. Dacă s-ar fi ales celălalt nod, timpul de "așteptare" al nodului ready cu prioritate mai mare ar fi fost mai scurt și deasemenea timpul total de execuție al grafului ar fi fost mai mic.

Graful din figura 3.12 indică o astfel de situație.

Nodurile 7 și 3 au aceeași prioritate și același număr de succesori. CP/MISF-ML îl alege arbitrar pe 3, iar apoi pe 7, ceea ce crește timpul de întârziere pentru nodul 6, care are o prioritate mai mare. Dacă s-ar planifica 7 înaintea lui 3, atunci în timpul

execuției lui 7, 6 ar deveni ready şi imediat după terminare lui 7, el ar putea fi lansat în execuție. Astfel, timpul de aşteptare a lui 6 ar fi cu 6 unități de timp mai mic (exact cât este timpul de execuție al lui 3).

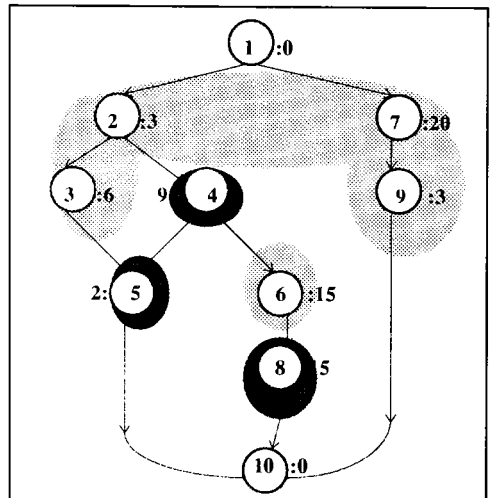


Figura 3.12 Alungirile planificării datorită nodurilor de egală prioritate

Regula de planificare, ce se poate folosi în situația în care mai multe noduri cu aceeaşi prioritate sunt ready, este aceea de a selecta nodul, care determină un timp de aşteptare mai mic pentru un nod mai prioritar, ce devine ready între timp pe aceeaşi resursă. Dacă nodul prioritar devine ready la un timp mai scurt de cît timpul de execuție al oricăruia din nodurile cu egală prioritate, atunci selectăm pentru execuție pe cel cu timpul cel mai scurt. Dacă cel prioritar devine ready la un moment ce depăşeşte suma timpilor de execuție a nodurilor egal prioritare, atunci nu contează ordinea de planificare a acestora. Dacă însă nodul prioritar devine ready la un timp mai scurt de cât timpul de execuție cel mai lung al nodurilor ready şi mai lung de cât cel mai scurt, atunci este mai favorabil să îl planificăm primul pe acela cu timpul cel mai lung.

2) Apariția de concurențe "false": ilustrăm acest caz prin două exemple. Ambele cuprind secvențe "concurente" de noduri, dar care nu pot fi executate concurent din cauza modului lor de mapare pe resurse. Nodurile mapate aceleiaşi resurse trebuie executate secvențial.

Exemplul din figura 3.13 introduce un caz foarte simplu, în care drumul critic (şi nici una din euristicile bazate pe el) nu duce la soluția optimă. Nodul 2 are prioritatea mai mare ca şi 5, indiferent de prioritatea pe care o folosim, CP/MISF, CP/MISF-ML sau urgență. Planificând procesele conform priorităților lor, obținem un timp total de execuție pentru graf de 44 unități de timp. Dacă am planifica pe 5 înaintea lui 2,

lungimea totală care se obţine este 38. Explicaţia este că în cazul al doilea concurenţa sistemului este superioară, existând mai multe activităţi ce se desfăşoară în paralel.

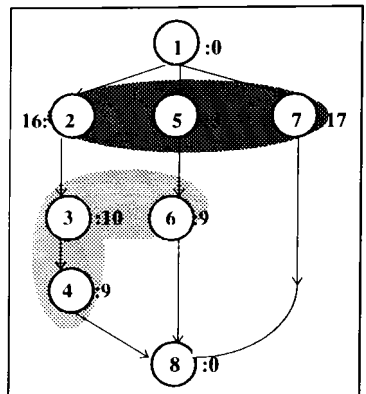


Figura 3.13 Implicaţia concurenţelor false asupra rezultatelor planificării cu CP/MISF-ML

CP/MISF-ML încurajează creşterea suprapunerilor de activităţi în sistem. În acest caz, planificând pe 2 înaintea lui 5, favorizăm "suprapunerea" lui 3 şi 4 cu 5. Datorită legării ce mapează toate cele trei noduri aceluiaşi procesor, de fapt scădem gradul de concurenţă în graf. Consecinţa este că 6 trebuie să aştepte terminarea lui 3 şi 4, iar timpul total de execuţie creşte corespunzător.

Acest exemplu ne duce la ideea că CP/MISF-ML se comportă slab pentru situaţii în care lanţuri independente de procese concurente sunt mapate aceleiaşi resurse.

Al doilea caz este prezentat în figura 3.14.

Calculând priorităţile nodurilor conform CP/MISF-ML, 2 are prioritatea mai mică ca şi 4. Planificând 4 înaintea lui 2, se va aglomera şi mai mult resursa constrânsă haşurată cu culoare închisă, ceea ce în final duce la un timp total de execuţie lung. Examinând modul de legare pentru succesorii lui 2, constatăm că cinci sunt alocaţi resursei constrânse. Succesorii lui 2 nu se pot termina mai repede de cât timpul necesar pentru executarea nodurilor de pe resursa constrânsă, timp egal cu $5 + 5 + 5 + 10 + 5 = 30$. Deoarece timpul necesar executării succesorilor lui 2 este mai mare ca cel pentru succesorii lui 4, ar trebui să planificăm pe 2 înaintea lui 4. Încărcarea resursei constrânse ar fi mai relaxată, iar timpul total de execuţie al grafului ar fi mai mic.

Acest exemplu arată că atunci când stabilim timpul necesar pentru executarea succesorilor unui nod, este benefic să studiem simultan toate drumurile ce pornesc de la nod şi nu fiecare drum pe rând. Analizând împreună toate drumurile putem aprecia întârzierile ce apar din cauza modului de legare a nodurilor pe aceleaşi resurse. Acest aspect este determinant pentru grafurile "late", cu multe noduri de pe acelaşi nivel grupate pe aceleaşi resurse.

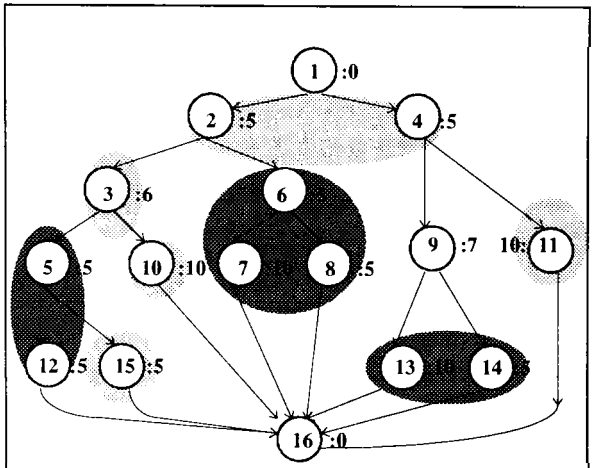


Figura 3.14 Implicaţia concurenţelor false asupra rezultatelor planificării cu CP/MISF-ML

3.7 Concluzii

Capitolul 3 prezintă experienţa noastră relativă la problema planificării grafurilor legate. Planificările depinde atât de timpii de execuţie şi relaţiile între nodurile unui graf, cât şi de felul în care ele sunt legate resurselor din arhitectură. De aceea, orice decizie legată de execuţia (ordinea sau timpul) nodurilor legate trebuie să surprindă ambele elemente menţionate.

Studiul nostru particularizează pentru grafuri legate mărimile ASAP, ALAP şi drum critic. Execuţia fiecărui grup de activităţi mapate aceluiaşi procesor programabil sau magistrală trebuie secvenţializată. Un succesor al grupului nu poate începe mai repede de timpul dat de suma între momentul când toate nodurile grupului îşi încheie execuţia secvenţializată şi cel mai scurt drum maxim de la un nod al grupului la nodul în cauză. Atunci când calculăm drumul de la un nod din grup la nodul curent, obţinem o valoare mai apropiată de cea reală, dacă considerăm simultan *toate* drumurile între cele două noduri şi nu fiecare drum în parte. Din cauza legării nodurilor drumurilor, pot rezulta alungiri ale timpului minim pentru execuţia tuturor drumurilor ce leagă cele două noduri. Drumul critic este definit ca şi ASAP-ul nodului sfârşit al grafului, iar ALAP-ul unui nod ca şi diferenţa între drumul critic şi timpul minim necesar pentru parcurgerea tuturor drumurilor care îşi au originea în nod. Din cauza legării nodurilor, pentru a obţine cât mai exact timpul minim este util să considerăm simultan toate drumurile de la un nod la nodul sfârşit. Argumentaţia este aceiaşi ca şi pentru ASAP.

Drumul critic poate fi interpretat ca şi limită inferioară pentru timpul de execuţie a unui graf legat. Această limită devine mai precisă, dacă repetăm raţionamentul lui Fernandez

şi particularizăm relaţiile sale pentru grafuri legate. Drumul critic surprinde în special caracteristicile nodurilor între care există relaţii de precedenţă (nodurile plasate pe acelaşi drum). Complementare acestor noduri, nodurile aparţinând la drumuri independente se pot influenţa reciproc prin modul în care ele sunt legate resurselor. Instanţierile formulelor lui Fernandez, pe care noi le propunem pentru grafuri legate, surprind aceste influenţe şi au rolul de a cuantiza alungirile drumului critic în graf datorate lor. Formulele reprezintă o metodă de estimare rapidă a timpului de execuţie a unui graf legat, fără a planifica explicit nodurile sale. Ele pot fi utile ca şi limită inferioară (lower bound) pentru implementarea unui algoritm de planificare optimă prin metoda *branch-and-bound*.

O altă metodă de planificare optimă este cea a programării liniare. Noi am proiectat şi implementat o componentă pentru generarea automată a modelului LP pentru un graf şi am experimentat diferite idei pentru reducerea timpului de rezolvare a modelului. Faptul că timpii au fost prohibitivi de mari chiar şi pentru grafuri cu un număr realist de noduri, ne-au determinat să considerăm (cel puţin pentru moment) metoda ca şi nepotrivită scopurilor noastre.

Noi propunem o euristică, CP/MISF-ML, pentru obţinerea rapidă a unor planificări apropiate optimelor. Euristicile din literatură folosesc atribute ce nu face nici o referire la legarea nodurilor. Ele încurajează executarea timpurie a nodurilor de pe drumul critic (dar în fond ce este drumul critic pentru grafuri legate?), dar nu ţin cont de caracteristici de bază ale sistemelor concurente, cum ar fi gradul de concurenţă rezultat. Acest fapt lungeşte adeseori timpul total de execuţie al grafului. CP/MISF-ML depinde de drumul critic definit considerând maniera de legare a nodurilor pe resurse şi încurajează suprapuneri cît mai mari ale execuţiilor. O planificare cu numeroase execuţii concurente are un timp total de execuţie mai scurt. De aceea, CP/MISF-ML se comportă foarte bine în situaţiile în care drumurile ce pornesc din acelaşi nod sau din noduri legate aceleiaşi resurse, sunt mapate pe elemente de procesare diferite. O astfel de mapare este absolut rezonabilă din punct de vedere practic, deoarece încurajează execuţia paralelă a activităţilor descrise în specificare ca fiind concurente. CP/MISF-ML a fost experimentat şi evaluat statistic, folosind un număr foarte mare de grafuri. Calitatea soluţiilor sale şi timpul său redus de execuţie sunt foarte încurajatoare pentru folosirea lui CP/MISF-ML în planificarea grafurilor legate. Totodată, am identificat cauzele cele mai frecvente, responsabile pentru abaterea soluţiilor sale faţă de optime. Rezolvarea acestor cauze şi dobândirea de experienţă nouă în problema planificării grafurilor legate vor permite, probabil, dezvoltarea unor metode noi, superioare de planificare.

Capitolul 4

Formalizarea problemei planificării grafurilor cu dependenţe de date şi de control

Rezumat

Capitolul 4 defineşte formal graful dependenţelor de date şi de control (GDC), precum şi problema planificării GDC-urilor legate. Specificarea este descrisă printr-un GDC în vederea co-sintezei hardware/software. La începutul capitolului motivăm necesitatea de a folosi instrucţiuni de control pentru a mări puterea de descriere a specificărilor şi acurateţea lor faţă de realitate. Folosirea GDC-urilor în proiectare permite obţinerea unor implementări eficiente la un preţ mai mic. Noi definim formal un GDC printr-un cvint-uplu ce cuprinde nodurile, arcele, structura alternativelor condiţionale, precum şi condiţiile şi valorile lor pentru care se selectează fiecare din alternative. Fiecare nod are ataşat un atribut pentru modelarea timpului său de execuţie. Un GDC trebuie să satisfacă un grup de restricţii, necesare pentru a păstra caracterul determinist al planificării, dar şi pentru a simplifica prezentarea noastră. Al treilea tip de restricţii ţine de limita modelării noastre. Noi am definit formal planificarea GDC-urilor legate ca fiind calcularea funcţiei timpilor de start pentru noduri, astfel încât funcţia să îndeplinească un set de constrângeri. Capitolul schiţează regulile de transformare ale unei specificări într-una echivalentă semantic, dar care să permită obţinerea directă a GDC-ului ataşat. Aceste reguli urmăresc ca specificarea transformată să evidenţieze dependenţele de date şi de control şi să fie la granularitatea de lucru dorită de proiectant. Rezultatele capitolului sunt folosite pentru fundamentarea algoritmilor noştri de planificare a GDC-urilor (capitolul 5), iar regulile de transformare reprezintă scheletul teoretic necesar pentru implementarea componentei ce generează GDC-ul unei specificări.

4.1 De ce dependenţe de control?

Capitolul 3 discută problema planificării proceselor şi a comunicărilor legate de resurse, pornind de la ipoteza că între acestea există doar dependenţe de date. Consecinţa acestui fapt este că pentru o execuţie a grafului, setul proceselor activate este întotdeauna acelaşi, deoarece singura cerinţă pentru ca un proces să fie activat este ca toţi predecesorii săi să fi fost deja executaţi.

Această ipoteză este nerealistă pentru majoritatea aplicațiilor reale. De regulă, felul acțiunilor executate de un sistem depinde de date de intrare sau calculate în timpul execuției, astfel că, nu toate activitățile modelate în graf sunt executate pentru o singură parcurgere a acestuia. Un exemplu foarte simplu este cel în care sistemul primește o valoare de intrare și funcție de ea, el execută fie $activitate_1$, fie $activitate_2$, dar nu ambele concomitent. Timpii de execuție și resursele necesare lui $activitate_1$ și lui $activitate_2$ sunt foarte diferiți, astfel că execuțiile sistemului diferă mult în cele două situații. Concluzia exemplului precedent este că algoritmii de proiectare nu trebuie să considere de cât activitățile ce corespund unei anumite situații (scenariu) de execuție a grafului.

După cum rezultă din scurta introducere făcută, modelul activităților între care există doar dependențe de date nu este acoperitor și pentru sisteme a căror funcționalitate depinde de mărimi ce devin disponibile doar pe parcursul execuției. Dependența funcționalității sistemului de date de intrare sau date calculate în timpul execuției este modelată prin *dependențe de control*.

În cazul limită putem considera soluția de implementare acoperitoare, ce execută toate activitățile din specificare. Aceasta va duce însă la implementări supradimensionate, menite să rezolve situații nerealiste din punct de vedere funcțional. Întru-cât scopul final al proiectării noastre este găsirea unei soluții ce satisface restricțiile de proiectare la un cost cât mai ieftin, bineânțeles, că această abordare nu poate să ne mulțumească.

Argumentația precedentă motivează necesitatea de a introduce în graful proceselor și comunicațiilor unui sistem, pe lângă dependențele de date și pe cele de control. Graturile rezultate le denumim **grafuri cu dependențe de date și de control** (GDC).

După cum am detaliat anterior, dependențele de control indică *circumstanțele* în care are loc executarea (inclusiv numărul de execuții) sau nu a *anumitor* procese și comunicări. Limbajele pentru specificare de sistem [EP94][VN95][GM93] folosesc instrucțiuni condiționale (*if*, *case*) și iterative (*for*, *while*, *loop*, etc.) pentru definirea fluxului de control. Secvențele de instrucțiuni din instrucțiunile de control sunt executate de un număr de ori (0, 1 sau mai multe ori) funcție de valoarea unor variabile sau expresii. În această lucrare, variabilele și expresiile folosite în controlarea fluxului de execuție sunt desemnate prin termenul *condiții*. Cercetările în domeniul limbajelor de programare [GO79][SC88] discută în detaliu impactul acestor construcții asupra semanticii programelor, dar aceste abordări se situează în afara scopului acestei teze.

Fără a reduce generalitatea problemei, din punctul nostru de vedere modelarea fluxului de control cuprinde următoarele două situații:

- Un proces calculează valoarea unei condiții, iar funcție de ea activează un set de activități sau altul. Pentru simplitate, considerăm că domeniul de valori al condiției cuprinde două valori, *adevărat* și *fals*, și funcție de ele sunt activate una din două alternative posibile, dar niciodată ambele. Această situație corespunde de fapt construcției *if*. Facem observația că construcția *case* generalizează *if*, când domeniul condiției cuprinde mai mult de două valori, funcție de care se selectează o alternativă dintr-o mulțime compusă din mai mult de două alternative. *case* poate fi redus la un număr de *if*-uri încuibate.

- Existența ciclurilor în specificare introduce în proiectare un factor de nedeterminism, din cauza dependențelor de date care intervin. *Ciclurile nelegate* (pentru care numărul maxim de iterații nu este cunoscut) nu pot fi planificate, din cauză că numărul lor de iterații depinde de mărimi disponibile doar în timpul execuției (mărimi de intrare sau calculate în timpul execuției). Cercetările în domeniu [CW94][PK89] indică cerința ca toate ciclurile din specificare să fie *legate* (să aibă precizat numărul lor maxim de iterații), ca fiind necesară pentru planificarea statică a activităților unei specificări. În acest caz, algoritmul de planificare ia în considerare cazul cel mai defavorabil, respectiv situația când toate iterațiile ciclurilor sunt parcurse. Problema planificării grafurilor cu cicluri legate este un subiect de cercetare în sine, care depășește cadrul acestei teze. Din punctul nostru de vedere, toate construcțiile ciclice au specificate numărul lor maxim de iterații, N, și ele sunt înlocuite în graf prin desfășurarea ciclului de N ori (*loop unrolling*). Prin această transformare, graful specificării este păstrat *aciclic*. Deși această soluție este o sursă de "explozie" a numărului de noduri în graf (mai ales pentu cicluri cu număr mare de iterații), ea este totuși folosită de numeroase abordări [CW94][PK89]. Menționăm că unul din algoritmii de planificare propuși în capitolul 5, realizează o desfășurare *implicită* a ciclurilor în timpul procesului de planificare, fără a genera noduri noi în grafuri. La această oră planificarea ciclurilor legate nu este o problemă definitiv rezolvată din punctul de vedere al problemei noastre de proiectare (acesta ar putea constitui subiectul unor cercetări ulterioare), dar presupunem că grafurile specificărilor sunt aciclice, eventualele lor cicluri fiind desfășurate.

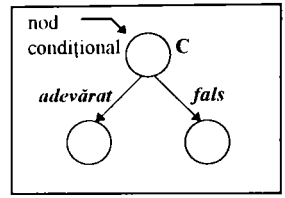
În finalul acestei introduceri concluzionăm că în contextul acestei teze, dependențele de control sunt introduse prin instrucțiuni *if*, prin care sunt selectate pentru execuție funcție de valoarea condițiilor una din două alternative posibile.

Capitolul 4 continuă cu prezentarea adnotările introduse în graful proceselor și al comunicărilor pentru a captura dependențele de control și apoi, identifică restricțiile pe care acest graf trebuie să le îndeplinească pentru a păstra caracterul determinist al planificării. În continuare, folosind acest model definim formal problema planificării grafurilor legate, cu dependențe de date și de control. În finalul capitolului indicăm un set posibil de reguli pentru transformarea specificării de intrare, descrisă într-un limbaj abstract pentru specificare de sistem, într-una echivalentă semantic și care să evidențiaze dependențele de control. Specificarea transformată este folosită pentru generarea directă a GDC.

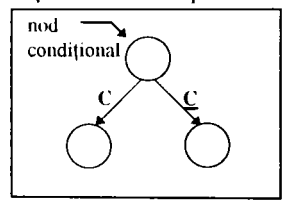
4.2 Graful dependențelor de date și de control

Revenim la notația folosită pentru grafurile cu dependențe de date, $G = (V,E)$, unde V este mulțimea nodurilor din graf și corespunde proceselor și comunicărilor între procese alocate la procesoare diferite, iar E este mulțimea arcelor și indică relațiile de precedență între noduri. Un arc de la n_i la n_j denotă că n_j nu își poate începe execuția până când n_i nu a fost terminat.

Suplimentar dependenţelor de date, trebuie să modelăm faptul că un proces poate calcula valoarea unei condiţii, folosită apoi pentru selectarea uneia din cele două alternative posibile. Nodurile ce calculează valoarea unei condiţii le numim *noduri condiţionale*. Fiecare nod condiţional are asociat în graf un atribut pentru indicarea condiţiei calculate, iar arcele spre alternativele condiţiei sunt etichetate cu valorile condiţiilor pentru care ele sunt selectate (figura următoare).



În cadrul acestei lucrări folosim şi următoarea reprezentare simplificată, echivalentă:



Deşi în graf nodul condiţional are două arce de ieşire, în timpul unei execuţii a grafului, este parcurs unul singur, deoarece cele două alternative se exclud reciproc.

Pentru a păstra rigurozitatea prezentării noastre, definim acum noţiunea de *alternativă* a unui nod condiţional, deşi am folosit-o deja pe parcursul prezentării.

Pentru nodul condiţional ce calculează condiţia **C**, definim *alternativa adevărat* ca fiind formată din acele noduri care pot fi executate dacă şi nu mai dacă valoarea lui **C** este *adevărat*. Reciproc, *alternativa fals* cuprinde acele noduri care pot fi executate dacă şi numai dacă **C** este *fals*. Pentru simplitatea expunerii, considerăm că fiecare alternativă are un singur nod de intrare şi unul singur de ieşire. Dacă specificare iniţială nu respectă această cerinţă, ea este transformată prin adăgarea a două noduri fictive, cu timpul de execuţie 0 şi a arcelor de legătură între nodurile fictive şi cele ale alternativelor.

Reciproc unui nod condiţional, în graf apare un nod special denumit *join*, având rolul marca sfârşitul construcţiei condiţionale (implicit şi sfârşitul celor două alternative). Din punct de vedere formal, definim nodul join corespunzător unui nod condiţional, ca fiind nodul care satisface următoarele două proprietăţi:
1) Între oricare din nodurile celor două alternative ale nodului condiţional şi nodul în cauză există un drum.
2) Nodul nu are nici un predecesor cu proprietatea 1.

Activarea nodului join nu depinde de valoarea condiţiei corespunzătoare, iar considerarea sa pentru execuţie începe cu momentul în care unul din predecesorii săi imediaţi a fost executat. Nodurile join sunt noduri fictive în sensul că nu indică

activități în graf, timpul lor de execuție este 0 și sunt marcate grafic în graf prin dreptunghiuri (figura 4.1).

În concluzie, unei instrucțiuni *if* îi corespunde în graf următoarea construcție:

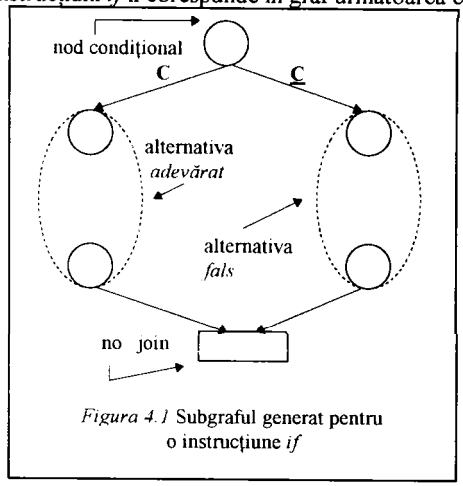


Figura 4.1 Subgraful generat pentru o instrucțiune *if*

Din punct de vedere formal, un graf cu dependențe de date și de control este notat ca cvint-uplul:

$$(V, E, condition, joinnode, alternative),$$

elementele sale având următoarele semnificații:

- V este mulțimea nodurilor în graf. Fiecare nod indică o activitate în graf și are asociat un timp de execuție. $w(n_i)$ este timpul de execuție al nodului n_i.
- E este mulțimea arcelor în graf. Un arc de la nodul n_i la n_j, indică o dependență de date între n_i și n_j, astfel că n_j nu își începe execuția până când n_i nu și-a terminat-o.
- *condition* este o funcție definită pe mulțimea nodurilor în graf și cu valori în mulțimea numerelor naturale:

  $$condition: V \rightarrow \mathbf{N}, \text{ definită astfel:}$$

  - $condition(n_i) = 0$, dacă nodul n_i nu este nod condițional, și
  - $condition(n_i) = \mathbf{C}_i$, un număr natural diferit de zero indicând condiția calculată de n_i, dacă n_i este un nod condițional.

  Considerăm condițiile calculate în noduri diferite ca fiind distincte și identificate prin numere naturale, nenule și diferite. Asupra acestei observații revenim în paragraful destinat restricțiilor pentru modelarea noastră.
- *joinnode* este o funcție definită pe mulțimea nodurilor în graf și cu valori în $\{0, 1\}$:

  $$joinnode: V \rightarrow \{0, 1\}$$

  - $joinnode(n_i) = 0$, dacă n_i nu este nod join și
  - $joinnode(n_i) = 1$, dacă n_i este un nod join.
- *alternative* este o funcție definită pe mulțimea arcelor și cu valori în mulțimea $\{adevărat, fals, nedefinit\}$. *alternative* indică valoarea condiției (dacă există una), pentru care arcul respectiv este selectat.
  - $alternative((n_i,n_j)) = nedefinit$, dacă n_i nu este un nod condițional și
  - $alternative((n_i,n_j)) = adevărat$, dacă n_i este un nod condițional, iar arcul (n_i,n_j) corespunde alternativei *adevărat*, și

- *alternative* $((n_i,n_j)) = fals$, dacă n_i este un nod condiţional, iar arcul (n_i,n_j) apare în alternativa *fals*.

Definirea formală a GDC nu include referiri la modul de legare a nodurilor pe resurse. Acest fapt îl modelăm printr-o funcţie suplimentară, *resouces*.

resouces: $V \rightarrow$ {mulţimea resurselor din arhitectură}

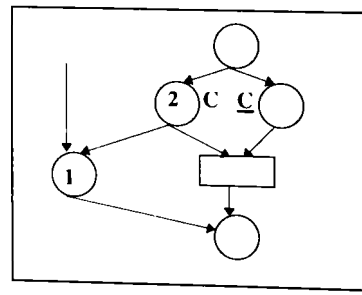
- ***resouces***(n_i) = resursa de care este legat n_i.

Restricţii de modelare ale GDC

Încheiem acest paragraf detaliind restricţiile impuse unui GDC, menite să păstreze caracterul determinist al planificării, să simplifice prezentarea algoritmului de planificare, dar altele consituie o limitare a modelării noastre.

1) Prima restricţie este legată de forma unei alternative. Astfel, fiecare alternativă are un singur nod de intrare şi unul singur de ieşire. Impunem această restricţie pentru a uşura descrierea algoritmului de planificare. Dacă alternativele specificării nu au această formă, ele pot fi transformate prin adăugarea de noduri fictive şi respectiv a arcelor de legătură între ele şi nodurile alternativelor.

2) Algoritmii de construire a GDC, şi mai departe cei de proiectare, nu evaluează condiţiile din nodurile condiţionale. De exemplu, un nod condiţional calculeaza condiţia $C_1 = ab$, iar alt nod condiţional, condiţia $C_2 = \underline{ab}$, unde a şi b au întotdeauna aceleaşi valori în cele două condiţii. În GDC cele două noduri condiţionale apar adnotate cu C_1 şi C_2, fără a face referire la felul cum se calculeaza C_1 şi C_2. Algoritmul de planificare consideră valide toate cele patru combinaţii posibile de valori ale lui C_1 şi C_2 (C_1C_2, $C_1\underline{C_2}$, $\underline{C_1}C_2$, $\underline{C_1}\underline{C_2}$), deşi situaţiile pentru C_1C_2 şi $\underline{C_1}\underline{C_2}$ nu pot apare în realitate. Putem spune că aceste situaţii sunt limitări ale puterii de modelare pentru GDC folosite, deşi unele cazuri pot fi eliminate prin rescrierea adecvată a specificării de intrare.



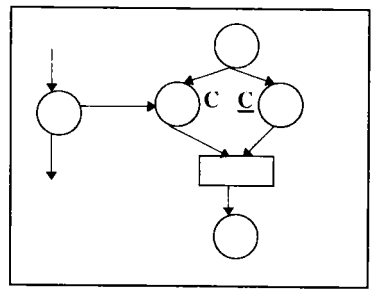
3) În GDC se consideră că fiecare nod condiţional calculează o condiţie distinctă. Această observaţie este legată de cea de la punctul 2. S-ar putea ca două condiţii să fie calculate folosind întotdeauna aceleaşi valori pentru variabile şi aceeaşi formulă de calcul, deci ele să fie identice. Totuşi, din punct de vedere al GDC ele reprezintă condiţii distincte, iar algoritmul de planificare analizează situaţiile pentru toate combinaţiile posibile de valori. Dacă în modelarea funcţionalităţii se impun luarea de

decizii funcție de aceeași condiție (cu aceeași valoare), acest fapt este reprezentat prin folosirea unei singure construcții decizionale și "comasarea" în aceleași alternative a tuturor acțiunilor dependente de condiție.

4) Dependențele de control indică executarea sau nu a unor procese funcție de valoarea unei condiții. Aceste procese sunt grupate în alternative. În figura precedentă, procesul 1 nu aparține vreunei alternative pentru **C**, dar activarea lui este controlată indirect de **C**, fiindcă el este un succesor al lui 2 (și care aparține alternativei pentru **C**). În cazul pentru **C**, execuția specificării este blocată, deoarece 1 nu este activat niciodată.

Pentru a evita astfel de cazuri, impunem GDC-ului restricția ca pentru orice arc (n_i, n_j) din graf și dacă n_j nu este nod join, atunci combinația de condiții pentru care n_j este executat va executa și pe n_i.

Aceste restricții nu impun însă ca un proces dintr-o alternativă să comunice doar cu procese ale alternativei respective. Ca și motivare indicăm următorul exemplu corect.



4.3 Definirea formală a planificării grafurilor legate, cu dependențe de date și de control

Planificarea unui graf stabilește momentele de timp pentru lansarea în execuție a nodurilor sale. Aceste noduri reprezintă raportat la specificarea de sistem procese și comunicările între procese legate la procesoare diferite. Nu toate nodurile unei alternative sunt legate aceleiași resurse cu cea ce execută nodul condițional. Prin urmare, după ce nodul condițional a calculat condiția este necesară transmiterea valorii ei și celorlalte procesoare din arhitectură, pentru a putea decide dacă se planifică sau nu nodurile din anumite alternative. Transmiterea condițiilor trebuie planificată asemenea celorlalte comunicări de date din graf.

Formal, planificarea unui graf legat, cu dependențe de date și de control înseamnă găsirea funcției:

$$Start: V \cup \{\text{transmisiile de condiții}\} \rightarrow \mathbf{N} \cup \{\infty\},$$

unde $Start(n_i) = t_i$ este momentul când n_i este planificat pentru execuție. Funcția *Start* trebuie să satisfacă următoarelor restricții:

- *Start*(nodul rădăcină) = 0. Întotdeauna, nodul rădăcină este planificat la 0.

- Prin convenție $Start(n_i) = \infty$, dacă $n_i \in V$ nu este activat pentru execuția curentă a grafului. Deasemenea dacă nu este necesară transmiterea condiției **C** altor procesoare, atunci $Start$(transmisia lui **C**) = ∞.
- Considerăm că $Start(n_i) \neq \infty$ și notăm cu MVC mulțimea valorilor condițiilor calculate la momentul $Start(n_i)$. Atunci mulțimea valorilor condițiilor ce controlează execuția lui n_i (denumită în capitolul 5 garda lui n_i) este o submulțime a lui MVC, deoarece garda unui proces trebuie cunoscută la momentul planificării sale.
- Dacă activarea lui n_i depinde de condiția **C**, calculată pe un alt procesor dé cât $resources(n_i)$ și $Start(n_i) \neq \infty$, atunci $Start$(transmisia lui **C**) + w(transmisia lui **C**) ≤ $Start(n_i)$. w(transmisia lui **C**) este timpul necesar transmiterii valorii lui **C**.
- $Start(n_i) \geq Start(n_k) + w(n_k)$, pentru orice n_k, predecesor al lui n_i. Această cerință este consecință a dependențelor de date din graf și spune că toți predecesorii nodului trebuie să fi fost executați, înainte ca acesta să poată să-și înceapă execuția. În cazul transmiterii unei condiții cerința impune ca înainte de transmiterea condiției, nodul condițional corespunzător să se fi terminat.
- Mulțimea $\{n_k \mid [Start(n_i), Start(n_i) + w(n_i)] \cap [Start(n_k), Start(n_k) + w(n_k)] \neq 0 \wedge resources(n_i) = resources(n_k) \neq$ procesorul hardware$\} = \phi$. Această restricție impune că dacă n_i este legat unei resurse diferită de procesorul hardware, atunci nici un nod n_k legat aceleiași resurse nu are intervalul său de execuție suprapus celui pentru n_i. Această restricție este validă și pentru transmisia condițiilor.
- Criteriul optimizat de planificare este ca $Start$(nodul final) să fie minim, deoarece planificarea minimizează timpul de execuție al grafului.

4.4 Transformarea specificării în vederea generării GDC. Generarea GDC-ului pentru o specificare de intrare

Acest paragraf schițează transformarea specificării de intrare într-o specificare echivalentă semantic, dar care să evidențieze dependențele de control. Proiectantul controlează granularitatea de lucru, indicând prin adnotări speciale instrucțiunile de control ale specificării inițiale, ce sunt folosite în procesul de proiectare. Apoi, specificarea este transformată astfel încât acțiunile dependente de condiții să fie grupate în procese distincte și care sunt tratate în mod specific în procesul de proiectare. Specificarea transformată este folosită pentru generarea GDC-ului.

Limbajul de specificare folosit de noi pentru a exemplifica construirea GDC-urilor este unul abstract, cu un număr minim de construcții, suficiente însă pentru a trata problemele relevante ale subiectului nostru. Alternativa ar fi fost să ne referim la un limbaj real pentru specificare de sistem (VHDL, SpecCharts, etc.), dar atunci ne-am fi "lovit" de două probleme suplimentare, dar secundare pentru noi: ar fi trebuit motivată alegerea limbajului de specificare, respectiv generarea GDC ar fi fost complicată inutil de complexitatea limbajului de specificare.

În continuare, detaliem construcțiile "limbajului" nostru pentru specificare de sistem. Specificarea unui sistem cuprinde un număr de procese concurente, unde procesul are semnificația din programarea concurentă [AS83]. Fiecare proces operează doar asupra datelor locale, neexistând date globale mai multor procese. Fiecare proces execută în

mod ciclic corpul său şi deci în momentul în care a executat ultima instrucţiune a corpului său, el continuă cu prima instrucţiune. Reluarea următoarei iteraţii în execuţia unui proces nu depinde de starea de execuţie a celorlalte procese din specificare, astfel că nu sunt necesare sincronizări suplimentare. Corpul unui proces este compus din instrucţiuni de atribuire, de ramificare (*if*), cicluri (*loop*), respectiv instrucţiuni de comunicare cu alte procese (*send*, *receive*). Deoarece nu există memorie comună, pentru transmiterea datelor altui proces se foloseşte mecanismul de transmitere bazat pe mesaje [AS83]. Procesul care transmite datele efectuează un apel la rutina *send*, iar procesul care le recepţionează execută *receive*. Pentru a evita "capcana sincronizărilor", adică o specificare puternic sincronizată, mecanismul send-receive folosit este cel fără blocare. Astfel un proces care execută *send* nu îşi blochează execuţia până când procesul corespunzător execută *receive*, dar dacă un proces execută *receive* în mod necesar execuţia lui se suspendă până când procesul transmiţător a executat *send*. Mecanismul comunicării prin mesaje fără blocare generalizează pe cel cu blocare, în sensul că putem implementa pe cel cu blocare având disponibil pe cel fără blocare.

Specificăm construcţiile pseudo-limbajului pentru specificare de sistem folosind notaţia BNF, iar pentru fiecare construcţie descriem semantica sa.

```
Specificarea sistemului = proces1 || proces2 || .... || procesx
```

Operatorul || inidică faptul că $proces_1$, $proces_2$, etc. reprezintă procese concurente.

```
proces = definiţii_de_date_şi_tipuri secvenţă_de_instrucţiuni
```

Partea `definiţii_de_date_şi_tipuri` introduce tipurile şi variabilele folosite în `proces`. După cum am menţionat deja, aceastea au semnificaţie locală, doar pentru proces. `secvenţa_de_instrucţiuni` constituie corpul procesului, şi este executat în mod ciclic.

```
secvenţă_de_instrucţiuni = secvenţă_de_instrucţiuni | instrucţiune

instrucţiune = atribuire | send(x,expresie) | receive(x)
             | if condiţie then secvenţa_de_instrucţiuni1
               else secvenţă_de_instrucţiuni2
             | while condiţie do secvenţă_de_instrucţiuni
             | instrucţiune_vidă
```

Această definiţie precizează felul instrucţiunilor disponibile: `atribuire` atribuie unei variabile valoarea unei expresii, `send(x,expresie)` transmite valoarea lui `expresie` lui `x`, folosit în `receive` din alt proces, iar `receive(x)` recepţionează pentru `x` valoarea transmisă dintr-un alt proces prin `send`. **if** `condiţie` **then** `secvenţă_de_instrucţiuni1` **else** `secvenţă_de_instrucţiuni2` execută pentru `condiţie` *adevărat* `secvenţă_de_instrucţiuni1`, iar pentru `condiţie` *fals* `secvenţă_de_instrucţiuni2`. Regula pentru **while** arată că `secvenţă_de_instrucţiuni` este executată atâta timp cât `condiţie` este adevărată. `instrucţiune_vidă` defineşte instrucţiunea vidă.

```
condiţie = expresie
```

Nu detaliem ce înseamnă `definiții_de_date_și_tipuri`, `atribuire`, `expresie` deoarece aceastea sunt nerelevante pentru construirea GDC. În schimb, înainte de a prezenta regulile pentru construirea GDC, discutăm implicația granularității de lucru a GDC asupra calității proiectării.

Granularitatea fină (de exemplu, fiecare nod al grafului reprezintă o instrucțiune) permite o modelare exactă a sistemului de proiectat, obținerea unei soluții de calitate bună, dar pentru că spațiul soluțiilor este foarte mare, timpul de proiectare crește corespunzător. *Granularitatea mare* lucrează la nivel ridicat, în sensul ca un nod al grafului corespunde unui proces din specificare. Complexitatea spațiului soluțiilor este mică, prin urmare și timpul de proiectare este mult mai mic, dar calitatea soluțiilor este redusă din cauza unor aproximări mai mari folosite în modelarea sistemului. Putem afirma, că adoptarea unei anumite granularități pentru reprezentarea grafului este un compromis între calitate și timp de proiectare.

Pentru a focaliza discuția, precizăm ce înțelegem din punct de vedere al GDC-ului prin asertiunea: aproximări în modelarea sistemului. Literatura recentă de specialitate [YW96] consideră că timpul de execuție al unui activități (proces sau comunicare) trebuie mai degrabă modelat printr-un interval de timp de cât printr-o valoare unică. Cauzele care fac ca timpul de execuție să nu poată fi precizate cu precizie sunt: existența dependențelor de control, arhitectura folosită pentru execuție (memorie cache, pipelining, etc.) și respectiv alte inexactități de modelare. Modelarea noastră atașează fiecărui proces un timp de execuție unic și pentru ca acesta să fie cât mai apropiat realității, trebuie să reducem la minim influența dependențelor de control asupra timpul de execuție.

Abordarea noastră consideră că proiectantul fixează granularitatea de lucru implicit prin specificare. El decide care sunt procesele ce compun specificarea sistemului, iar acestea sunt punctul de plecare pentru stabilirea nodurilor în graf. Procesele cu cicluri sunt desfăcute în trei componente: secvența ce precede ciclul, ciclul și secvența următoare ciclului. Fiecare componentă generează un nod în graf, cu mențiunea că corpul ciclului este desfășurat corespunzător numărului de iterații (dacă ciclul are N iterații se introduc N noduri în graf). Influența ramificațiilor asupra codului executat este captată prin desfacerea proceselor în locul unde apare o construcție *if*. Cele două alternative generează două noduri distincte, ca și secvența ce urmează lui *if*. Pentru a evita fărâmițarea specificării în procese cu granularitate foarte mică (mai ales pentru specificări dominate de control), proiectantul are posibilitatea să indice ciclurile sau ramificațiile care nu sunt folosite în generarea proceselor noi. Metoda de grupare a activităților dependente de control în procese noi are două consecințe benefice: surprinde dependențele de control și timpul de execuție al proceselor devine mai puțin dependent de relațiile de control.

Al doilea aspect care determină construirea GDC pentru o specificare este legat de tratarea comunicările existente între procese: în momentul în care se execută un *send*, procesul care este suspendat pentru *receive*-ul corespunzător, devine pregătit pentru execuție. În punctele *send* și *receive* se modifică setul de procese pregătite pentru execuție, ceea ce trebuie surprins și folosit pentru planificarea proceselor. Menționăm că procesele sunt planificate în mod nepreemptiv, astfel când un proces își începe execuția ea continuă până când el se încheie. În timpul execuției el poate transmite date

către un proces mult mai constrâns de cât el, dar care deşi astfel devine pregătit pentru execuţie, nu poate fi lansat datorită caracterului nepreemptiv al planificării. Această limitare este consecinţa granularităţii de lucru şi poate fi eliminată scăzând-o pe aceasta. De aceea, procesele trebuie desfăcute în punctele unde se execută instrucţiuni *send* sau *receive*.

În continuare detaliem regulile pentru transformarea specificării iniţiale într-o formă echivalentă semantic, dar pentru care fiecare proces reprezintă un nod în GDC. Specificarea astfel obţinută este folosită pentru obţinerea directă a GDC, iar apoi după paşii de partiţionare şi planificare, pentru generarea componetei hardware, respectiv a codului pentru partea software.

- Introducem un proces în specificare, conţinând o singură instrucţiune, `send (start,1)`. Acest proces simulează începutul execuţiei specificării şi păstrează consistenţa cu GDC-ul ataşat. Fiecarui proces din specificare, care nu începe cu *receive*, îi este adăugat ca primă instrucţiune `receive(start)`, iar în continuare instrucţiunile corpului său. Procesele care încep cu *receive* aşteaptă oricum date de la procesele predecesoare, astfel că `receive(start)` nu este necesar.
- Adăugăm specificării un proces conţinând doar instrucţiunea `receive(finish)`. Rolul procesului este de a marca sfârşitul execuţiei specificării şi el păstrază concordanţa cu GDC-ul ataşat. Fiecărui proces din specificare, ce nu se încheie cu *send*, îi adăugăm ca şi ultima instrucţiune pe `send(finish,1)`. Procesele cu *send* ca şi ultimă instrucţiune în corp efectueză oricum o transmitere de date către succesori, astfel că încheierea lor nu trebuie semnalată printr-un `send(finish,1)`.
- Pentru o instrucţiune `send(x,expresie)`, procesul este spart în două procese: primul proces conţine secvenţa premergătoare lui *send* şi instrucţiunea *send*, iar al doilea proces începe cu un `receive(x)` (adăugat la codul procesului), urmat de secvenţa următoare lui *send*.
- Dacă în corpul unui proces apare `receive(x)`, atunci procesul este desfăcut în două subprocese: un subproces care conţine secvenţa precedentă lui *receive* şi se încheie cu `send(next,1)` (adăugat la cod), iar al doilea subproces începe cu `receive(next,x)` şi conţine instrucţiunile următoare lui `receive(x)`. Semantica lui `receive(next,x)` arată că al doilea subproces nu poate începe până nu s-au executat *send*-urile pentru `x` şi `next`.
- Transformarea efectuată pentru un proces care conţine o instrucţie **`if`** `condiţie` **`then`** `secvenţă_de_instrucţini`$_1$ **`else`** `secvenţă_de_instrucţiuni`$_2$ este următoarea: secvenţa predecesoare lui *if* constituie un proces distinct. Acesta se încheie cu instrucţiunea **`if`** `condiţie` **`then`** `send(alternativa`$_1$`,1)` **`else`** `send (alternativa`$_2$`,1)`. Instrucţiunile din cele două alternativa ale *if*-ului generează două procese noi. Primul proces corespunde uneia din alternative, începe cu `receive(alternativa`$_1$`)` şi se încheie cu `send(join,1)`. Asemănător, al doilea proces conţine instricţiunile celeilalte alternative, începe cu `receive (alternativa`$_2$`)` şi se termină cu instrucţiunea `send(join,1)`. Al patrulea proces, este unul "dummy", el conţine doar secvenţa `receive(join); send(next,1)`. În fine, al cincilea proces începe cu `receive(next)` şi conţine secvenţa de cod următoare lui *if*.
- Pentru a putea transforma *while* trebuie să existe o adnotare în specificare, care să precizeze numărul maxim de iteraţii al ciclului. Dacă un proces conţine o instrucţiune *while*, având indicat un număr N de iteraţii, atunci se construiesc

următoarele procese în specificarea transformată. Un proces care conține secvența predecesoare lui *while* și se încheie cu `send(start_iterații,1)`. Fiecărei iterații îi corespunde un proces format din: `receive(start_iterații);` corpul de instrucțiuni ale lui *while*; `send(următoarea_iterație,1);`. Deasemenea, se generează un proces pentru secvența ce urmează lui *while* și care începe cu `receive(semnal_de_la ultima_iterație)`.

Algoritmul pentru transformarea specificării inițiale. Algoritmul pentru construirea GDC

transformă_specificare implementeză regulile enunțate anterior și este executat pentru fiecare proces din specificarea inițială. Rutina are două argumente: *procesul_curent* care memorează instrucțiunile ce formează proximul proces generat și *cod_neprocesat* pentru secvența de instrucțiuni care nu au fost încă parcurse.

```
void transformă_specificare (procesul_curent, cod_neprocesat){
   if (cod_neprocesat == ϕ)
      return;
   else
      case (instr = prima_instrucțiune(cod_neprocesat)) este {
      send(x,expresie):
         if (instr este ultima instrucțiune din cod_neprocesat) {
            generează cod pentru procesul_curent + instr;
         }
         else {
            generează cod pentru procesul_curent + instr;
            transformă_specificare (receive(x), cod_neprocesat);
         }
         break;
      receive(x):
         if (procesul_curent == NULL)
            transformă_specificare (instr, cod_neprocesat);
         else {
            generează cod pentru procesul_curent + send(urm,1);
            transformă_specificare (receive(urm), cod_neprocesat);
         }
         break;
      if expresie then secvența_de_instrucțiuni1 else
      secvență_de_instrucțiuni2:
         generează cod pentru procesul_curent + if expresie then
         send(alternativa1,1) else send(alternativa2,1);
         transformă_specificare (receive(alternativa1),
         secvență_de_instrucțiuni1 + send(join,1));
         transformă_specificare (receive(alternativa2),
         secvență_de_instrucțiuni2 + send(join,1));
         generează cod pentru receive(join) + send (next, 1);
         transformă_specificare (receive(next), cod_neprocesat);
         break;
      while expresie do secvență_de_instrucțiuni:
         generează cod pentru procesul_curent + send(start_iterații,
         1);
         generează cod pentru N procese receive(start_itarație) +
         secvență_de_instrucțiuni + send(următoarea_itarație,1);
         // N este numărul maxim de iterații al ciclului
```

```
            transformă_specificare (receive(semnal_de_la_ultima
               iterație), cod_neprocesat);
            break;
         other:
            transformă_specificare (procesul_curent + instr,
            cod_neprocesat);
      }
}
```

După transformarea specificării, apelând *construieşte_GDC* se obţine GDC-ul specificării. Fiecărui proces din specificarea transformată îi corespunde un nod în GDC. Pentru procesele care calculează condiţii se fixează funcţia *condition*, astfel încât ea să indice condiţiile calculate, iar pentru nodurile join se setează funcţia *joinnode* pe valoarea 1. Deasemenea, se generează noduri pentru comunicările între procese legate la resurse diferite. Arcele din graf reflectă dependenţele între procese.

```
void construieşte_GDC (specificarea transformată) {
      crează un nod în graf pentru fiecare proces din specificarea
      transformată;
      folosind specificarea transformată construieşte funcţiile
      condition şi joinnode;
      for (toate comunicările din specificarea transformată){
            x = procesul care execută send;
            y = procesul care execută receive;
            if (resources(x) == resources(y))
                  introdu în graf un arc de la x la y;
            else {
                  z = nodul pentru comunicarea de la x la y;
                  introdu în graf un arc de la x la z;
                  introdu în graf un arc de la z la y;
            }
      }
      folosind specificarea transformată construieşte funcţia
      alternative;
      evaluează timpul de execuţie pentru fiecare nod şi stabileşte w;
}
```

4.5 Concluzii

În acest capitol am definit formal graful dependenţelor de date şi de control, precum şi problema planificării GDC-urilor legate. Totodată, am indicat un set de reguli pentru transformarea unei specificări într-una echivalentă semantic, dar care să uşureze generarea GDC-ului ataşat. Deasemenea, am detaliat algoritmul care construieşte GDC-ul folosind specificarea transformată.

Folosirea GDC-urilor pentru co-sinteza hardware/software are rolul de a îmbunătăţii calitatatea implementărilor finale. Faţă un graf cu dependenţe de date, un GDC permite modelarea mai exactă a unei specificări, luarea unor decizii de proiectare mai eficiente şi în final obţinerea unei implementări mai ieftine. Dependenţele de control izolează alternativele a căror execuţii depind de condiţii şi stabilesc corespondenţa între o alternativă şi perechea condiţie-valoare ce o activează. GDC-ul este formal un cvint-uplu format din mulţimea nodurilor, mulţimea arcelor, mulţimea nodurilor condiţionale,

mulțimea nodurilor join și funcția de asociere a condițiilor la arce. Funcția de asociere identifică valoarea condiției pentru care este selectat arcul spre alternativă. Structura GDC-urilor trebuie să satisfacă patru tipuri de restricții. Acestea asigură existența unei planificări deterministe, simplifică proiectarea și prezentarea algoritmului de planificare, dar reprezintă și limitări de modelare ale metodei noastre. Aceste restricții nu restrâng gradul de aplicabilitate al metodei noastre, ci implică anumite cerințe pentru structurarea specificărilor de intrare.

Am definit formal problema planificării GDC-urilor legate ca fiind calculul funcției ce indică pentru fiecare nod al grafului timpul când începe execuția sa. Această funcție trebuie să satisfacă un set de constrângeri, care depind în ultimă instanță de informația pe care un nod trebuie să o cunoască în momentul execuției sale și de necesitatea de a secvențializa execuțiile nodurilor legate aceleiași resurse. Abordarea formală a planificării este utilă pentru că evidențiază criteriile de principiu, pe care trebuie să le îndeplinească orice soluție de planificare. Orice soluție care nu satisface aceste criterii este automat eronată. Acest rezultat este punctul de plecare pentru proiectarea euristicilor noastre de planificare a GDC-urilor din capitolul 5.

Capitolul prezintă în încheiere un set de reguli pentru transformarea unei specificări într-una echivalentă semantic, dar destinată generării GDC-ului atașat. Specificarea este compusă dintr-un set de procese, care comunică între ele prin mesaje. Setul de reguli scade granularitatea de lucru și urmărește să evidențieze dependențele de date și de control ale specificării. Proiectantul controlează fărâmițarea excesivă (și nerelevantă) a proceselor, adnotând acele instrucțiuni de control menite să genereze procese noi. Dependențele de date și de control sunt subliniate prin desfacerea proceselor pentru instrucțiunile *if*, *while*, *send* și *receive* aflate în componența sa. GDC-ul este construit direct folosind specificarea transformată. Nodurile sunt generate pentru fiecare proces al specificării și pentru comunicările între procesele legate la procesoare diferite, iar arcele ilustrează transferurile de date și de control. GDC-ul obținut este folosit pentru co-sinteza hardware/software.

Capitolul 5

Planificarea grafurilor legate, cu dependențe de date și de control

Rezumat

Capitolul 5 discută problema planificării grafurilor legate, având dependențe de date și de control. Algoritmii pentru planificarea GDC-urilor sunt bazați pe liste de priorități, deoarece aceștia sunt rapizi și găsesc în general soluții apropiate optimelor. Unele din deciziile de planificare sunt dependente de arhitectura ce execută implementarea finală. Arhitectura pe care noi am folosit-o este un heterogenă, formată din procesoare programabile de diferite tipuri, un procesor dedicat și diverse circuite de memorie. Componentele își transmit date folosind un grup de magistrale comune, la care fiecare are acces. Tabela de planificare este distribuită tuturor resurselor din arhitectură și este accesată pe baza momentului curent și a setului activ de condiții al resursei. Influența condițiilor din graf asupra planificării proceselor este grupată în două clase, în condiții care stabilesc execuția unui nod și condiții care influențează momentul planificării. Condițiile folosite în planificare pe alte resurse, de cât cele ce le calculează, trebuie comunicate. Capitolul discută două strategii privind fixarea momentului când este comunicată o condiție și două variante de stabilire a destinatarilor unei transmisii. Apartenența unui nod la mai multe trace-uri de execuție face ca el să fie caracterizat mai degrabă printr-o mulțime de priorități, de cât de una singură. Relațiile de control implică însă ca unele din noduri să fie planificate conform priorităților lor din alte trace-uri și un algoritm de planificare bun trebuie să micșoreze impactul acestor noduri asupra lungimii planificării finale. Primul algoritm euristic de planificare discutat este cel bazat pe liste de priorități. El atașează fiecărui nod o prioritate unică funcție de situația cea mai defavorabilă în care el poate fi implicat, dar fără ca prioritatea să țină cont de valorile curente ale condițiilor din graf. A doua euristică este cea prin ajustarea și potrivirea planificărilor. Ea planifică individual fiecare trace, după care le potrivește, lăsând trace-urile lungi nemodificate și adaptându-le pe cele scurte conform lor. Calitatea medie a soluțiilor sale este superioară celei pentru primul algoritm, mai ales din cauză că algoritmul păstrează o legătură cât se poate de actuală între priorități și trace-ul curent. Timpul său mediu de execuție este mai lung de cât cel al algoritmului bazat pe liste și el nu poate trata în mod natural grafurile ciclice.

5.1 Reiterarea problemei planificării grafurilor legate, cu dependențe de date și de control

Capitolul dedicat problematicii planificării grafurilor legate, cu dependențe de date și de control pornește de la rezultatele obținute pentru planificarea grafurilor legate, cu dependențe de date De aceea, prezentarea începe cu recapitularea concluziile noastre referitoare la planificarea grafurilor legate cu dependențe de date.

Mulțimea de activități (procese, comunicări între procese) ce trebuie planificată este modelată printr-un graf, a cărui noduri corespund activităților din sistem, iar arcele indică relațiile de precedență între noduri. Fiecare nod este *legat*, deoarece activitatea corespondentă lui este executată pe o resursă precizată din arhitectură. Criteriul optimizat prin planificare este minimizarea timpului total de execuție al grafului.

Un nod poate fi lansat la un moment dat în execuție, dacă la acel moment toți predecesorii săi au fost executați și resursa atașată lui este liberă. Cele două cerințe sunt *necesare*, dar nu și *suficiente*, pentru ca nodul să fie într-adevăr executat începând cu acel moment. Acest lucru se datorește faptului că la momentul implicat, mai multe noduri pot fi simultan pregătite pentru execuție pe resursa respectivă. Selecția *optimă* a următorului nod executat nu poate fi decisă în cazul general printr-o regulă calculabilă în timp polinomial, datorită caracterului NP-hard al problemei planificării optime. Consecința practică imediată a acestei proprietăți este că grafurile cu câteva zeci de noduri deja nu mai pot fi planificate optim într-un timp rezonabil de lung.

Soluția propusă în literatura de specialitate pentru evitarea caracterului NP-hard al planificării optime este definirea de euristici menite să găsească soluții cât mai apropiate celor optime. Suplimentar, o euristică trebuie să fie rapidă (ordinul ei să fie cel mult $O(n^3)$ [MI94b]), astfel că ajungem la concluzia că ea trebuie să fie un algoritm de tip *greedy*. Euristicile greedy de planificare sunt cele *bazate pe liste* [WG90][KA96]. Planificările bazate pe liste (*list scheduling*) atașează fiecărui nod o prioritate, pe care o folosesc în stabilirea următorului nod executat, atunci când mai multe noduri sunt pregătite de execuție pe aceeași resursă.

Cercetările în domeniu [KA96][WG90][KN84] s-au axat intens pe metoda bazată pe liste, iar "piatra de încercare" a reprezentat-o definirea unor priorități simple (ca ele să poată fi calculate repede), dar care să ducă spre soluții apropiate de optime. Prioritățile descrise în literatura de specialitate reflectă doar proprietățile nodurilor corespunzătoare (timpul lor de execuție, numărul succesorilor lor direcți, etc.), fără a reflecta maniera în care ele sunt legate pe resurse. Din acest motiv, potențialul lor de a găsi planificări bune pentru grafuri legate este mic. Prioritatea experimentată și prezentată de noi în capitolul 3 surprinde atât timpii de execuție și numerele de succesori direcți, cât și legările nodului și ale succesorilor săi. Planificările obținute sunt în general cu până la 10% mai lungi ca și optimele, dar într-un număr mare de cazuri ele sunt chiar optimele.

Pornind de la aceste rezultate și concluzii, capitolul 5 analizează folosirea euristicilor bazate pe liste la planificarea grafurilor legate, cu dependențe de date și de control. Studiul nostru discută modul în care planificarea nodurilor depinde de relațiile de

control şi identifică elementele de control, ce trebuie considerate pe parcursul algoritmului de planificare. Influenţa unei condiţii asupra planificării unei activităţi se poate manifesta în două moduri. Valoarea condiţiei poate hotărî executarea sau nu a activităţii sau poate modifica momentul când execuţia ei începe, însă fără a putea decide neexecutarea ei. Pentru fiecare nod al grafului trebuie cunoscute toate condiţiile ce hotărăsc dacă el este sau nu executat, respectiv toate condiţiile ce intervin la stabilirea momentului său de planificare. Orice algoritm de planificare a grafurilor legate, cu dependenţe de date şi de control, fără a fi neapărat din categoria celor bazaţi pe liste, trebuie să determine aceste informaţii preliminare.

Algoritmul de planificare trebuie să includă strategiile prin care condiţiile sunt transmise spre procesoarele ce le folosesc, astfel încât ele să fie disponibile la momentele când ele sunt necesare.

Paragraful 4.3 prezintă constrângerile pe care orice soluţie de planificare trebuie să le îndeplinească. În acest capitol prezentăm doi algoritmi euristici, care produc tabele de planificare satisfăcând implicit aceste constrângeri. Algoritmii prezentaţi reprezintă abordări complementare ale acestui gen de planificare. În prima situaţie, GDC-ul este tratat ca un tot unitar şi fiecare nod primeşte o prioritate unică potrivit situaţiei celei mai defavorabile în care el poate apare. Un nod are întotdeauna aceeaşi prioritate faţă de alte noduri şi ea nu depinde de valorile curente ale condiţiilor din graf. Condiţiile intervin la fixarea mulţimii nodurilor executate şi la calcularea momentelor lor de planificare. Al doilea algoritm desface GDC-ul în mulţimea tuturor secvenţelor posibile de execuţie, pe care apoi le prelucrează separat. O secvenţă de execuţie, denumită şi *trace*, reprezintă un subgraf legat, cu dependenţe de date al GDC-ului global. Nodurile unei secvenţe sunt planificate strict după priorităţile lor din acea secvenţă, fără a ţine cont de alte situaţii în care ele sunt deasemenea executate. Secvenţele planificate sunt apoi "potrivite", astfel încât secvenţele mai lungi să sufere cât mai puţine modificări şi implicit alungirile lor să fie cât mai mici.

Capitolul cinci are următoarea structură. Paragraful doi prezintă arhitectura folosită pentru executarea grafurilor cu dependenţe de date şi de control. Apoi, în paragraful trei identificăm maniera în care dependenţele de control determină planificarea activităţilor şi "izolăm" acele elementele de control, care condiţionează planificarea fiecărui nod din graf. În continuare, detaliem două euristici de planificare pentru grafurile legate, cu dependenţe de date şi de control. Paragraful şase prezintă o euristică bazată pe liste, iar paragraful şapte cea care reuneşte planificările obţinute pentru secvenţele (trace) distincte de execuţie. Pentru fiecare algoritm detaliem structura sa, modul său de comportare în situaţii relevante şi rezultatele experimentale vizând timpul lor de execuţie şi calitatea soluţiilor obţinute. Capitolul se încheie cu prezentarea concluziilor noastre, precum şi a direcţiilor de cercetare rămase deschise.

5.2 Arhitectura folosită pentru execuţia grafurilor legate, cu dependenţe de date şi de control. Structurile de date utilizate în planificare

Discuţia noastră despre planificarea grafurilor legate, cu dependenţe de date şi de control începe cu prezentarea arhitecturii implementării finale a sistemului. Acest lucru

este necesar, deoarece anumite decizii de planificare, cum ar fi de exemplu transmiterea condițiilor folosite în planificare, sunt dependente de această arhitectură.

Figura 5.1 detaliază arhitectura generică a implementării finale a sistemului. Câteva comentarii asupra arhitecturii sunt absolut necesare:

- Arhitectura cuprinde atât *procesoare programabile*, denumite uneori și procesoare "software", ce sunt destinate execuției componentei software a sistemului, precum și *procesoare dedicate* - ASIC-uri, FPGA, numite în continuare și procesoare "hardware". Acestea din urmă implementează funcționalitatea proceselor destinate implementării hardware. A treia categorie de componete ale arhitecturii sunt *circuitele de memorie*, folosite pentru păstrarea structurilor mari de date, accesibile mai multor procesoare din arhitectură. De aceste componente sunt legate procesele ce modelează activitățile de citire și scriere în/din structurile de date.

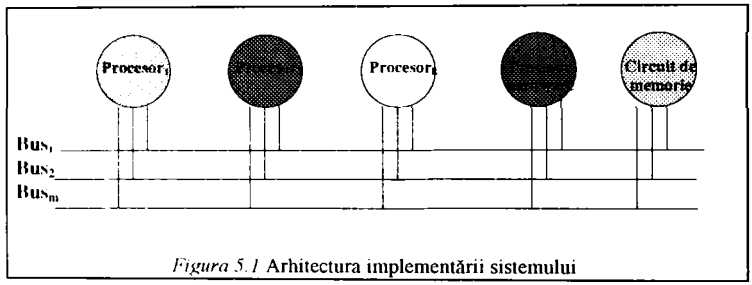


Figura 5.1 Arhitectura implementării sistemului

- Arhitectura este formată dintr-un număr arbitrar de procesoare programabile, de diverse tipuri. Tipul unui procesor programabil este exprimat prin costul și viteza sa de calcul. Arhitectura poate conține un singur procesor dedicat. Acest fapt ține de simplitatea arhitecturii generice și implicit a algoritmului de planificare și nu este o constrângere de modelare sau proiectare. Fizic însă, procesorul dedicat poate fi compus dintr-un set de circuite numerice. În fine, arhitectura cuprinde un număr arbitrar de circuite de memorie, de diferite feluri. Un circuit de memorie este caracterizat prin costul său și timpii săi de citire și de scriere. Această arhitectură extinde cele folosite de cercetări în domeniu, care presupun fie că arhitectura conține un singur procesor dedicat și unul singur programabil [EHB93][GM93], fie că procesoarele programabile sunt de același tip [KN84] sau că arhitectura este formată doar din procesoare programabile [YW96].
- Comunicarea între procesele legate la resurse diferite se face folosind magistralele arhitecturii. Fiecare comunicare este legată unei magistrale și fiecare procesor are acces la oricare dintre magistralele arhitecturii. Această cerință implică aparent ca topologia de interconectare a procesoarelor să fie neapărat una cu magistralele comune. După cum se va vedea în continuare, topologia particulară de interconectare a componentelor din arhitectură influențează doar strategia de trensmitere a condițiilor, pentru că restul comunicărilor sunt legate. Adoptarea altei topologii de interconectare, de exemplu punct la punct, schimbă doar metoda de transmitere a condițiilor.

În continuare detaliem structurile de date folosite în planificarea nodurilor între care sunt dependențe de date și de control:

- *Tabela de planificare* este distribuită procesoarelor componente. Fiecare procesor are un *planificator* (*scheduler*), care pe baza tabelei locale de planificare, a setului curent al condițiilor active, precum și a momentului curent de timp, lansează în execuție activitatea necesară. Momentul curent de timp și setul curent al condițiilor active sunt păstrate în regiștrii dedicați. Figura 5.2 indică structurile de date folosite de planificator.

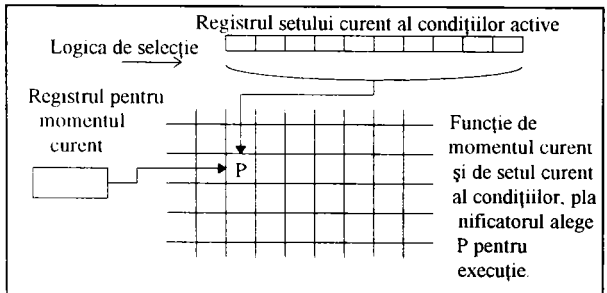


Figura 5.2 Structurile de date folosite de planificator

Tabela de planificare este organizată în felul următor:

- Liniile tabelei corespund la momente crescătoare de timp, astfel că selectarea unei linii se face pe baza momentului curent al execuției.
- Coloanele tabelei indică toate combinațiile valorilor condițiilor active, care pot apare pe parcursul execuției. Dacă un procesor calculează condiția **C**, atunci din momentul în care **C** devine cunoscută, ea aparține cu valoarea corespunzătoare setului curent al condițiilor active de pe procesor. Când condiția **C** este plasată pe magistrală de către procesorul ce o calculează, ea este preluată de toate procesoarele ce o folosesc în planificare și adăugată (cu valoarea transmisă) la setul curent al condițiilor active de pe acele procesoare. Decizia privind momentul când condiția este transmisă celorlalte procesoare este luată de către planificator.
- Elementul din tabelă selectat de momentul curent și setul curent al condițiilor active reprezintă activitatea (nodul) planificat în continuare.

Una din consecințele felului în care o specificare este modelată printr-un GDC este aceea că fiecare nod al grafului este executat cel mult o dată pentru o parcurgere a grafului. Această proprietate este denumită *determinism logic* [ED97]. Tabela de planificare trebuie astfel construită încât ea să respecte proprietatea de determinism logic. Dacă în tabelă nodul P este planificat în coloana cu eticheta logică E (setul curent al condițiilor active), pentru a respecta proprietatea de determism logic este necesar ca pentru execuția grafului ce selectează coloana E să nu se mai selecteze o coloană care să-l conțină pe P. Altfel formulat, dacă execuția curentă face ca expresia E să fie adevărată, nici una din expresiile ce etichetează coloane ce îl conțin pe P nu trebuie să fie adevărate. Situația în care cerința formulată nu este adevărată se numește *conflict*. Tabela de planificare trebuie astfel calculată încât între coloanele sale să nu apară conflicte.

În continuare definim o proprietate importantă referitoare la coloanele unei tabele de planificare ce se pot găsi în conflict.

Lemă: Fie E şi E' expresiile ce etichetează două coloane ale tabelei de planificare ce conţin planificarea nodului P. Între expresiile E şi E' există următoarele două proprietăţi

1) $E - E' \neq \emptyset$
2) $E' - E \neq \emptyset$

Atunci nu se poate ca P să fie executat pe parcursul aceleaşi execuţii a grafului atât conform coloanei cu E cât şi a celei cu E' sau altfel formulat, coloanele E şi E' nu pot fi în conflict.

Demonstraţie: Din proprietăţile 1 şi 2 ale lui E şi E' deducem că există condiţia $c \in E$ şi $c \notin E'$ şi condiţia $c' \in E'$ şi $c' \notin E$.

Presupunem că P ar fi executat prima dată în timp pentru coloana cu expresia E, deci la momentul respectiv sunt cunoscute toate condiţiile din E, inclusiv c. P este excutat a doua oară pentru coloana expresiei E', dar acest moment este ulterior primei execuţii, astfel că pe procesor sunt cunoscute atât condiţiile din E' dar şi c, ceea ce contrazice ipoteza făcută.

Consecinţa acestei leme este că în tabela de planificare nu pot apare conflicte între coloane a căror expresii logice sunt disjuncte. Conflictele pot apare doar între coloane etichetate cu E şi E' ce satisfac relaţiile $E \subset E'$ sau $E' \subset E$.

5.3 Modul în care dependentele de control influenţează planificarea

Mulţimea nodurilor executate (activate) este stabilită de valorile setului de condiţii din specificare. De exemplu, pentru o condiţie adevărată sunt executate nodurile alternativei sale *adevărat*, iar când ea este falsă cele ale alternativei *fals*, dar niciodată ambele.

Figura 5.3 exemplifică modul în care valorile condiţiilor stabilesc seturile de activităţi (procese, comunicări) executate. Modul de legare al proceselor pe procesoare este indicat prin haşuri, iar comunicările sunt plasate pe aceeaşi magistrală.

Pentru combinaţia de condiţii **CK** procesele şi comunicările executate sunt: 1, 2, (2,3), 3, 4, 9, 11, (11,15), 12, 13, (13,14), 14, 15, (15,16), 16, 10. Pentru **CK** procesele şi comunicările activate sunt: 1, 2, (2,3), 3, 4, 9, 11, (11,15), 12, (12,14), 14, 15, (15,16), 16, 10. **CGK** activează pe 1, 2, 5, (5,6), 6, (6,8), 8, (8,9), 9, 11, (11,15), 12, 13, (13,14), 14, 15, (15,16), 16, 10, iar **CGK** pe 1, 2, 5, (5,6), 6, (6,8), 8, (8,9), 9, 11, (11,15), 12, (12,14), 14, 15, (15,16), 16, 10. În fine, pentru **CGK** avem 1, 2, 5, (5,7), 7, (7,8), 8, (8,9), 9, 11, (11,15), 12, 13, (13,14), 14, 15, (15,16), 16, 10, şi pentru **GGK** 1, 2, 5, (5,7), 7, (7,8), 8, (8,9), 9, 11, (11,15),12, (12,14),14,15, (15,16), 16, 10.

Timpul când execuţia unui nod *join* începe depinde de alternativa selectată, iar această influenţă se "propagă" asupra tuturor succesorilor nodului *join*. *Execuţia* nodului *join* nu este condiţionată de valoarea condiţiei corespunzătoare, dar *momentul* când ea începe depinde de valoarea condiţiei.

Referindu-ne la exemplul din figura 5.3, nodurile 8 şi 9 sunt executate indiferent de valorile condiţiilor **G** şi **K**. Însă, ceea ce depinde de valorile celor două condiţii sunt momentele de timp când 8 şi 9 sunt planificate. Pentru **K** este activat 13 şi (13,14), iar pentru **K** comunicarea (12,14), astfel că

14 devine "ready" la momente de timp diferite pentru cele două cazuri. În mod asemănător, pentru **G** se execută (5,6), 6 şi (6,8), iar pentru **G** (5,7), 7 şi (7,8), astfel că 8 devine ready la timpi diferiţi.

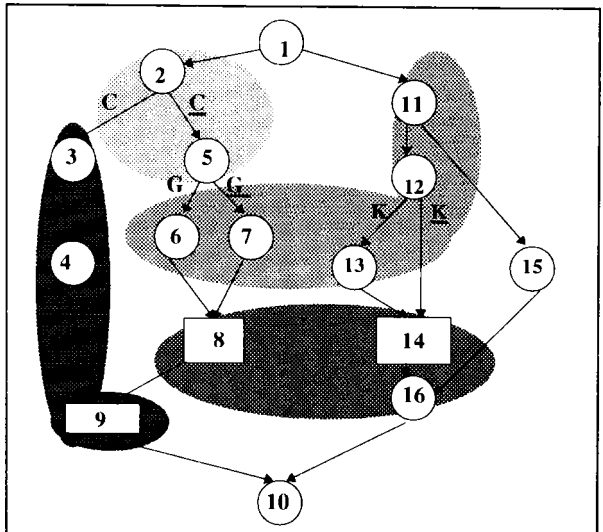


Figura 5.3 Execuţia nodurilor funcţie de valoarea condiţiilor

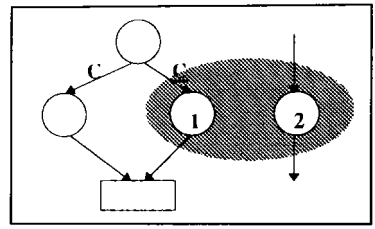
Resursele arhitecturii sunt încărcate diferit funcţie de alternativa selectată. Pentru că nodurile unei alternative sunt legate de elementele de procesare, execuţia sau nu a acelei alternativei modifică încărcarea resurselor folosite de ea. Ducând raţionamentul mai departe, execuţia celorlalte noduri legate aceloraşi resurse ca şi nodurile alternativei depinde de valoarea condiţiei. Această influenţă se transmite mai departe succesorilor nodurilor mapate pe aceleaşi resurse.

În figura 5.3, procesele 6, 7, 11, 12 şi 13 sunt mapate aceluiaşi procesor. Pentru **K**, procesul 13 nu este executat, iar resursa lui este mai puţin constrânsă. În acest fel, se poate influenţa planificarea lui 6 sau 7. Asemănător, funcţie de **G** este executat 6 sau 7 şi fiecare execuţie determină încărcări diferite ale procesorului. Astfel se pot modifica momentele de timp când încep execuţiile lui 11, 12 şi 13.

După cum se manifestă dependenţele de control în cele două exemple descrise anterior, ele sunt grupate în două categorii diferite:

- Când un nod (proces, comunicare) aparţine unei alternative condiţionale, valoarea acelei condiţii determină executarea sau nu a sa. Condiţiile cu aceste proprietăţi formează *lista de gărzi* a nodului. Pentru a preciza dacă un nod este sau nu executat, trebuie să cunoaştem valorile tuturor condiţiilor din garda sa. Lista gărzilor unui nod este o mărime statică, deoarece ea depinde doar de alternativele din care acesta face parte.
- În figura 5.3 nodul 14 începe la timpi diferiţi după cum **K** este sau nu adevărat. Concluzia noastră a fost că 14 este executat întotdeauna, independent de valoarea lui **K**, dar timpul la care el îşi începe execuţia depinde de această valoare. În figura următoare, nodul 2 este legat aceleiaşi resurse cu nodul 1, care este gardat de condiţia **C**. Funcţie de valoarea lui **C**, 1 poate sau nu să fie activat, ceea ce modifică

gradul de încărcare al resursei și influențează timpul de planificare a lui 2. Astfel, după cum 1 este sau nu activ, 2 este planificat mai devreme sau mai târziu. Prin urmare, o condiție poate să nu condiționeze activarea unui nod, dar să intervină în stabilirea momentului când începe execuția sa. Condițiile cu aceste proprietăți formează *lista de influență* a nodului. Lista de influență a unui nod are deasemenea un caracter static și este calculată funcție de structura grafului și a modului de legare a nodurilor pe resursele din arhitectură.



Conflictele din tabela de planificare sunt întotdeauna produse de condiții din lista de influență a unui nod. Un conflict apare în tabela de planificare atunci când același nod este plasat în două coloane distincte, a căror expresii logice nu se exclud reciproc. În mod evident cele două coloane aparțin la trace-uri diferite, deoarece un nod nu poate fi planificate de două ori (pentru fiecare din coloanele aflate în conflict) pentru același trace. Dacă notăm cu E_1 și cu E_2 expresiile logice ale celor două coloane, atunci conform paragrafului 5.2 avem $E_1 \subset E_2$ sau $E_1 \supset E_2$. Condițiile suplimentare din E_1 sau E_2 pot aparține doar listei de influență a nodului, din cauză că lista de gărzi trebuie întotdeauna cunoscută complet în momentul planificării nodului.

Secțiunea următoare prezintă regulile pentru calculul gărzilor și a listei de influență pentru un nod.

Definirea listei de gărzi, respectiv a listei de influență pentru un nod

Lista de gărzi este mulțimea valorilor tuturor condițiilor care determină activarea sau nu a unui nod. Ea este formată din condițiile corespunzătoare tuturor alternativelor din care nodul face parte. Gărzile unui nod sunt calculate după următoarele reguli:

- Rădăcina grafului are întotdeauna garda 1 sau altfel spus, ea nu este controlată de nici o condiție.
- Dacă nodul nu este un nod *join* și nu este descendent direct al unui nod condițional, atunci garda sa este reuniunea gărzilor tuturor predecesorilor săi.
- Dacă nodul nu este nod *join* și este un descendent direct al unui nod condițional (a cărui condiție o notăm cu **C**), atunci garda sa este egală cu garda nodului condițional reunită cu **C**, când procesul aparține alternativei *adevărată* și cu garda nodului condițional reunită cu $\underline{\mathbf{C}}$, dacă el aparține celei *false*.
- Garda unui nod *join* este egală cu cea a nodului condițional atașat.

Ca și exemplu prezentăm calculul gărzilor pentru procesele și comunicările din figura 5.3. Nodul 1, ca rădăcină, are garda 1. 2 și 11 au fiecare garda egală cu reuniunea gărzilor predecesorilor lor, deci

Garda(2) = Garda(11) = Garda(1) = **1**. Comunicarea (2,3) este descendent direct al nodului condiţional 2 pentru alternativa adevărată, deci garda lui (2,3) este garda lui 2 reunită cu **C**. Garda((2,3)) = {**C**}. Garda(3) = Garda(4) = Garda((2,3)) = {**C**} şi asemănător Garda(5) = Garda(2) U {**C**} = {**C**}, Garda((5,6)) = Garda(5) U {**G**} = {**C**, **G**}, Garda(6) = Garda((6,8)) = Garda((5,6)) = {**C**,**G**}, iar Garda((5,7)) = Garda(5) U {**G**} = {**C**,**G**}, Garda(7) = Garda((7,8)) = Garda((5,7)) = {**C**,**G**}. Garda lui 8 este egală cu cea a lui 5 şi este {**C**}. iar cea a lui 9 este **1**. adică tocmai garda lui 2. Garda(12) = Garda(11) = Garda((11,15)) = Garda(15) = Garda((15,16)) = Garda(14) = Garda(16) = **1**. Garda(13) = Garda((13,14)) = {**K**}, iar Garda((12,14)) = {**K**}.

Înainte de a defini regulile noastre pentru *listelor de influenţă*, revenim asupra rolului acestora şi anume acela de a indica setul condiţiilor care influenţează momentul planificării, dar nu condiţionează executarea unui nod. Doarece activarea sau nu a unui nod funcţie de o condiţie este modelat prin gărzi, impunem restricţia ca *o condiţie din lista de gărzi a unui nod să nu apară şi în lista sa de influenţă*. Suplimentar, o condiţie din lista de influenţă a unui nod trebuie să îndeplinească una din următoarele cerinţe:

- Nodul este succesor al unui nod având condiţia în lista sa de gărzi. Setul predecesorilor executaţi este dependent de valoarea condiţiei şi prin urmare planificarea predecesorilor şi implicit a nodului depinde de acea condiţie. De exemplu, activarea unui nod *join* nu depinde de valoarea condiţiei din nodul condiţional ataşat. În schimb, momentul de planificare al nodului *join* depinde de valoarea condiţiei, deoarece funcţie de ea sunt activate seturi distincte de procese.
- Nodul este legat aceleiaşi resurse cu un nod având condiţia în lista sa de gărzi sau de influenţă. Cum cele două procese partajează aceeaşi resursă şi pentru că planificarea celui de-al doilea depinde de valoarea condiţiei, ea poate influenţa şi planificarea primului. Un caz special legat de dependenţa de condiţii a încărcării resurselor este cel al influenţării planificării de însăşi comunicarea condiţiei. Comunicarea unei condiţii determină o încărcare suplimentară a unei magistrale şi poate cauza modificarea planificării altor comunicări mapate pe acea magistrală. Pentru a suprinde static această influenţă, ar trebui ca fiecare comunicare de condiţii să fie legate unei anumite magistrale, iar atunci am asimila comunicarea condiţiei cu un nod gardat cu acea condiţie. Dacă acceptăm că transmiterea condiţiilor nu este legată de resurse, atunci evităm acest element pentru a păstra natura statică a listelor de influenţă, dar îl vom considera separat pe parcursul algoritmului de planificare.
- Nodul este succesor al unui nod având condiţia în lista sa de influenţă. În acest caz este posibil ca predecesorul să fie planificat dependent de valoarea condiţiei, iar această dependenţă se transmite şi succesorilor săi, inclusiv procesului curent.

Când toate comunicările, inclusiv cele ale condiţiilor folosesc aceeaşi magistrală, procesele şi comunicările din figura 5.3 au următoarele liste de influenţă. Procesele 1, 2, 5 nu au listă de influenţă, iar 11, 12, 13 au pe {**C**,**G**} datorită faptului că sunt pe aceeaşi resursă cu 6 şi 7, care au pe **C** şi **G** în garda lor. Procesul 8 şi comunicarea (8,9) au pe **G** şi **K** în lista lor de influenţă, atât datorită legării lor, cât şi a predecesorilor lor. Datorită legării sale (2,3) depinde de {**G**,**K**}, (5,6), (5,7), (6,8) şi (7,8) de {**K**}, (13,14) şi (12,14) de {**C**,**G**}, iar (11,15) şi (15,16) de {**C**,**G**,**K**}. Datorită predecesorilor, 14, 15 şi 16 au lista de influenţă {**C**,**G**,**K**}. Pentru că 6 şi 7 împart aceeaşi resursă cu 13, lista lor de influenţă este compusă din **K**. În fine, 3 şi 4 depind de **G** şi **K**, iar 9 de {**C**, **G**, **K**}.

Listele de gărzi și de influența ale nodurilor din graf sunt calculate static, înainte de a începe algoritmul de planificare. Astfel stabilim pentru fiecare nod mulțimea condițiilor de care depinde planificarea și momentul planificării sale.

5.4 Cerintele generale ale strategiilor de transmitere a condițiilor

Folosirea dependențelor de control în planificare pornește de la următorul criteriu "de bun simț": *valorile condițiilor folosite în luarea deciziilor de planificare trebuie cunoscute când aceste decizii sunt luate.* Acest paragraf derivă consecințele acestui criteriu pentru transmiterea într-un sistem multiprocesor a valorilor condițiilor. Secțiunea următoarea introduce două reguli euristice pentru transmiterea condițiilor, care satisfac criteriului și pot fi înglobate în orice algoritm de planificare cu dependențe de date și de control.

Criteriul menționat anterior induce două cerințe pe care orice strategie de transmitere a condițiilor în sistem trebuie să le satisfacă:

- **Aspectul local**: o condiție trebuie cunoscută de toate procesoarele unde valoarea ei este folosită în luarea deciziilor de planificare. Prin urmare, dacă aceste procesoare sunt diferite de cel ce calculează condiția, atunci condiția trebuie transmisă lor.
- **Aspectul temporal**: o condiție trebuie cunoscută de un procesor la momentele când sunt luate decizii de planificare dependente de valoarea ei. Această cerință cuprinde o doză de "incertitudine", pentru că momentele de planificare sunt necunoscute și ele depind la rândul lor de momentul când are loc transmiterea condiției. De aceea, înlocuim cerința generală printr-o regulă euristică, dar perfect justificată de obiectivul dorit: transmiterea unei condiții trebuie pe cât posibil să nu întârzie planificarea activităților ce depind de ea. Momentul de planificare al unei activități este cel stabilit fără a lua în calcul timpul și resursele necesare transmiterii condiției. Acest fapt definește intervalul de timp în care o condiție trebuie transmisă altor resurse, ca fiind cuprins între momentul când condiția este calculată (adică momentul când procesul ce calculează condiția își încheie execuția), respectiv cel mai devreme moment când ea este folosită în luarea unei decizii de planificare pe un alt procesor de cât cel ce o calculează.

Prin urmare, un algoritm de planificare a grafurilor legate, cu dependențe de date și de control, executate pe sisteme multiprocesor trebuie să răspundă la următoarele două întrebări:

- Care sunt procesoarele care folosesc o condiție pentru planificare și deci, cine sunt destinatarii unei transmisii de condiție?
- Care este momentul de timp când se transmite o condiție celorlate resurse care o folosesc?

Răspunsul la prima întrebare este relativ simplu: o condiție trebuie cunoscută cel puțin de către resursele unde sunt luate decizii de planificare dependente de ea sau altfel spus, resursele care au legate noduri cu condiția în gardă sau în lista lor de influență.

Răspunsul la întrebarea doi este mai nuanțat, întru-cât el presupune formularea unei reguli pentru stabilirea celui mai târziu moment când transmiterea unei condiții trebuie făcută și pe baza ei, definit algoritmul pentru planificarea transmiterii condiției. Transmiterea unei condiții este o activitate de comunicare, ce ocupă magistralele asemenea oricărei transmisii de date și care ar trebui să aibă atașată o prioritate, pe baza căreia euristica de planificare să o execute asemeni oricărei activități din graf. După cum este motivat în următoarea secțiune, această abordare este greu de realizat.

Anterior am motivat că cel mai târziu moment când valoarea unei condiții trebuie cunoscută pe un procesor este primul moment când pe acel procesor se ia o decizie de planificare dependentă de acea condiție.

Când nodul este legat unei alte resurse de cât cea care calculează condiția, valoarea condiției intervine în planificarea nodului după cum urmează:
1) O condiția din lista de gărzi a nodului trebuie cunoscută, deci să fi fost deja transmisă, în momentul începerii execuției sale.
2) O condiție din lista de influență a nodului trebuie cunoscută în momentul planificării sale, dacă ea a fost deja folosită în planificarea unui predecesor al său (de exemplu, înaintea predecesorului este planificat un proces cu condiția în garda sa).

Cele două exemple de mai jos arată felul în care planificarea nodurilor depinde de condițiile din listele lor de influență precum și modul în care trebuie stabilite cele mai târzii momente când valorile condițiilor trebuie cunsocute. În figura 5.5 procesul 2 calculează condiția **C**, iar modul de legare a proceselor este evidențiat prin haşuri. Presupunem că toate comunicările folosesc aceeași magistrală şi că 2 şi 3 se termină la acelaşi moment. **C** apare în garda proceselor 4, 5, 6 şi 7 şi a comunicărilor (2, 4) şi (6, 8), precum şi în lista de influență a lui 9 şi 10 şi a comunicărilor (3, 9) şi (9, 10). Comunicările (2, 4) şi (3, 9) devin pregătite pentru execuție în acelaşi moment, iar secvența în care sunt executate depinde de prioritatea lor. Dacă (3,9) are prioritate mai mare, atunci ea va fi întotdeauna planificată la acelaşi moment indiferent de valoarea lui **C**, iar procesorul care o planifică (procesorul lui 3) nu trebuie (neapărat) să cunoască valoarea lui **C** în momentul planificării lui (3, 9). Explicația este că nici un predecesor a lui (3,9) nu a fost planificat funcție de **C** şi (3,9) nu are pe **C** în garda sa, la planificarea lui nu trebuie s-o cunoaştem pe **C**. Dacă însă prioritatea lui (3,9) este mai mică, atunci momentul la care este planificată depinde de valoarea lui **C**. Pentru **C**, (3,9) este planificată după (2, 4), iar pentru **C**, ea începe imediat după terminarea lui 3, pentru că (2,4) nu este executat. Prin urmare, procesorul trebuie să cunoască valoarea lui **C** pentru planificarea lui (3, 9). Momentul când condiția trebuie să fi fost deja transmisă este cel mai devreme moment între planificarea lui (2,4) pentru **C** şi planificarea lui (3,9) pentru **C**. Caracteristica acestui caz este că pe magistrala lui (3,9) există o altă comunicare, (2,4), de prioritate mai mare şi având **C** în garda sa.

În al doilea caz păstrăm toate ipotezele, mai puțin legarea comunicărilor. Suplimentar, considerăm că 9 este planificat după 5. Dacă (2,4) şi (6,8) sunt legate altei magistrale de cât (3,9) şi (9,10) atunci planificarea lui (3,9) nu depinde de **C** indiferent de prioritatea ei. Prin urmare, la planificarea lui (3,9) resursa lui 3 nu trebuie să cunoască pe **C**. 9 este planificat funcție de valoarea lui **C**, astfel că planificarea succesorilor săi (9, 10) şi 10 depinde de valoarea lui **C**. Prin urmare, procesorul lui 10 trebuie să cunoască valoarea lui **C** pentru a stabili momentul lansării în execuție a lui 10 şi de aceea condiția trebuie fi fost deja transmisă la cel mai devreme moment al planificării lui 10 pentru **C** şi respectiv pentru **C**.

Concluzia ce rezultă din cele două exemple este că valoare unei condiții trebuie să fi fost deja transmisă altor procesoare, cel târziu, la cel mai devreme moment când se planifică un nod ce îndeplinește una din cerințele 1 sau 2. Cel mai devreme moment este stabilit analizând planificările trace-urilor, ce corespund la toate combinațiile de

valori posibile pentru condiţiile din graf şi reţinând timpul primei activităţi planificate funcţie de acea condiţie.

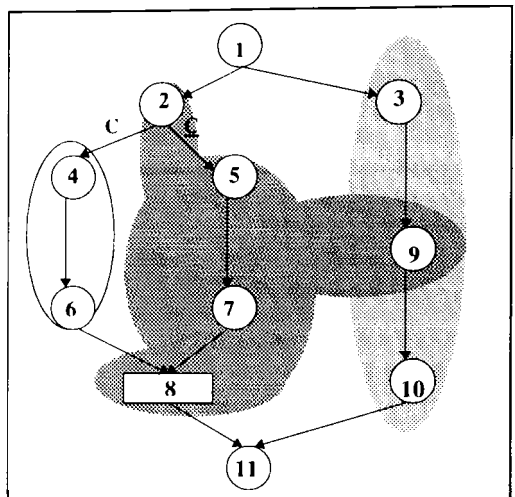


Fig.ura 5.5 Planificarea proceselor funcţie de condiţiile din lista lor de influenţă

Deoarece o condiţie este cunoscută la momentul cel mai devreme când ea este folosită în planificare, nu există "pericolul" ca în tabela de planificare să apară o situaţie de conflict. Demonstrăm această afirmaţie. Presupunem că există două coloane etichetate cu expresiile E_1 şi E_2, coloane ce sunt în conflict pentru nodul P. P este planificat la momentul t_1 pentru coloana E_1 şi la t_2 pentru coloana E_2. Fără a restrânge generalitatea demonstraţiei presupunem că t_1 este minimul între t_1 şi t_2 şi că $E_1 \subset E_2$. Condiţiile din E_1 şi E_2 aparţin fie listei gărzi sau a listei de influenţă a lui P. După cum am motivat în paragraful 5.3, conflictul poate apare doar din cauza listelor de influenţă. Fie c o condiţie din $E_2 - E_1$, atunci ea este necunoscută la momentul t_1, ceea ce contravine regulii după care trebuie transmise condiţiile din listele de influenţă. Prin urmare nu există nici o condiţie $c \in E_2 - E_1$, astfel că $E_2 - E_1 = \varnothing$ şi deci $E_1 = E_2$. Concluzia este că coloanele etichetate cu E_1 şi E_2 nu sunt distincte şi deci ele nu pot fi în conflict.

Strategii de comunicare ale condiţiilor

Anterior am prezentat criteriile referitoare la folosirea condiţiilor în planificare. Aceste criterii se referă la setul de procesoare ce folosesc o anumită condiţie, respectiv la momentul când o condiţie trebuie cunoscută acestora. În continuare particularizăm aceste criterii generale în reguli, pe care le folosim apoi în construirea algoritmilor de planificare cu dependenţe de date şi de control.

Prima variantă este aceea de a transmite *neselectiv* condiţiile, pe măsură ce ele devin disponibile, tuturor procesoarelor din arhitectură. Transmiterea condiţiilor este simplă,

pentru că nu trebuie calculați destinatarii transmisiei. Dezavantajul constă în faptul că o condiție este folosită în planificarea nodurilor unui procesor, chiar dacă ele nu depind de ea (de exemplu, condiția nu apare în niciuna din listele de gărzi sau de influență ale nodurilor). Prin urmare, numărul coloanelor în tabela de planificare este mai mare de cât dacă transmiterea condițiile s-ar face selectiv.

Varianta *selectivă* comunică o condiție doar acelor procesoare unde ea este folosită în luarea deciziilor de planificare. Pentru a păstra lucrurile simple, considerăm că un procesor trebuie să cunoască o condiție, dacă el are legat un nod cu condiția în garda sa sau în lista sa de influență. Astfel, destinatarii unei transmisii de condiție se stabilesc static, după ce s-au calculat gărzile și listele de influență ale nodurilor și înainte de a începe algoritmul de planificare.

Planificarea transmiterii unei condiții ar trebui modelată printr-un nod în graf, având timp de execuție și o prioritate folosită pentru planificare. Comunicarea condiției ar fi luată în calcul de algoritmul de planificare și astfel s-ar evita situații defavorabile care altfel pot apare, cum ar fi întârzierea de către această comunicare a alteia cu prioritate mai mare. Dificultatea acestei metode constă în faptul că este greu de stabilit în mod static (înainte de planificare) dacă și unde trebuie transmisă condiția. Acest lucru se exprimă din punct de vedere formal prin faptul că nu putem stabili static cine sunt succesorii imediați ai comunicărilor de condiții. Transmiterea sau nu a condițiilor din listele de influență depinde esențial de felul în care s-a făcut planificarea și în mod evident aceste informații nu sunt disponibile înainte de planificare. Prin urmare, comunicarea condițiilor trebuie făcută la momente stabilite prin reguli euristice, care să influențeze cât mai puțin restul deciziilor de planificare și să nu întârzie pe cât posibil planificarea altor noduri.

O primă variantă este de a transmite o condiție imediat după ce ea a fost calculată, pe prima magistrală găsită liberă. Această variantă o denumim *ASAP*. Regula se traduce prin a atribui întotdeauna, în mod formal, prioritate maximă comunicărilor de condiții. Această strategie este simplu de implementat, dar are dezavantajul că nu ține cont de particularitățile grafului și de maniera de legare a nodurilor pe resurse. De exemplu, prin această strategie se pot întârzia comunicări către noduri cu prioritate mare, deși condiția este folosită doar pentru planificarea unor procese cu prioritate mică. Deasemenea, la acest nivel este imposibil de stabilit dacă condiția trebuie *într-adevăr* transmisă, motiv pentru care uneori ea este comunicată, deși ea nu este folosită în planificare. Transmiterea condițiilor prin această regulă este naturală atunci când ea se face neselectiv, către toate procesoarele din arhitectură, dar ridică probleme dacă implementăm maniera selectivă de comunicare a condițiilor către procesoarele arhitecturii.

Varianta a doua, pe care o numim *ALAP*, este aceea de a întârzia transmiterea unei condiții până la limita superioară admisă, adică până la prima decizie de planificare dependentă de acea condiție și care este luată pe un alt procesor de cât cel care o calculează. Astfel se întârzie momentul de start al acelui proces, deoarece condiția se transmite la momentul când ea ar trebui deja folosită la planificarea lui. În anumite cazuri timpul total de execuție al grafului se poate lungi. Această metodă are însă următoarele avantaje: este simplu de implementat, se poate stabili simplu dacă comunicarea condiției este necesară și care sunt procesoarele destinatare ale condiției și

poate cel mai important, se pot transmite simultan valorile mai multor condiții. Dacă la momentul stabilit pentru comunicarea condiției, procesorul curent are disponibile și necomunicate alte k condiții, atunci toate pot fi transmise într-un același pachet de date. În acest fel, "economisim" k - 1 transmisii ulterioare de condiții.

5.5 Definirea priorităților într-un graf cu dependențe de date și de control

Când nodurile sunt planificate folosind doar dependențele de date, mulțimea nodurilor executate este întotdeauna aceeași. Nodurile au prioritate unică, ordinea lor de execuție rămâne nemodificată și timpul total de execuție al grafului este întotdeauna același.

În cazul grafurilor cu dependențe de date și de control, setul nodurilor executate depinde de valorile condițiilor din specificare. Fiecare combinație de valori pentru condițiile grafului activează o mulțime specifică de noduri (numită *trace*), executate într-o ordine proprie și cu un timp total de execuție specific. Un nod executat de două sau mai multe combinații de condiții poate avea importanțe diferite "vis-a-vis" de criteriul de planificare dorit, astfel că el este planificat la timpi diferiți pentru trace-uri diferite. Acest lucru face ca nodurile unui graf cu dependențe de date și de control să aibă mai degrabă o *mulțime de priorități* de cât o prioritate unică. Această mulțime cuprinde prioritățile ce corespund la toate combinațiile de condiții pentru care nodul este executat.

Prioritatea stabilește o ordine "bună" de execuție a nodului în cadrul trace-ului static din care el face parte. Una din problemele planificării grafurilor cu dependențe de date și de control este că mulțimea nodurilor executate nu poate fi descrisă complet pentru fiecare nod în momentul planificării sale. De exemplu, când este planificat predecesorul unui nod condițional nu putem preciza mulțimea completă a nodurilor ce vor fi apoi executate, deoarece aceasta depinde de condiții ce vor fi calculate ulterior. În astfel de situații nodul este planificat conform cazului cel mai defavorabil în care el este implicat.

Un alt caz defavorabil este cel în care un nod are prioritate ridicată pentru o combinație de condiții și o prioritate scăzută pentru o altă combinație. Dacă el este planificat potrivit priorității mari, atunci el va întârzia noduri prioritare ale combinației de condiții în care el are prioritate mică și reciproc. Acest dezavantaj devine și mai accentuat când planificările celor două combinații de condiții sunt aproximativ la fel de lungi și mai lungi de cât cele pentru restul combinațiilor de condiții. Orice decizie de planificare a nodului conform uneia din priorități duce la lungirea planificării totale în cealaltă situație. De aceea, când se analizează planificarea unui nod la un moment dat, se consideră impactul pe care aceasta îl are asupra timpilor de execuție ai tuturor trace-urilor din care el face parte. Decizia privind nodul ce va fi planificat următorul este un compromis, favorizând (prin timpul total de execuție ce rezultă) anumite trace-uri și dezavantajându-le pe altele. Dintre toate deciziile de planificare posibile la un moment dat, trebuie aleasă aceea care minimizează cel mai lung trace.

Algoritmii noștrii de planificare acoperă în acest moment două situații. În primul caz prioritatea după care un nod este planificat, este definită ca fiind cea mai mare dintre cele rezultate pentru trace-urile din care nodul face parte. Acest lucru este perfect

rezonabil, deoarece planificarea unui nod se face conform situației sale cele mai defavorabile. Dintre mai multe noduri ready la același moment este ales cel cu prioritatea mai mare. Algoritmul este ușor de formulat și are complexitate redusă în timp. Dezavantajul său este că pot apare situații în care planificarea nodurilor se face folosind priorități ce nu corespund unor situații reale (situații care se exclud reciproc), ceea ce conduce la planificări mai lungi de cât cele obținute prin algoritmul următor.

Al doilea algoritm execută unele noduri conform priorităților lor pentru trace-ul curent, iar pe altele după cele pentru trace-uri mai importante. Metoda ordonează trace-urile după importanța pe care ele o au în obținerea unui timp maxim de execuție cât mai scurt. Trace-urile mai lungi sunt considerate mai importante ca și cele mai scurte. Când se planifică nodurile trace-ului curent, nodurile deja planificate în trace-uri mai importante sunt păstrate la timpii stabiliți, iar restul nodurilor sunt ordonate conform priorităților trace-ului curent. Astfel se evită planificarea unor noduri după prioritățile altor trace-uri, deși procesele deja planificate (noduri considerate prioritare pentru trace-uri) sunt ordonate conform priorităților din trace-urile mai importante. Putem spune că nodurile au asociate mulțimi de priorități conform trace-urilor din care ele fac parte. Algoritmul are complexitate în timp mai mare de cât cel precedent. Situațiile sale defavorabile sunt cele în care există trace-uri aproximativ la fel de lungi, în care același nod are odată prioritate mare și apoi mică. Planificarea acestor noduri conform unui caz îl dezavantajează pe celălalt, conducând în general la soluții mai lungi ca și algoritmul precedent.

Experimentele noastre arată că fiecare algoritm oferă în medie soluții de calitate bună. În anumite situații planifcările obținute sunt foarte bune și slabe pentru altele. Noi considerăm acest studiu doar ca un pas de plecare în dezvoltarea de euristici superioare pentru planificarea grafurilor legate, cu dependențe de date și de condiții.

5.6 Planificarea euristcă bazată pe liste de priorități

Paragraful curent își propune să prezinte o metodă de folosire a dependențelor de control în cadrul unei euristici de planificare bazată pe liste. Euristica atribuie fiecărui nod (proces, comunicare) din graf o prioritate unică. Când pe aceeași resursă, la același moment, mai multe noduri sunt pregătite pentru execuție, atunci este planificat cel cu prioritatea cea mai mare.

Fiecare nod este planificat conform cazului celui mai defavorabil care el poate apare. Cu cât nodul este mai constrâns din punctul de vedere al criteriului urmărit prin planificare, cu atât prioritatea sa este mai mare. Pentru a stabili cazul cel mai constrâns al nodului, trebuie să analizăm toate trace-urile din care el face parte. Prioritatea sa este atunci maximul priorităților ce îi corespund în trace-urile cărora el le aparține.

Pentru planificarea unui nod conform situației sale cele mai defavorabile există următoarea justificare. Executând nodul conform cazului cel mai constrâns favorizăm reducerea timpului pentru trace-urile lungi și defavorizăm trace-urile mai puțin constrânse, dar în general mai scurte. Dezavantajul metodei se manifestă prin faptul că la planificarea unui nod nu se verifică consistența trace-ului curent raportată la cel căruia îi corespunde prioritatea nodului. Consecința este că un nod poate fi planificat

într-un trace conform priorităţii sale rezultate pentru alt trace exclus mutual de cel curent. Un astfel de exemplu este al doilea caz discutat la sfârşitul paragrafului, iar soluţia obţinută cu acest algoritm este mai lungă de cât cea pentru algoritmul din 5.7.

Ca să planificăm nodurile unui graf cu dependenţe de date şi de control trebuie să răspundem în prealabil la două întrebări:

- În ce măsură intervin dependenţele de control în calculul priorităţii nodurilor?
- Cum se tratează pe parcursul planificării influenţa condiţiilor asupra setului de noduri executate?

Analiza capitolului 3 asupra priorităţilor folosite în euristicile bazate pe liste arată că prioritatea unui nod depinde de timpul său de execuţie, legarea sa, respectiv timpul de execuţie şi legarea predecesorilor şi succesorilor săi. *Mobilitatea* şi *urgenţa* sunt definite folosind timpul de execuţie al nodului, a predecesorilor şi succesorilor săi. *CP/MISF* depinde de timpul de execuţie al nodului şi succesorilor săi, precum şi de numărul succesorilor direcţi. *CP/MISF-ML* foloseşte timpul de execuţie şi legarea nodului şi a succesorilor săi, dar şi numărul succesorilor direcţi, iar *forţa* [PK93] este definită pe baza timpilor de execuţie şi a legării nodului, predecesorilor şi succesorilor săi. Prin urmare, exceptând *forţa*, *prioritatea unui nod este definită doar pe baza caracteristicilor nodurilor ce sunt în relaţie de ordine topologică cu nodul curent*.

Pe baza rezultatelor experimentale, am ales prioritatea nodurilor ca fiind cea definită de CP/MISF-ML. Aceasta oferă un compromis bun între calitatea soluţiilor, care de regulă sunt apropiate de optim, respectiv timpul de planificare, care este foarte scurt. Deşi analiza noastră se referă în continuare la CP/MISF-ML, rezultatele noastre sunt valabile pentru toate priorităţile definite folosind doar noduri ce sunt într-o relaţie de ordine topologică cu nodul implicat.

Revenind la întrebarea referitoare la calculul priorităţilor într-un graf cu dependenţe de date şi de control, discutăm următoarele cinci cazuri:

- Nodurile din alternative complementare nu se influenţează reciproc la calcularea priorităţilor, din cauză că între ele nu există o relaţie de ordine topologică. Regula este de "bun simţ" şi perfect justificată, deoarece nodurile din alternative complementare nu pot fi simultan active şi prin urmare execuţia lor nu se influenţează reciproc.
- Prioritatea unui nod dintr-o alternativă nu depinde de nodurile din alternative ortogonale. Spunem că două alternative sunt ortogonale când nici una nu este încuibată în cealaltă. Această proprietate a priorităţilor este o consecinţă tot a faptului că între aceste noduri nu apar relaţii de ordine topologică.
- Prioritatea unui nod dintr-o alternativă nu este influenţată de noduri negardate de condiţii şi care nu sunt predecesoare sau succesoare ale nodului implicat. Concluzia rezultă pe baza faptului că între nodurile negardate şi cel curent nu apar nici un fel de relaţii de ordine topologică.
- Pentru nodurile care preced un nod contiţional prioritatea trebuie astfel fixată încât să fie acoperitoare pentru cazul cel mai defavorabil. Deoarece în momentul în care se planifică un precedesor al nodului condiţional nu se cunoaşte valoarea condiţiei (deci nu se cunoaşte setul complet al nodurilor active), trebuie luate decizii care să fie acoperitoare inclusiv pentru cazul cel mai defavorabil. De aceea sunt analizate

toate seturile posibile de noduri succesoare, pentru fiecare este calculată prioritatea nodului, iar în final prioritatea nodului este definită ca maximul priorităților posibile. Aceasta caracterizează conform regulii folosite pentru definirea funcției prioritate, situația cea mai constrânsă a nodului curent.

Prioritățile nodurilor succesoare unui nod *join* nu depind de prezența construcției condiționale. Regula este motivată de faptul că prioritatea CP/MISF-ML a unui nod nu depinde de predecesorii săi.

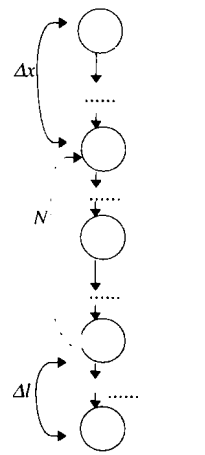


Figura 5.6 Planificarea grafurilor ciclice

Folosind observațiile anterioare prezentăm în continuare regula de calcul a priorității CP/MISF-ML pentru grafuri legate, cu dependențe de date și de control. Calculul priorităților folosește *graful simplificat al dependențelor de date* (graful în care apar doar dependențele de date), deoarece prioritățile nu depind de relațiile de control. Nodurile condiționale sunt reprezentate în graful simplificat al dependențelor de date ca noduri cu doi succesori direcți (succesorii direcți ai celor două alternative), iar nodurile *join* ca noduri cu doi predecesori, unul pentru fiecare alternativă. Algoritmul de calcul al priorității CP/MISF-ML este cel prezentat în capitolul 3 și se aplică grafului simplificat al dependențelor de date, obținându-se prioritățile nodurilor din graf. Aceste priorități sunt folosite în continuare de algoritmul de planificare.

Pentru grafuri ciclice (figura 5.6) calculul priorităților se face astfel:

- Prioritatea nodurile succesoare ale ciclului nu depinde de existența ciclului. Aceasta se datorește faptului că prioritatea CP/MISF-ML a unui nod nu depinde de cât de proprietățile succesorilor săi.
- Pentru nodurile care preced ciclul, prioritatea ia în considerare faptul ca în situația cea mai defavorabilă ciclul este executat de N (fixat în specificare) ori. Astfel, dacă Δp reprezintă lungimea unei iterații calculată după regula CP/MISF-ML atunci prioritatea nodului include valoarea $N \times \Delta p$. Formal, aceasta corespunde desfășurării implicite a ciclului (*loop unrolling*) de N ori. Prioritatea nodului este dată de relația

$$Prioritate = \Delta x + N \times \Delta p + \Delta l,$$

unde Δx este influența succesorilor săi care preced ciclul, iar Δl este cea a succesorilor care urmează ciclului.

- Pentru nodurile din interiorul ciclului, prioritățile sunt calculate dinamic, în timpul planificării, pe măsură ce sunt tratate iterațiile consecutive. Pentru iterația k, un nod din ciclu are prioritatea definită de formula

$$Prioritate = \Delta r + (N - k) \times \Delta p + \Delta l,$$

unde Δr corespunde nodurilor succesoare din iterația k.

Stabilirea nodurilor executate pentru setul curent de condiții se face pe parcursul algoritmului de planificare. Algoritmul folosește o listă a tuturor nodurilor pregătite pentru execuție, din care este întotdeauna selectat pentru execuție nodul cu prioritatea cea mai mare. La această listă sunt adaugate noduri noi, pe măsură ce ele devin pregătite pentru execuție. După planificarea unui nod condițional, algoritmul adaugă pe rând la lista nodurilor ready întâi nodurile din alternativa *adevărată*, apoi pe cele din alternativa *falsă*. Astfel se evită încărcarea resureselor cu noduri care nu pot fi simultan active și se planifică doar cele corespunzătoare setului curent de condiții active. După planificarea nodului condițional, algoritmul stabilește momentul pentru transmiterea condiției către alte procesoare din arhitectură.

Secțiunea următoare detaliază algoritmul nostru de planificare bazat pe liste de priorități pentru grafuri legate, cu dependențe de date și de control și executate pe arhitecturi multiprocesor

Algoritmul de planificare

Înainte de a prezenta algoritmul nostru de planificare bazat pe liste, enumerăm ipotezele de lucru folosite în planificarea nodurilor din graf:

- Arhitectura implementării este cea prezentată în paragraful 5.2.
- Planificarea folosește principiul și structurile de date descrise în paragraful 5.2.
- Comunicarea unei condiții durează întotdeauna o unitate de timp (acest interval este denumit *cuantă de timp*).
- Comunicarea unei condiții poate avea loc pe oricare din magistralele arhitecturii.
- O condiție este transmisă selectiv doar către procesoarele unde sunt luate decizii de planificare dependente de ea.
- Condițiile sunt transmise potrivit metodei ALAP pentru ușurarea stabilirii destinatarilor și pentru a "economisi" eventuale transmisii ulterioare de condiții.

Pseodocodul algoritmului de planificare bazat pe liste este:

```
se calculează gărzile nodurilor;
se calculează listele de influență ale nodurilor;
lista condițiilor parcurse = NULL;
mulțimea nodurilor ready = nodul rădăcină;
schedule (mulțimea nodurilor ready, lista condițiilor
      parcurse);
```

Algoritmul de planificare stabilește gărzile și lista de influență pentru fiecare nod din graf, inițializează `lista condițiilor parcurse` cu `NULL` și respectiv `lista`

nodurilor ready cu rădăcina grafului. Apoi apelează *schedule* pentru planificarea propriu-zisă a nodurilor.

```
void schedule(mulţimea nodurilor ready, lista condiţiilor parcurse) {
      while (mulţimea nodurilor ready nu este vidă şi nodul curent nu
calculează o condiţie) {
            nodul curent = următorul nod (mulţimea nodurilor
             ready);
            timp curent = momentul propus pentru planificare
                        (nodul curent);
            case (nodul curent) este {
            comunicare:
                  condiţii de transmis = determină mulţimea
                   condiţiilor ce trebuie transmise şi care sunt
                   calculate la timp curent;
                  planifică transmiterea lui condiţii de transmis;
                  condiţii necesare = stabileşte din lista
                   condiţiilor parcurse, pe cele necesare pentru
                   planificarea lui nodul curent;
                  timp condiţii necesare = momentul când condiţii
                   necesare devin cunoscute procesorului pe care este
                   părintele lui nodul curent;
                  planifică pe nodul curent la momentul max (timp
                   curent, timp condiţii necesare), corespunzător
                   coloanei lui condiţii necesare;
                  break;
            proces:
                  condiţii necesare = stabileşte din lista
                   condiţiilor parcurse, pe cele necesare pentru
                   planificarea lui nodul curent;
                  timp condiţii necesare = momentul când condiţii
                   necesare devin cunoscute procesorului pe care este
                   nodul curent;
                  planifică pe nodul curent la momentul max (timp
                   curent, timp condiţii necesare), corespunzător
                   coloanei lui condiţii necesare;
                  break;
            }
            elimină pe nodul curent din mulţimea nodurilor ready;
            if (nodul curent nu calculează o condiţie)
                  adaugă la mulţimea nodurilor ready, nodurile care
                   devin ready prin execuţia lui nodul curent;
      }

      if (nodul current != NULL şi el calculează o condiţie) {
            adaugă la mulţimea nodurilor ready nodurile ready ale
             alternativei true;
            adaugă la lista condiţiilor parcurse condiţia lui nodul
             curent cu valoarea true;
            schedule (mulţimea nodurilor ready, lista condiţiilor
             parcurse);
            adaugă la mulţimea nodurilor ready nodurile ready ale
             alternativei false;
            adaugă la lista condiţiilor parcurse condiţia lui nodul
             curent cu valoarea false;
            schedule (mulţimea nodurilor ready, lista condiţiilor
             parcurse);
```

Înainte de a detalia *schedule*, facem un scurt comentariu privind rolul lui *lista condițiilor parcurse*, precum și a lui *mulțimea nodurilor ready*. *mulțimea nodurilor ready* cuprinde în permanență toate nodurile pregătite pentru execuție pentru combinația curentă a condițiilor din graf. Prin pregătit pentru execuție înțelegem că toți predecesorii nodului au fost planificați. Pentru a ușura prelucrări ulterioare, această mulțime păstrează nodurile în ordinea celui mai devreme moment când ele și-ar putea începe execuția, adică primul moment când toți predecesorii lor devin executați. Acest moment reprezintă limita inferioară, înaintea căreia execuția nodului nu poate începe. Dacă există mai multe noduri cu aceeași limită inferioară, atunci ele sunt ordonate folosind prioritățile lor. *lista condițiilor parcurse* conține condițiile pentru care nodurile care le calculează au fost deja planificate. Această listă este folosită pentru a stabili care condiții trebuie transmise, precum și determinarea momentelor când aceste transmisii au loc.

În continuare detaliem rutina *schedule*. Din *mulțimea nodurilor ready* se selectează următorul nod executat. Pentru aceasta se caută primul moment când o resursă (procesor, magistrală) cu noduri ready la acel moment devine disponibilă. Dacă resursa are mai multe noduri ready se alege cel cu prioritate mai mare. Variabila *nodul curent* indică nodul ales la pasul curent pentru planificarea, iar *timp curent* momentul când această planificare ar putea avea loc. Dacă *nodul curent* reprezintă o comunicare, se stabilește dacă nu se impune în prealabil și o transmitere de condiții. Pentru aceasta se verifică dacă în *lista condițiilor parcurse* există la *timp curent* condiții calculate (adică nodul care le calculează și-a încheiat execuția) și care intervin în lista de gărzi sau de influență a nodurilor de pe procesorul destinatar al comunicației. În caz afirmativ se planifică transmisiile condițiilor necesare. Menționăm că dacă pe un același procesor sunt calculate mai multe condiții, atunci toate sunt transmise în același pachet de date. Pe baza *listei condițiilor parcurse* se stabilesc condițiile semnificative pentru planificarea lui *nodul curent*. Condițiile semnificative reprezintă condițiile din *lista condițiilor parcurse* calculate la *timp curent* și care apar în lista de gărzi sau de influență a nodurilor de pe procesorul părintelui comunicării. Pentru că *nodul curent* este o comunicare, planificarea sa este marcată în tabela de planificare a procesorului părintelui său. Transmisia condițiilor folosește de preferință altă magistrală de cât cea folosită pentru *nodul curent*, deoarece dorim să nu o întârziem pe aceasta. În cazul extrem, când restul magistralelor sunt ocupate, transmisia este inițiată chiar pe cea folosită de *nodul curent*. Aceasta va întârzia însă planificarea lui *nodul curent*, astfel că trebuie reactualizată valoarea lui *timp curent*.

Deoarece s-ar putea ca unele din condițiile din *condiții necesare* să fi fost subiectul unei transmisii de condiții, se stabilește momentul când ele sunt cunoscute procesorului părinte. *timp condiții necesare* memorează acest moment. *nodul curent* este planificat la maximul dintre *timp curent* și *timp condiții necesare*, pentru a asigura atât cunoașterea condițiilor necesare, precum și disponibilitatea resursei sale. Planificarea lui *nodul curent* apare în tabela de planificare a procesorului părintelui său în coloana condițiilor active în momentul planificării sale, adică coloana etichetată cu *condiții necesare*.

Dacă *nodul curent* indică un proces, secvența de planificare este simplificată. În mod asemănător cu cazul planificării comunicărilor, se determină lista *condiții necesare*. Această listă este formată din acele condiții din *lista condițiilor parcurse*, calculate la *timp curent* și care apar în lista de gărzi sau cea de influență a nodurilor de pe procesorul lui *nodul curent*. Acest set este setul activ de condiții pe procesorul lui *nodul curent* la *timp curent*. *nodul curent* este planificat la maximul dintre *timp curent* și *timp condiții necesare*, fiind marcată în tabela de planificare a procesorului său în coloana lui *condiții necesare*.

După ce *nodul curent* a fost planificat, el este eliminat din *mulțimea nodurilor ready*. Dacă procesul nu calculează o condiție, atunci în *mulțimea nodurilor ready* se adaugă nodurile ce devin pregătite pentru execuție odată cu executarea lui *nodul curent*. Pașii descriși anterior se repetă până la epuizarea tuturor nodurilor din *mulțimea nodurilor ready* sau până la planificarea unui nod condițional.

Dacă *nodul curent* este un nod condițional ce determină valoarea unei condiții, atunci sunt tratate separat alternativele pentru situațiile când condiția este adevărată și falsă. Când condiția este adevărată în *mulțimea nodurilor ready* se introduce doar succesorul direct din alternativa adevărat. În mod identic se întâmplă lucrurile pentru situația când condiția este falsă. Pentru ambele alternative, la *lista condițiilor parcurse* este adăugată condiția curentă împreună cu valoarea ei prezentă (adevărat sau fals) și momentul la care ea este disponibilă pentru a fi folosită apoi în planificarea nodurilor succesoare. Pentru fiecare din cele două alternative avem un apel recursiv al lui *schedule*, cu *mulțimea nodurilor ready* și *lista condițiilor parcurse* corespunzătoare. Astfel prin apelurile recursive evităm în mod natural tratarea simultană a nodurilor ce aparțin alternativelor complementare și care se exclud reciproc.

Exemple de planificare

Această secțiune exemplifică euristica noastră bazată pe liste în două situații: prima este una favorabilă, pentru care prioritatea este o măsură fidelă a "importanței" cu care nodurile trebuie planificate. A doua corespunde unui caz defavorabil, deoarece prioritățile nodurilor reflectă o situație ce nu poate apare în realitate (din cauza dependențelor de control). Intersant este faptul că cei doi algoritmi de planificare pentru grafuri cu dependențe de date și de control propuși de noi în lucrare, au rezultate complementare calitativ pentru cele două exemple: respectiv metoda bazată pe liste de priorități se comportă bine pentru primul exemplu și slab pentru al doilea, iar metoda bazată pe ajustarea și potrivirea planificărilor este inferioară pentru primul exemplu și superioară pentru al doilea.

Cazul favorabil

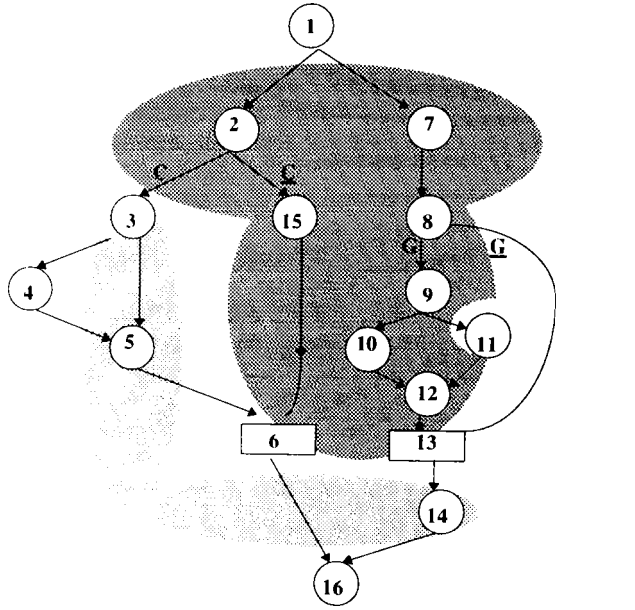


Figura 5.7 Graful folosit pentru ilustrarea unei situaţii favorabile algoritmului de planificare

Exemplul este ilustrat de figura de mai sus. Modul de legare al nodurilor pe resurse este indicat prin haşuri (procesele nehaşurate aparţin aceluiaşi procesor), iar comunicările, indiferent că sunt pentru date sau condiţii, folosesc aceeaşi magistrală. Tabelul următor prezintă timpii de execuţie pentru procesele şi comunicările grafului.

Activitate	Timp de execuţie	Activitate	Timp de execuţie
1	0	13	0
2	3	14	4
3	8	15	38
4	30	16	0
5	2	(2,3)	1
6	0	(5,6)	3
7	4	(3,4)	1
8	2	(4,5)	1
9	2	(13,14)	1
10	3	(9,11)	1
11	30	(11,12)	1
12	3		

Folosind graful simplificat al dependențelor de date, se calculează următoarele priorități pentru procese şi comunicări. Menționăm că prioritatea creşte invers cu valorea ei.

Activitate	Prioritate	Activitate	Prioritate
1	0	13	10
2	1	14	18
3	7	15	21
4	13	16	22
5	17	(2,3)	2
6	19	(5,6)	20
7	5	(3,4)	15
8	3	(4,5)	14
9	4	(13,14)	16
10	12	(9,11)	6
11	8	(11,12)	9
12	11		

La momentul 0 este planificat procesul 1. Tot la 0, 2 şi 7 devin pregătite pentru execuție. Deoarece 2 are prioritatea mai mare, el este cel planificat la 0. La momentul 3, 2 îşi termină execuția, astfel că valoarea lui **C** devine disponibilă. Dacă **C** este adevărată, atunci trebuie planificată comunicarea între 2 şi 3, dar pentru că 3 are **C** în garda sa şi este plasat pe alt procesor de cât cel ce calculează pe **C**, se impune acum şi transmiterea valorii lui **C** altor resurse. Prin urmare, la momentul 3 se planifică comunicarea valorii lui **C**, iar la 4 comunicarea (2,3). Procesul 3 porneşte imediat după încheierea lui (2,3), adică la 5. Tot la 3, pentru că se eliberează procesorul, este planificat 7 şi în continuarea lui 7 este lansat în execuție nodul 8. La momentul 9, 8 îşi termină execuția, iar valoarea condiției **G** este calculată. Dacă valoarea lui **G** este adevărat, atunci la 9 este planificat 9. 9 se termină la 11 şi pentru că este necesară cunoaşterea valorii lui **G** pe procesorul lui 11, se impune transmiterea valorii lui **G**. Prin urmare, comunicarea lui **G** este planificată la 11. Comunicare (9,11) începe imediat în continuare, adică la 12, iar 11 este planificat la 13. După terminarea lui 9, pe acelaşi procesor, este pornit 10. La momentul 13 pe procesorul 2 se termină 3. Prin urmare, comunicarea (3,4) este planificată la 13, iar procesul 4 la 14. Deoarece 11 şi 14 sunt legate de procesorul hardware, execuția lor are loc concurent. 11 se termină la 43, urmând ca tot atunci să fie planificată comunicarea (11,12). La 44, odată cu încheierea lui 4, urmează (3,4). În continuare sunt planificate 12 la 44, 13 la 47 şi (13,14) tot la 47. 5 porneşte la 45, iar (5,6) la 48. Procesul 14 este planificat la 48 şi 6 la 51. Pentru combinația de condiții **CG**, execuția se încheie la momentul 52, odată cu planificarea lui 16.

Revenim la momentul 9, când 8 îşi încheie execuția şi presupunem că **G** este fals. Atunci este planificat 13, iar după terminarea lui, pentru că se impune comunicarea către 14 (care are pe **G** în lista lui de influență), trebuie transmisă în prealabil valoarea lui **G**. Deci la 9 este planificată transmiterea valorii lui **G**, după care urmează (13,14) la 10. Pe procesorul 1, la 13 este planificată comunicarea (3,4), astfel că 4 porneşte la 14. Execuția lui 4 se încheie la 44, când este planificat (4,5). Tot la 13, odată cu încheierea

lui 3, este planificat 14 (procesul fiind ready de la 11). La 45 începe 5, la 47 (5,6), iar 6 la 50. Pentru combinația **CG** timpul total de execuție al grafului este de 50 unități de timp.

Următoarea situație este pentru **C** fals. La încheierea lui 2, procesele pregătite pentru execuție sunt 15 și 7. 7 este cel planificat pentru 3, pentru că are prioritatea mai mare. La 7, 8 devine pregătit și având prioritate mai mare ca și 15, este următorul lansat în execuție. Dacă **G** este adevărat, la 9 este planificat procesul 9, iar la 11 procesul 10. Tot la 11 se planifică și transmiterea lui **G**, deoarece urmează comunicarea către 11 (are pe **G** în garda sa). La 12 începe (9,11), iar după terminarea lui 10 la 14, este planificat procesul 15. Procesul 11 este executat în hardware începând cu 13 și se încheie la 43, când se planifică (11,12). Deoarece procesorul este ocupat de 15 până la momentul 52, 12 este planificat doar la 52. La 55 urmează 13 și tot la 55 comunicarea (13,14) și procesul 6. În final, procesul 14 este planificat la momentul 56, iar execuția grafului pentru combinația **CG** este terminată la momentul 60.

Revenim la momentul când se încheie 8, și presupunem ca **G** este fals. Prin urmare, la 9 este planificat 13, iar apoi transmiterea condiției **G**. La 10 este comunicat (13,14). Tot la 9, pentru ca este eliberat procesorul 1, este planificat procesul 15, care se încheie la 47, moment în care 6 își începe execuția. Pe procesorul 2, 14 este planificat la 11. Execuția grafului durează 47 unități de timp pentru **CG**. Tabela finală de planificare are următoarea structură

Procesorul 1

	1	C	CG	CG	C	CG	CG
0	1						
0	2						
3		transm. lui C					
3					7		
3		7					
4		(2,3)					
7		8					
7					8		
9			9				
9				13			
9				transm. lui G			
9						9	
9							13
9							transm. lui C,G
10				(13,14)			15
10							
11			transm. lui				(13,14)

			G				
11			10				
11						transm. lui C,G	
11						10	
12			(9,11)				
12						(9,11)	
14						15	
44			12				
47			13				
47			(13,14)				
47							6
50				6			
51			6				
52						12	
55						13	
55						(13,14)	
55						6	

Procesorul 2

	1	C	CG	CG	C	CG	CG
5		3					
11							14
13			(3,4)				
13				(3,4)			
13				14			
45			5				
45				5			
47				(5,6)			
48			(5,6)				
48			14				
56						14	

Procesorul 3

	1	C	CG	CG	C	CG	CG
13			11				
13						11	
14			4				
14				4			
43			(11,12)				
43						(11,12)	
44			(4,5)				
44				(4,5)			

După cum rezultă din tabele, timpul de execuție al grafului pentru cazul cel mai defavorabil (pentru combinația **CG**) este 60. După cum se va prezenta ulterior, dacă

aplicăm metoda ajustării şi reunirii planificărilor, rezultatul ar fi 66, deci inferior. Explicaţia comportării favorabile a acestei euristici este următoarea. Graful conţine două condiţii **C** şi **G**, neîncuibate una în alta şi prin urmare, toate combinaţiile lor de valori sunt valide. În graf nu apar noduri care să aibă atât succesori cu **C** în gardă, cât şi succesori cu **G** în gardă. Prin urmare, priorităţile sunt o măsură fidelă a importanţei cu care procesele trebuie planificate. Dacă în graf ar exista şi noduri cu proprietăţile menţionate, atunci prioritatatea lor poate reflecta situaţii excluse de combinaţiile valide ale condiţiilor, ceea ce perturbă calitatea planificării finale. O astfel de situaţie apare în următorul exemplu.

Pentru C, transmiterea valorilor lui **C** şi **G** se face în acelaşi pachet de date, ceea ce "economiseşte" o unitate de timp.

Cazul defavorabil

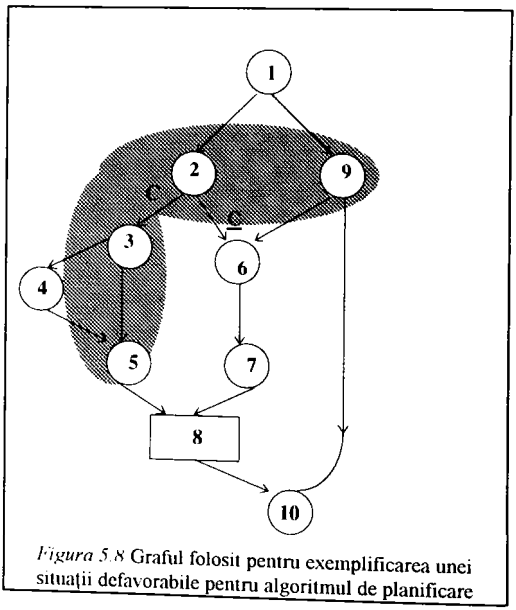


Figura 5.8 Graful folosit pentru exemplificarea unei situaţii defavorabile pentru algoritmul de planificare

Exemplul este ilustrat de figura 5.8. Modul de legare al proceselor pe resurse este indicat prin haşuri (procesele nehaşurate aparţin aceluiaşi procesor), iar comunicările, indiferent că sunt pentru date sau condiţii, folosesc aceeaşi magistrală. Tabelul următor prezintă timpii de execuţie pentru procesele şi comunicările grafului.

Activitate	Timp de execuţie	Activitate	Timp de execuţie
1	0	9	20
2	3	10	0
3	35	(3,4)	1
4	25	(4,5)	1

5	10	(5,8)	1
6	18	(9,6)	1
7	20	(2,6)	1
8	0		

Folosind graful simplificat al dependențelor de date se calculează următoarele priorități pentru procese şi comunicări. Menționăm că prioritățile cresc invers cu valorea lor.

Activit ate	Prioritate	Activit ate	Prioritate
1	1	9	0
2	2	10	14
3	3	(3,4)	6
4	7	(4,5)	8
5	9	(5,8)	12
6	10	(9,6)	5
7	11	(2,6)	4
8	13		

Procesul 1 este primul lansat în execuție la momentul 0. Tot la 0 devin pregătite pentru execuție 2 şi 9. Cum 9 are prioritate mai mare, el este cel planificat la 0. Execuția lui 9 durează până la momentul 20, când este pornit 2. Tot la 20 comunicarea (9,6) este planificată pe magistrala arhitecturii. La 23, procesul 2 îşi încheie execuția, astfel că valoarea condiției **C** devine disponibilă. Dacă **C** este adevărată, atunci sunt active procesele 3, 4, 5 şi comunicările (3,4), (4,5) şi (5,8). Pentru **C** falsă devin active procesele 6,7 şi comunicarea (2,6). Prin urmare, pentru **C** rezultă următoarea planificare: procesul 3 este pornit la 23, iar execuția lui se încheie la 58. Acuma ar trebui planificată comunicarea (3,4), dar 4 este plasat pe alt procesor de cât cel ce calculează **C** şi are pe **C** în garda sa. Prin urmare, se impune transmiterea valorii lui **C**, activitate ce este planificată la 58, urmând ca la 59 sa înceapă comunicarea (3,4). La 60 este planificat 4, iar imediat după terminarea sa, la 85, urmează (4,5). Procesul 5 este planificat la momentul 86, iar comunicarea (5,8) la 96. În fine, procesul 8 începe la 97 şi tot la 97 porneşte 10, fapt ce marchează sfârşitul planificării grafului pentru **C**. Revenind la terminarea lui 2, pentru valoarea **C** a condiției **C**, la momentul 23 este transmisă valoarea condiției, iar la 24 are loc comunicarea (2,6). Procesul 6 începe la 25, 7 la 42, iar 8 la 63. Durata execuției grafului este 63 pentru **C**. Următoarele tabele reprezintă tabelele de planificare pentru procesoarele din arhitectură:

Procesorul 1

	1	C	C
0	1		
0	9		
20	2		
20	(9,6)		
23			condiția C
23		3	

24			(2,6)
58		condiția C	
59		(3,4)	
86		5	
96		(5,8)	

Procesorul 2

	1	C	C
25			6
43			7
60		4	
63			8
85		(4,5)	
97		8	

În concluzie, timpul total de execuție al grafului pentru situația cea mai defavorabilă este 97. După cum va rezulta ulterior, dacă planificarea grafului s-ar face folosind metoda ajustării și potrivirii planificărilor, timpul pentru cazul cel mai defavorabil ar fi 76. Principala cauză pentru această diferență provine de la prioritatea asociată lui 9, cu implicațiile negative de rigoare privind momentul când 9 este planificat. 9 are prioritatea 0, rezultată din faptul că 6, 7 și 8 îi sunt succesori, ceea ce se întâmplă în cazul **C**. Dar timpul 97 rezultă pentru **C**, când 6,7 și 8 nu sunt activi, astfel că prioritatea lui 9 ar trebui sa fie mai mică de cât cele ale lui 2 și 3. Prin urmare, această metodă de planificare poate folosi pentru o anumită combinație de condiții priorități corespunzătoare unor combinații excluse de ea. Bineînțeles că acest fapt este o limitare a euristicii noastre de planificare. A doua observație se referă la faptul că (9,6) este planificat indiferent dacă 6 este activat sau nu. Aceasta nu reprezintă o eroare a metodei, fiind consecința faptului că în momentul planificării lui (9,6), informația privind activarea sau nu a lui 6 nu este încă disponibilă.

Rezultate experimentale

Setul de experimente definit în acest paragraf a fost folosit și pentru experimentarea algoritmului de planificare prin ajustarea și potrivirea planificărilor, prezentat în paragraful 5.7. În experimente am folosit un număr total de 5400 de grafuri, câte 1800 pentru fiecare din dimensiunile 60, 80 și 120 de noduri. Grafurile au o structură aleatoare, nodurile fiind interconectate între ele aleator cu o probabilitate de 0.8. Timpii de execuție asociați proceselor sunt distribuiți uniform în intervalul 1..20, iar timpii pentru comunicări sunt distribuiți uniform în intervalul 1..3 .

Fiecare tip de experiment se deosebește prin numărul nodurilor în graf, modul de structurare al condițiilor din graf - funcție de care rezultă numărul trace-urilor, modul de grupare a nodurilor în alternative, arhitectura folosită pentru execuția nodurilor și maniera de legare a nodurilor pe resurse (procesoare și magistrale). Pentru fiecare tip distinct de experimente s-au generat automat un număr de 30 de grafuri.

Toate grafurile cuprind 5 noduri condiționale, care sunt structurate în toate modurile posibile de încuibare. De aici rezultă 5 structuri diferite ale grafurilor, pentru un număr de 10, 12, 18, 24 și 32 de trace-uri. Nodurile pot fi repartizate în mod diferit la structurile condiționale ale grafului. Experimentele noastre au inclus cazul când nodurile apar doar într-una din cele două alternative complementare și situația când ele sunt distribuite cu o probabilitate egală la fiecare pereche de alternative condiționale. Pentru fiecare dimensiune de noduri a grafurilor s-au experimentat 3 tipuri de arhitecturi. Ele au fost astfel alese încât numărul mediu de noduri/resursă să fie același și să reflecte situațiile pentru resurse puternic încărcate, mediu încărcate și slab încărcate de activități. Pentru grafurile cu 60 de noduri s-au folosit arhitecturi cu 2 procesoare-1 magistrală, 3 procesoare-1 magistrală și 6 procesoare-4 magistrale. Pentru grafuri cu 80 de noduri arhitecturile utilizate cuprind 2 procesoare-1 magistrală, 4 procesoare-2 magistrale și 8 procesoare-4 magistrale. Grafurile cu 120 de noduri au fost experimentate pentru arhitecturi cu 3 procesoare-2 magistrale, 6 procesoare-4 magistrale și 12 procesoare-8 magistrale. Fiecare arhitectură folosită conține 1 procesor hardware. Legarea nodurilor pe resurse este de două feluri. În primul caz nodurile sunt repartizate aleator pe resurse, iar în cazul al doilea s-a folosit o legare inteligentă. Nodurile au fost astfel mapate pe resurse încât să rezulte un număr cât mai mic de comunicări între procesoarele din sistem (lanțurile de procese au fost legate de aceeași resursă), să fie încurajate activitățile paralele din sistem (lanțurile distincte de noduri au fost legate la resurse diferite) și încărcarea resurselor să fie aproximativ identică.

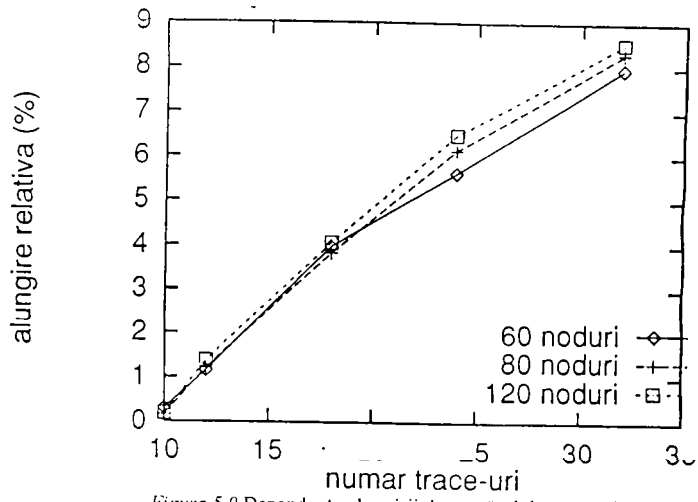


Figura 5.9 Dependența alungirii de numărul de trace-uri

Parcurgerea experimentelor și prelucrarea statistică a rezultatelor a fost făcută automat, folosind un calculator PC 486 DX2 la 80 MHz cu sistemul de operare Linux.

Primul gen de concluzii se referă la diferența dintre lungimea planificării unui GDC și lungimea celui mai lung trace, planificat individual prin euristica CP/MISF-ML. Lungimea planificării unui GDC este definită prin cel mai lung trace obținut prin planificarea GDC-ului total cu algoritmul nostru. Graficul din figura 5.9 prezintă

alungirea relativă procentuală definită ca fiind raportul în procente între diferenţa precedentă şi lungimea celui mai lung trace, planificat individual.

Timpii medii de execuţie (în secunde) ai algoritmului sunt ilustraţi în graficul din figura 5.10.

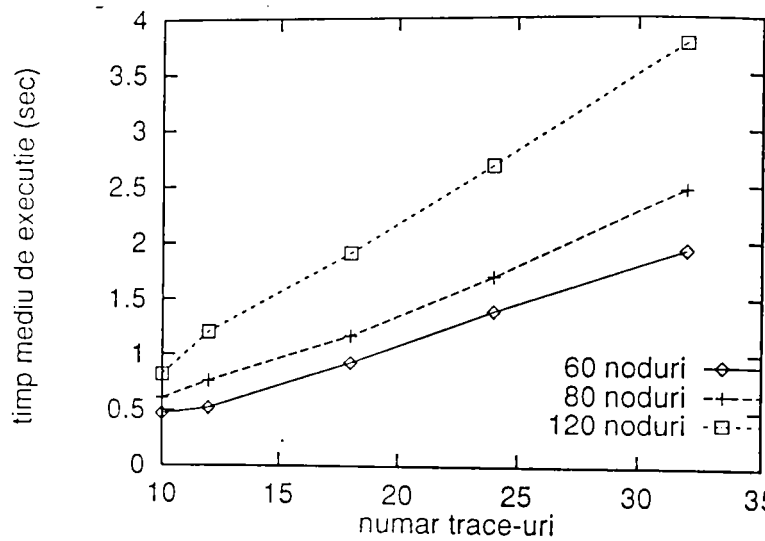


Figura 5.10 Dependenţa timpului de execuţie de numărul de trace-uri

În concluzie algoritmul de planificare bazat pe liste de priorităţi găseşte în timpi de execuţie scurţi planificări bune, cu alungiri medii de până la 9%. Acest algoritm este un compromis bun între viteză şi calitate. O posibilitate de îmbunătăţire a algoritmului este aceea de a planifica nodurile după maximul priorităţilor lor, apărute pentru trace-uri acoperite de valorile condiţiilor cunoscute în momentul planificării lor. Spunem că un trace este *acoperit* de un set de valori ale condiţiilor (nu neapărat pentru toate condiţiile din graf), dacă acest set este o submulţime a mulţimii de condiţii ce execută trace-ul.

5.7 Planificarea grafurilor cu dependente de date şi de control prin ajustarea şi potrivirea planificării trace-urilor

Fiecărui graf cu dependenţe de date şi de control îi corespunde o mulţime de trace-uri, generate pentru toate combinaţiile posibile de valori ale condiţiilor din graf. Unei combinaţii îi corespunde un trace specific şi unic, care este un subgraf al grafului global. Între nodurile acestui subgraf există *doar* dependenţe de date şi prin urmare, ele pot fi planificate folosind unul din algoritmii indicaţi în capitolul 3. În urma planificării trace-ului luat individual rezultă timpul său de execuţie, care este o măsură a complexităţii activităţilor implicate de trace.

Discuţia ce urmează se referă la două moduri de planificare a trace-urilor. Primul este cel când trace-ul este planificat individual, doar pe baza dependenţelor sale de date şi fără a face nici un fel de referire la GDC-ul din care el face parte, iar al doilea consideră că trace-ul este planificat în cadrul algoritmului de planificare a întregului graf. Dacă nu facem nici o precizare suplimentară, atunci prin planificarea unui trace înţelegem situaţia a doua.

Timpul de execuţie al grafului global este stabilit de timpul de execuţie al trace-ului cel mai lung. După cum s-a detaliat în paragraful precedent, unele decizii de planificare privind nodurile unui trace influenţează şi planificările altor trace-uri. Pentru că dorim un timp de execuţie minim al grafului global, este de "bun simţ" ca o euristică de planificare să avantajeze trace-urile lungi, cu riscul de a le defavoriza pe cele scurte. Orice decizie de a planifica un nod conform priorităţii sale pentru un alt trace de cât cel curent (lucru necesar pentru a potrivi planificările a două trace-uri), înseamnă posibilitatea de a întârzia noduri prioritare, ceea ce în final va duce la alungirea timpului de execuţie a trace-ului curent. De aceea, o euristică ar trebui să planifice cât mai multe noduri conform priorităţilor pentru trace-urile lungi, astfel că alungirea acestora să fie cât mai mică. Deşi trace-urile scurte se vor alungi, probabilitatea ca lungimea lor să depăşească pe cea a trace-urilor lungi rămâne mică.

Acelaşi nod poate aparţine la trace-uri diferite şi pentru fiecare să aibă o importanţă (prioritate) specifică la planificare. Astfel un nod dintr-un GDC are ataşată mai degrabă o mulţime de priorităţi de cât o valoare singulară. Euristica paragrafului 5.6 asimilează mulţimea de priorităţi a unui nod prin elementul ei maxim, astfel că fiecare nod este planificat după situaţia cea mai defavorabilă în care el apare. Însă, marele dezavantaj al euristicii provine din faptul că în selecţia următorului nod executat sunt comparate priorităţi pentru trace-uri diferite. Deoarece nu se păstrează nici o "legătură" între trace-ul curent şi prioritatea nodurilor, există, din cauza regulii după care sunt atribuite priorităţi nodurilor, şanse ca planificarea rezultată pentru un trace să fie alungită faţă de cea obţinută pentru trace-ul individual.

Bazându-ne pe cele prezentate până acuma putem concluziona că strategia de planificare trebuie să ajute cazurile defavorabile (trace-urile lungi), păstrându-le nemodificate. Bineânţeles că alte trace-uri vor fi dezavantajate şi alungite corespunzător. Deasemenea, o euristică de planificare a GDC trebuie să minimizeze impactul nodurilor planificate după prioritatea altor trace-uri asupra alungirii trace-ului curent. Totuşi, anumite cazuri nu pot fi eliminate, pentru că ele ţin de natura

dependențelor de control. Un trace este cunoscut complet abia în momentul când se cunosc valorile tututror condițiilor din graf. Astfel, când se execută predecesorii nodurilor condiționale nu se poate decide care din trace-uri va fi cel urmat, dar se poate indica o mulțime din care acesta face parte.

Presupunem situația intermediară de pe parcursul planificării, în care o parte din condițiile grafului sunt cunoscute, restul urmând a fi calculate ulterior. Nodurile rămase neplanificate urmează a fi executate potrivit scenariului cel mai defavorabil. Acesta corespunde trace-ului cel mai lung dintre cele ce pot fi parcurse pentru condițiile cunoscute. Lungimea trace-ului reprezintă timpul său total de execuție, rezultat în urma planificării sale individuale. După ce s-a identificat cazul cel mai dezavantajos posibil, nodurile sunt planificate în continuare după prioritățile pe care acesta le atribuie lor. În momentul în care valoarea unei condiții noi devine cunoscută, mulțimea trace-urilor ce pot fi parcurse este restrânsă corespunzător. Apoi, dintre ele este identificată situația cea mai dezavantajoasă, iar prioritățile ei sunt cele folosite pentru planificarea nodurilor, până când valoarea unei alte condiții va fi calculată.

Felul în care se face *adaptarea* priorităților pe parcursul planificării ajută trace-urile lungi, încercând să execute preferențial nodurile potrivit priorităților fixate de ele. Ea încearcă să păstreze trace-urile lungi neschimbate sau să le alungească cât mai puțin și să le potrivească pe cele scurte conform lor. Euristica consideră că un trace lung este mai prioritar ca și unul scurt. Folosirea lungimii trace-urilor ca și criteriu de "ordonare" a trace-urilor este în mod cert discutabilă. Ea este simplu de definit, de calculat și de aplicat. Deasemenea, ea derivă din principiul de "bun simț" de a ajuta cu precădere trace-urile lungi. Însă dacă două trace-uri au lungimi egale, atunci secvența în care ele sunt considerate este aleatoare. Din păcate, rezultatele planificării GDC-urilor totale nu sunt aceleași pentru orice ordine de prelucrare a trace-urilor, ceea ce impune necesitatea de a folosi criterii secundare de ierarhizare a trace-urilor. Deasemenea, alungirea unui trace, ca urmare a planificării întregului graf, depinde de măsura în care ordinea sa de planificare contravine celei stabilite pentru trace-ul luat individual. Alungirile depind de numărul nodurilor prioritare întârziate în planificarea trace-ului curent de noduri și care sunt executate conform priorităților lor din alte trace-uri. Cercetările noastre viitoare se vor concentra pe găsirea de criterii de ierarhizare a trace-urilor alternative și/sau suplimentare la lungime. La această oră credem că un astfel de criteriu ar putea fi suma între lungimea trace-ului și o estimare a alungirii lui ca urmare a planificării anterioare a altor trace-uri.

Următoarea secțiune detaliază algoritmul de planificare și discută pseudocodul său.

Algoritmul de planificare prin ajustarea și potrivirea planificării trace-urilor

Înainte de a detalia pseudocodul acestui algoritm de planificare trebuie să decidem modul în care sunt comunicate condițiile grafului spre resursele ce le folosesc în planificare. Reamintim că paragraful 5.4 precizează două strategii de transmitere a condițiilor. Prima (pe care am numit-o ASAP) încearcă să transmită fiecare condiție cât mai repede după ce ea a fost calculată către toate procesoarele arhitecturii. A doua (numită ALAP) amână transmiterea condiției cât este posibil, fără însă a întârzia

planificarea nodurilor ce depind de ea. Această strategie încurajează transmiterea selectivă a condițiilor doar spre procesoarele unde ea este folosită în decizii de planificare.

Această euristică de planificare a fost definită folosind strategia ASAP de comunicare a condițiilor, deși și cealaltă metodă ar fi putut fi aplicată.

Euristica de planificare începe prin a calcula toate trace-urile ce rezultă pentru un graf cu dependențe de date și de control. Pentru aceasta, ea stabilește toate combinațiile posibile de condiții și pentru fiecare din ele identifică nodurile activate de ea. Apoi, fiecare trace este planificat individual folosind algoritmul CP/MISF-ML, obținându-se prioritățile nodurilor, ordinea lor de execuție pentru acel trace, precum și timpul total de execuție al trace-ului. Deoarece trace-ul cuprinde și transmiterea condițiilor, pe măsură ce ele sunt calculate, către procesoarele arhitecturii, algoritmul CP/MISF-ML suferă o mică modificare față de varianta prezentată în capitolul 3. În momentul în care un nod condițional a fost executat, pe primul bus ce devine liber începând cu momentul terminării execuției nodului condițional, este planificată transmiterea condiției sale.

Planificările trace-urilor, împreună cu lungimile lor și combinațiile de condiții care le activează sunt memorate în lista *lista_trace_urilor_planificate*. Parcurgând această listă se identifică planificarea cea mai lungă, care apoi este primul subiect prelucrat de către euristica de planificare.

Pseudocodul algoritmului de planificare prin ajustarea și potrivirea planificărilor este:

```
calculează mulțimea tuturor combinațiilor posibile de condiții;
for (fiecare combinație posibilă de condiții) {
   trace_curent = trace-ul activat de combinația
   curentă de condiții;
   planifică trace_curent folosind CP/MISF-ML;
   adaugă la lista_trace_urilor_planificate planificarea lui
   trace_curent, împreună cu lungimea sa și combinația de
   condiții pentru care el este executat;
}
planificare_curentă = planificarea cea mai lungă dintre cele
din lista_trace_urilor_planificate;
Build_schedule (NULL, NULL, NULL);
```

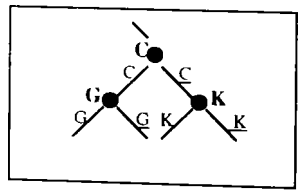
Principiul euristicii este de a favoriza trace-urile lungi (modificându-le cât mai puțin) în defavoarea celor scurte. Astfel, în urma planificării întregului graf, alungirile trace-urilor lungi față de lungimile lor pentru planificările individuale trebuie să fie cât mai mici. Este absolut rezonabil să considerăm că alungirea unui trace este cu atât mai mare, cu cât sunt mai multe noduri planificate în altă ordine de cât cea a planificării individuale. Orice nod planificat într-o altă ordine întârzie noduri cu prioritate mai mare, iar efectul rezultat este alungirea corespunzătoare a planificării trace-ului. De aceea, dorim să păstrăm cât mai multe noduri ale trace-urilor lungi în ordinea în care au fost planificate pentru trace-urile luate individual, în speranța că și alungirea lor va fi astfel mai mică.

Argumentația anterioară explică de ce euristica selectează în primul pas trace-ul cel mai lung, pe care îl păstrează neschimbat. Nodurile trace-ului cel mai lung sunt parcurse și planificate în ordinea stabilită de planificarea sa, când acesta este luată individual. Când

se întâlnește un nod condițional există două variante de a continua euristica, ce corespund celor două alternative ale nodului condițional. În acest moment, algoritmul de planificare continuă, bazându-se pe faptul că se cunosc valorile pentru o parte a condițiilor din graf. Pentru valorile condițiilor cunoscute se identifică trace-urile care ar putea fi parcurse și mulțimea condițiilor ce activează pe fiecare din aceste trace-uri conține pe cea a condițiilor deja cunoscute. Dintre aceste trace-uri euristica îl găsește pe cel care este cel mai important (cel mai lung) și continuă cu prelucrarea lui.

Pentru simplitatea prezentării notăm cu **C** condiția calculată de nodul condițional curent. Euristica continuă alegând prima acea alternativă care duce la trace-ul cel mai lung dintre cele posibile. Considerăm că pentru **C** adevărat este selectată această alternativă. După ce au fost prelucrate (planificate) toate trace-urile ce sunt activate pentru combinații de condiții în care **C** este adevărat, euristica continuă cu tratarea trace-urilor parcurse pentru combinații de condiții cu **C** fals. Aceste trace-uri sunt mai scurte de cât cel mai lung din alternativa complementară, motiv pentru care ele sunt considerate mai puțin prioritare pentru algoritmul de planificare, de cât cel mai lung al alternativei complementare. Nodurile planificate anterior celui condițional sunt executate în ordinea stabilită de trace-urile mai importante, iar aceasta poate fi diferită de ordinea fixată atunci când trace-urile au fost planificate individual. Condiția **C** este denumită *bifurcație* [ED97] și ea *distinge* trace-urile activate pentru **C** adevărat de cele executate pentru **C** fals. Cunoașterea unei bifurcații permite individualizarea mulțimii de trace-uri în cele două submulțimi pe care ea le distinge.

Următorul exemplu are rolul de aplica strategia de planificare prezentată pentru un caz simplu, dar relevant. Graful dependențelor de date și de control discutat este prezentat simplificat de următorul desen.



Graful cuprinde trei noduri condiționale, iar condițiile corespunzătoare sunt notate cu **C**, **G** și **K**. Numărul trace-urilor posibile este patru și este stabilit de combinațiile de valori posibile ale lui **C**, **G** și **K**, adică {**CG**, **C**$\underline{\mathbf{G}}$, $\underline{\mathbf{C}}$**K**, $\underline{\mathbf{CK}}$}. **C** este bifurcația pentru mulțimile de trace-uri {**CG**, **C**$\underline{\mathbf{G}}$} și {$\underline{\mathbf{C}}$**K**, $\underline{\mathbf{CK}}$}, **G** este bifurcația pentru trace-urile **CG** și **C**$\underline{\mathbf{G}}$, iar **K** pentru $\underline{\mathbf{C}}$**K** și $\underline{\mathbf{CK}}$. Euristica planifică pentru început, individual, fiecare din cele patru trace-uri posibile. Astfel rezultă estimări ale complexității computaționale pentru fiecare din cele patru trace-uri. Apoi, euristica selectează trace-ul cu planificarea cea mai lungă pentru a-l prelucra primul. Presupunem că acest trace ar fi cel activat pentru combinația de condiții **C**$\underline{\mathbf{G}}$.

Nodurile acestui trace sunt parcurse în ordinea stabilită prin planificarea sa individuală, până când se ajunge la un nod condițional. Primul nod condițional întâlnit este cel care calculează condiția **C**. Acuma alegem alternativa ce duce spre trace-ul cel mai lung, deoarece acesta trebuie prelucrat primul. De aceea, ajunsă în nodul **C**, euristica de planificare continuă selectând prima alternativa pentru **C** adevărat. Abia după au fost

parcurse toate trace-urile activate pentru combinații ce cuprind pe **C** adevărat, este rândul să fie prelucrate trace-urilor executate pentru combinații cu **C** fals. Nodurile alternativei **C** adevărat sunt parcurse în ordinea planificării individuale a trace-ului cel mai lung dintre cele activate, adică a lui **CG**. Când se ajunge la nodul **G**, operațiile executate pentru nodul **C** sunt repetate.

Din nou, trace-ul cel mai lung este ales primul, ceea ce pentru exemplul nostru înseamnă alternativa pentru **G** fals. Algoritmul continuă cu pargurea nodurilor rămase din alternativa **CG**, până când aceasta se termină. Din felul în care s-a făcut planificarea nodurilor pentru primul trace parcurs complet, care este şi cel mai lung din graf, se constată că ordinea sa este identică cu cea pentru planificarea sa individuală, astfel că pentru acest caz alungirea rezultată este 0.

După ce s-a epuizat trace-ul **CG**, euristica de planificare revine în situația existentă pentru nodul **G** şi studiază alternativa complementară. Trace-ul ce corespunde acestui caz este cel activat de combinația de condiții **CG**. Nodurile planificate înaintea calculării lui **G** sunt executate într-o ordine favorabilă trace-ului **CG**, care este considerat mai prioritar de cât cel curent. Această ordine poate diferi de cea considerată optimă pentru **CG**, motiv pentru care planificarea sa poate fi alungită în urma ajustării sale conform planificării pentru **CG**

În acest moment au fost planificate toate trace-urile executate pentru combinații de condiții care-l cuprind pe **C** adevărat. De aceea, euristica revine în situația existentă în momentul în care a fost planificat nodul **C** şi continuă cu prelucrarea trace-urilor activate pentru **C**. Primul prelucrat este cel mai lung dintre cele pentru **CK** şi **CK**. Să presupunem că acesta ar fi cel corespunzător lui **CK**. Nodurile precedente lui **C** sunt deja executate în ordinea stabilită de planificarea lui **CG**. De aceea, planificarea lui **CK** trebuie ajustată conform noii situații, când predecesorii lui **C** sunt executați în ordinea stabilită pentru **CG**. Nodurile succesoare lui **C** sunt parcurse în ordinea ajustată, până când este întâlnit nodul **K**. Din nou este aleasă prima alternativa cea mai lungă, adică cea pentru **CK** şi prelucrarea continuă cu ea. După aceasta, euristica revine în nodul **K** pentru a parcurge trace-ul rămas, adică cel activat pentru **CK**. Pentru acesta, toți predecesorii lui **K** sunt deja planificați: predecesorii lui **C** ca urmare a parcurgerii trace-ului **CG**, iar predecesorii lui **K** care sunt şi succesori ai lui **C**, ca urmare a prelucrării lui **CK**.

Acest exemplu arată felul în care euristica de planificare parcurge în adâncime graful dependențelor de date şi de control, ajustând planificările trace-urilor mai scurte conform celor mai lungi.

Planificarea trace-ului selectat cuprinde o fază de *ajustare* a sa. Când un trace este ajustat conform altora, se cunoaşte condiția de bifurcație ce îl deosebeşte de ele. Deoarece distincția între grupurile de trace-uri se face prin condițiile de bifurcație, pentru a evita apariția conflictelor este necesar să păstrăm pe fiecare resursă planificările trace-urilor de ajustare, până la momentele când condițiile de bifurcație devin cunoscute lor. Nodurile ajustate sunt executate după prioritățile ataşate lor de trace-urile de ajustare, acesta fiind "compromisul" făcut în dauna trace-urilor mai scurte pentru a alungi cât mai puțin pe cele lungi. Nodurile din trace, rămase

neplanificate sunt executate ASAP, în ordinea stabilită atunci când trace-ul a fost planificat individual. În urma acestei ajustări, trace-ul curent nu va genera conflicte în tabela de planificare cu trace-urile deja planificate, deoarece deosebiri în planificarea aceluiaşi nod apar abia după cunoaşterea condiţiei de bifurcaţie.

Ajustarea unei planificări de trace conform planificării altor trace-uri prelucrate anterior, se face prin rutina *adjust_schedule*. *planificare* este adaptată astfel încât nodurile sale, ce apar planificate în tabela de planificare parţială la momente de timp înaintea cunoaşterii bifurcaţiei, să fie păstrate în ordinea şi la timpii prevăzuţi în tabelă. Restul nodurilor sunt executate în ordinea stabilită de *planificare*, la momente de timp cât mai mici (ASAP). *adjust_schedule* returnează planificarea ajustată a lui *planificare*.

```
Schedule *adjust_schedule (planificare, condiţii cunoscute) {
   noua_planificare = NULL;
   timp cunoscut = 0;
   while (∃ nod ∈ planificare, planificat în tabela de
      planificare conform condiţiilor ∈ condiţii cunoscute) {
      for (∀resursă ∈ arhitectură) {
         coloana curentă = selectează coloana din tabelă, ce
            corespunde condiţiilor ∈ condiţii cunoscute,
            cunoscute pe resursă la timp curent;
         planifică în noduri_planificate toate nod ∈ planificare,
         ce sunt în coloana curentă la timpii la care au fost
         planificaţi în coloană curentă;
         pe baza planificării lui nod, calculează ASAP pentru toţi
            succesorii direcţi ai lui nod;
         marchează resursa lui nod ca şi ocupată pentru intervalul
            de timp în care ea este utilizată de nod;
      }
      timp curent = următorul moment când noi condiţii ∈ condiţii
         cunoscute devin disponibile unei resurse;
   }
   for (toate nod ∈ planificare rămase neplanificate şi în ordinea
      lor din planificare) {
      planificare_nod = maxim (nod.ASAP, timpul când resursa lui
      nod devine disponibilă);
      planifică nod în noua_planificare la timpul din
      planificare_nod;
      pe baza planificării lui nod, calculează ASAP pentru toţi
      succesorii direcţi ai lui nod;
      marchează resursa lui nod ca şi ocupată pentru intervalul de
      timp în care ea este utilizată de nod;
      if (nod este nod condiţional)
         planifică transmiterea condiţiei calculate de el la
         timpul când primul bus din arhitectură devine disponibil;
   }
   return noua_planificare;
// adjust_schedule
```

Ciclul *while* al lui *adjust_schedule* este cel care ajustează nodurile lui *planificare* conform trace-urilor deja prelucrate. El simulează execuţia pe resursele arhitecturii a nodurilor din *planificare*, activate pentru condiţiile din *condiţii cunoscute*. Pentru fiecare resursă a arhitecturii se identifică condiţiile cunoscute la *timp curent* şi

se selectează coloana tabelei, ce le are ca şi etichetă. Nodurile resursei din acea coloană sunt planificate în *noua planificare* la timpii prevăzuţi în tabelă. Când s-au epuizat toate resursele arhitecturii, se calculează următorul moment când se modifică setul condiţiilor cunoscute unei resurse (fie ca urmare a calculării condiţiei sau a recep-tării valorii ei). Nodurile de ajustat se epuizează atunci când valoarea bifurcaţiei devine cunoscută tuturor resurselor, astfel că nu se mai pot selecta coloane noi din tabelă.

Pentru ca *noua_planificare* să fie corectă trebuie ca *adjust_schedule* să asigure ca nici un nod să nu fie planificat înainte ca toţi predecesorii săi să se fi încheiat şi resursa sa să fie disponibilă. Aparent *adjust_schedule* nu face nici o verificare a respectării relaţiilor de precedenţă, dar după cum motivăm în continuare planificarea obţinută prin *adjust_schedule* este corectă. Pentru aceasta arătăm că dacă *planificare* şi nodurile din tabela de planificare satisfac relaţiile de precedenţă ale grafului, atunci şi *noua_planificare* le va respecta. Nodurile lui *noua_planificare* executaţi la timpii stabiliţi în tabela de planificare, respectă relaţiile de precedenţă din felul în care aceasta a fost construită. Restul nodurilor fiind planificate în ordinea lui *planificare*, respectă şi ele automat relaţiile de precedenţă. Prin urmare, ordinea nodurilor în *noua_planificare* este una topologica.

Pentru a respecta în cazul nodurilor lui *planificare* ce nu apar în tabelă cerinţa vis-a-vis de disponibilitatea resurselor, trebuie să calculăm, pe măsură ce sunt parcurse nodurile din tabelă, ASAP-urile succesorilor lor imediaţi şi respectiv să înregistrăm utilizarea resursei ataşate. Pentru fiecare nod a lui *planificare* neconţinut în tabelă, se stabileşte primul moment când resursa sa este disponibilă, iar predecesorii săi au fost executaţi. Acest moment este memorat în variabila *planificare_nod*. Nodul este introdus în *noua_planificare* la timpul *planificare_nod* şi corespunzător sunt actualizate ASAP-urile succesorilor săi imediaţi, respectiv se marchează execuţia sa pe resursa ataşată. Când se planifică noduri condiţionale trebuie prevăzută şi comunicarea condiţiilor calculate. De aceea se identifică magistrală ce devine prima disponibilă la un moment ulterior celui când nodul condiţional îşi termină execuţia. Condiţia este comunicată pe acea magistrală, la cel mai devreme moment posibil.

Vom reveni la sfârşitul paragrafului asupra ordinii în care sunt păstrate nodurile lui *noua_planificare*, astfel încât ea să uşureze cât mai mult scrierea lui *Build_schedule*. În continuare detaliem pseudocodul lui *Build_schedule*.

```
void Build_schedule (condiţii_disponibile, condiţii_netransmise,
e nevoie de ajustare) {
   planificare_maximă = cea mai lungă planificare, dintre cele ale
   trace-urilor parcurse pentru combinaţii de condiţii cuprinzând
   condiţii_disponibile ∪ condiţii_netransmise;
   if (e nevoie de ajustare)
      planificare_maximă = adjust_schedule (planificare_maximă,
      condiţii_disponibile ∪
      condiţii_netransmise);
   nod_curent = primul nod neplanificat ∈ planificare_maximă;
   while (nod_curent != NULL şi nod_curent nu este condiţional) {
      timp_curent = timpul stabilit pentru nod_curent în
      planificare_maximă;
      condiţii_calculate = condiţii ∈ condiţii_netransmise[procesorul
      lui nod_curent] disponibile la timp_curent;
```

```
      if (nu există în tabela de planificare o coloană pentru
      condiții_disponibile ∪ condiții_calculate)
         coloana_c = crează în tabela de planificare o coloană nouă,
         pentru condiții_disponibile ∪ condiții_calculate;
      marchează pe nod_curent în tabela de planificare, în coloană_c,
      la timp_curent;
      if (nod_curent indică comunicarea unei condiții) {
         condiția_curentă = condiția transmisă de activitatea
         indicată de nod_curent;
         valoarea_condiția_curentă = identifică valoarea lui
         condiția_curentă pentru care are loc activarea lui
         planificare_maximă;
         nod_condițional = nodul care calculează condiția_curentă;
         condiții_disponibile = condiții_disponibile ∪
         <condiția_curentă, valoare_condiția_curentă >;
         condiții_netransmise [procesorul lui nod_condițional] =
         condiții_netransmise [procesorul lui nod_condițional] -
         condiția_curentă;
      }
      nod_curent = următorul nod neplanificat ∈ planificare_maximă;
   }
   if (nod_curent == NULL)
      return;
   else // nod_curent este un nod condițional
   {
      timp_curent = timpul prevăzut pentru nod_curent în
      planificare_maximă;
      condiții_calculate = condiții ∈ condiții_netransmise[procesorul
      lui nod_curent] disponibile la timp_curent;
      if (nu există în tabela de planificare o coloană pentru
      condiții_disponibile ∪ condiții_calculate)
         coloana_c = crează în tabela de planificare o coloană nouă,
         pentru condiții_disponibile ∪ condiții_calculate;
      marchează pe nod_curent în tabela de planificare, în coloană_c,
      la timp_curent;
      condiția_curentă = condiția calculată de nod_curent;
      valoarea_condiția_curentă = identifică valoarea lui
      condiția_curentă pentru care are loc activarea lui
      planificare_maximă;
      Build_schedule (condiții_disponibile, condiții_netransmise ∪
         <condiția_curentă, valoarea_condiția_curentă>, 1);
      Build_schedule (condiții_disponibile, condiții_netransmise ∪
         <condiția_curentă, valoarea condiția curentă>, 0);
   }
// Build_schedule
```

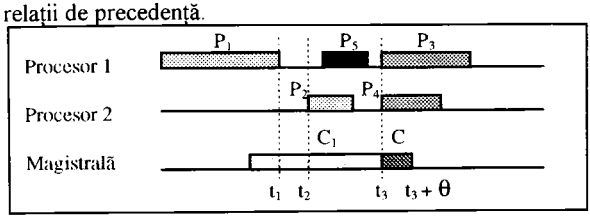
Fiecare apel al lui *Build_schedule* corespunde unei mulțimi de condiții calculate (nodurile lor condiționale au fost deja executate), dar nu neapărat și transmise. Aceste condiții sunt grupate în cei doi parametrii pentru condiții. Condițiile calculate și deja transmise tuturor resurselor din arhitectură aparțin parametrului *condiții_disponibile*, iar restul lui *condiții_netransmise*. *condiții_netransmise* este un tablou, accesat prin identificatorii procesoarelor din arhitectură și fiecare element al său indică condițiile cunoscute procesorului respectiv, dar încă necunoscute restului arhitecturii. Pentru mulțimea condițiilor calculate se identifică toate trace-urile care ar putea fi activate, iar dintre acestea este selectat cel mai lung. Variabila *planificare_maximă* indică acest trace. Deoarece ordinea

nodurilor stabilită de planificarea individuală a lui *planificare_maximă* poate diferi de ordinea indicată prin tabela de planificare, este necesară o ajustare a planificării lui. Acest lucru îl realizează apelul lui *adjust_schedule*, iar planificarea ajustată este memorată tot în variabila *planificare_maximă*.

Build_schedule parcurge nodurile neplanificate ale lui *planificare_maximă* în ordinea lor, până când s-au epuizat toate nodurile sau până când s-a întâlnit un nod condițional. În nodurile condiționale are loc modificarea mulțimii condițiilor calculate (o nouă bifurcație devine cunoscută), ceea ce implică apelurile recursive a lui *Build_schedule* pentru aceste schimbări. *nod_curent* arată nodul parcurs de o anume iterație, iar *timp_curent* momentul de timp când acesta își începe execuția conform lui *planificare_maximă*. Pentru acest moment, algoritmul identifică mulțimea condițiilor cunoscute procesorului unde este executat *nod_curent*. Această mulțime este compusă din *condiții_disponibile* și condițiile calculate pe resursa lui *nod_curent*, dar încă netransmise în arhitectură. Dacă în tabela de planificare nu există coloana pentru această mulțime a condițiilor cunoscute, atunci este creată o coloană nouă. *Build_schedule* marchează planificarea lui *nod_curent* la momentul *timp_curent* în coloana respectivă.

Dacă *nod_curent* indică comunicarea unei condiții, înseamnă că la terminarea sa este modificată mulțimea condițiilor cunoscute în sistem și este schimbată mulțimea condițiilor calculate, dar netransmise. Pentru a modela aceste activități, euristica de planificare parcurge următorii pași. În primul pas se identifică condiția corespunzătoare acestei comunicări, iar *condiția_curentă* reține identitatea ei. Apoi, în *valoarea_condiția_curentă* se memorează valoarea lui *condiția_curentă* pentru care are loc activarea lui *planificare_maximă*. Parametrul *condiții_disponibile* este actualizat, adăugându-i-se perechea <*condiția_curentă*, *valoare_condiția_curentă*>, iar din *condiții_netransmise* este eliminată *condiția_curentă*.

Exemplul următor discută modul în care *condiții_disponibile* și *condiții_netransmise* se modifică pe măsură ce condiții noi sunt calculate și comunicate. Procesul P_1 calculează condiția C, care este transmisă abia la momentul t_3, deși ea a fost disponibilă începând deja cu t_1. Acest lucru se întâmplă din cauză că magistrala este ocupată până la momentul t_3 de comunicarea C_1. După ce condiția C este calculată dar înainte ca ea să fi fost transmisă, pe procesorul 2 este executat începând cu t_2 procesul P_2. La momentul t_3 sunt planificate transmiterea condiției C, P_4 pe procesorul 2, respectiv execuția lui P_3 pe procesorul 1. Procesul P_5, executat în intervalul $[t_1, t_3]$, pe procesorul 1. Presupunem că între toate aceste activități nu apar nici un fel de relații de precedență.



"Fenomenul" ilustrat prin acest exemplu arată că pentru intervalul de timp $[t_1, t_3 + \theta]$ seturile condițiilor cunoscute de procesorul 1 și restul procesoarelor din arhitectură

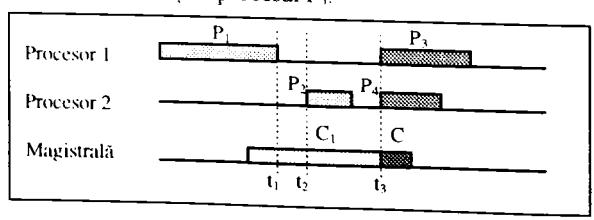
sunt diferite, ceea ce se traduce pentru acest interval de timp, în marcarea proceselor de pe procesorul 1 în coloane diferite ale tabelei de planificare față de restul procesoarelor. Acest lucu este ilustrat de faptul că *condiții_netransmise* pentru procesorul 1 conține pe C. Când s-a încheiat procesul P_1, algoritmul de planificare crează în tabela de planificare coloana pentru *condiții_disponibile* reunită cu C, unde va fi marcat P_5 și P_3. Pentru că resursele lor nu cunosc valoarea lui C, P_2 este planificat la t_2 și P_4 la t_3 în coloane ce nu-l conțin pe C. După ce condiția C a fost transmisă, deci începând cu momentul $t_3 + \theta$, *condiții_disponibile* va cuprinde și condiția C, iar C este eliminată din *condiții_netransmise* pentru procesorul 1 .

Algoritmul selectează următorul nod neplanificat din *planificare_maximă*. Dacă toate nodurile din *planificare_maximă* au fost parcurse, atunci apelul lui *Build_schedule* se încheie.

Dacă însă *nod_curent* corespunde unui nod condițional, ciclul *while* se încheie și se continuă cu următoarele acțiuni. În mod asemănător cu planificarea celorlalte noduri din *planificare_maximă*, *nod_curent* este planificat la *timp_curent*, în coloana tabelei de planificare pentru *condiții_disponibile* reunită cu condițiile netransmise de pe procesorul său la momentul *timp_curent*. În continuare se identifică *condiția_curentă* ca fiind condiția calculată de *nod_curent* și care este noua bifurcație găsită. Apoi, este stabilită valoarea ei pentru care este activat trace-ul lui *planificare_maximă*. Pentru că acesta este cel mai lung dintre cele executate pentru condițiile cunoscute, el va fi primul prelucrat și fără a fi ajustat. Un nod condițional lasă mulțimea *condiții_disponibile* neschimbată, dar modifică pentru procesorul său mulțimea condițiilor cunoscute, dar netransmise. De aceea, *Build_schedule* este apelat recursiv pentru *condiții_disponibile* și *condiții_netransmise* reunită cu perechea *<condiția_curentă, valoare_condiție_curentă>* (care activează *planificare_maximă*), iar apoi pentru *condiții_disponibile* și *condiții_netransmise* reunită cu perechea *<condiția_curentă, valoare_condiție_curentă>* și care activează alternativa complementară.

Anterior am făcut afirmația că ordinea nodurilor din *planificare_maxima* trebuie să fie una topologică (deci un nod să nu apară înaintea unui predecesor al său), iar cele între care nu apar relații de precedență să fie ordonate funcție de momentul lansării lor în execuție. Acest lucru este valabil inclusiv pentru nodurile care reprezintă activitățile de transmitere a condițiilor calculate. Deasemenea, detaliem două restricții de ordonare suplimentare, referitoare la cazul când comunicările condițiilor sunt planificate la același timp cu alte activități de pe resurse diferite, restricții menite să simplifice scrierea algoritmului nostru de planificare. Justificăm acuma toate aceaste restricții.

Considerăm situația de planificare reprezentată în următorul desen. Figura corespunde situației precedente, dar nu conține procesul P_5.



Pentru că procesele apar în planificarea ajustată în ordinea timpilor lor de lansare, pentru exemplul nostru aceasta este P_1, C_1, P_2. `Build_schedule` întâlneşte întâi pe P_1, apoi C_1 şi P_2. Toate cele trei procese sunt marcate în tabela de planificare în aceeaşi coloană, deoarece condiţiile cunoscute procesoarelor lor în momentul planificării lor sunt aceleaşi. Se observă că deşi condiţia C a fost calculată înaintea planificării lui P_2, din cauză că ea nu a fost încă comunicată, setul condiţiilor cunoscute tuturor procesoarelor este nu cuprinde pe C. În mod asemănător, deşi P_4 şi C încep la acelaşi moment de timp, planificarea lui P_4 nu presupune cunoscută valoarea lui C. Planificarea lui corespunde aceluiaşi `condiţii_disponibile` ca şi cea pentru P_1, C_1 şi P_2. Prin urmare, procesarea lui P_4 trebuie să aibă loc înaintea parcurgerii nodului ce indică transmiterea lui C. Acest lucru este echivalent cu faptul că în lista proceselor ordonate a lui `planificare maximă`, P_4 apare înaintea comunicării lui C. Pentru procesul P_3 situaţia este inversată. Pentru că el este mapat aceluiaşi procesor cu P_1, valoarea lui C este cunoscută încă de la t_1. Astfel, planificarea lui corespunde cunoaşterii inclusiv a lui C, motiv pentru care el trebuie parcurs după prelucrarea comunicării lui C. Aceasta este echivalent cu punerea în lista ordonată a lui P_3 dupa nodul corespunzător comunicării lui C.

Exemple de planificare

Această secţiune exemplifică euristica de planificare prin ajustarea şi potrivire pentru două situaţii menite să evidenţieze avantajele şi dezavantajele ei. Prima este una favorabilă, în care graful dependenţelor de date şi de control cuprinde două trace-uri a căror lungime diferă mult între ele. Algoritmul planifică trace-ul cel mai lung, pe care-l lasă nemodificat, după care îl potriveşte pe cel scurt conform lui. Deşi rezultă o alungire a celui scurt, trace-ul cel mai lung rămâne primul. A doua situaţie corespunde unui caz defavorabil, deoarece în graf apar trace-uri de lungime aproximativ egale. Suplimentar, anumite noduri apar devreme în secvenţa de planificare a unui trace şi târziu în cea pentru altul. Planificarea lor conform unui trace, defavorizează pe celălalt şi îl alungeşte corespunzător.

Un caz favorabil

Exemplul pe care-l folosim pentru a ilustra o situaţie în care planificarea prin ajustare şi potrivire găseşte o soluţie bună este cel din figura 5.8. Interesant este că planificând acest caz prin algoritmul de planificare din paragraful 5.6 obţinem o soluţie mai slabă.

În primul pas, algoritmul calculează toate combinaţiile posibile de condiţii care pot apare. Pentru acest exemplu, ele sunt {**C**, **C**}. Apoi, identifică trace-urile corespunzătoare: pentru **C** sunt activate nodurile 1, 2, 3, 4, 5, 8, 9, 10 şi comunicările (5,8) şi (9,6), iar pentru **C** avem nodurile: 1, 2, 9, 6, 7, 8, 10 şi comunicările (2,6) şi (9,6). Planificăm individual cele două trace-uri cu CP/MISF-ML şi constatăm că trace-ul pentru **C** este cel mai lung. Prin urmare, acesta este prelucrat primul. Algoritmul parcurge acest trace începând cu nodul 1. Apoi, la momentul 0 planifică nodul 2. Acesta fiind nod condiţional, în urma execuţiei sale (momentul 3) se modifică mulţimea

condiţiilor calculate, dar netransmise. La timpul 3, pentru că magistrala sistemului este liberă, se planifică transmiterea condiţiei **C**. Astfel, începând cu 3, valoarea condiţiei **C** este cunoscută pe procesorul nodului 2, iar începând cu 4 pe toate procesoarele din arhitectură. Activităţile planificate la timpi următori lui 3 sunt marcate în tabela de planificare în coloana pentru **C**. După ce este prelucrat nodul 2, `Build_schedule` iniţiază două apeluri recursive, primul pentru **C**, iar al doilea pentru **C̲**. Primul apel parcurge în continuare toate nodurile trace-ului şi lasă nemodificaţi timpii prevăzuţi pentru execuţia activităţilor în planificarea individuală. După aceea, algoritmul revine în nodul 2 şi continuă cu prelucrarea trace-ului pentru **C̲**. După ce s-a făcut ajustarea sa în concordanţă cu activităţile deja planificate (nodurile 1, 2 şi comunicarea condiţiei **C**), se marchează restul activităţilor în tabela de planificare, la timpii ajustaţi, în coloana pentru **C̲**. Tabela de planificare ce se obţine la sfârşitul algoritmului este:

	1	C	C̲
0	1		
0	2		
3	condi ţia C		
3		3	
3			9
23			(9,6)
24			(2,6)
25			6
38		(3,4)	
38		9	
39		4	
43			7
63			8
63			10
64		(4,5)	
65		5	
75		(5,8)	
76		8	
76		10	

Un caz defavorabil

Exemplul pe care-l folosim pentru a ilustra o situaţie în care planificarea cu ajustarea trace-urilor găseşte o soluţie slabă este cel din Fig. 5.7. Interesant este că planificând acest caz prin algoritmul de planificare a paragrafului 5.6 obţinem o soluţie mai bună.

	1	G	G̲C̲	GC	G̲	G̲C̲	G̲C̲
0	1						
0	7						
4	8						
6	G						

6		9					
6					2		
8		(9,11)					
8		10					
9		11					
9					C		
9						13	
9							13
9							15
10						(2,3)	
10							(13,14)
11		2					
11						3	
11						(13,14)	
11							14
14		comunicare lui C					
14			15				
15				(2,3)			
16				3			
19						(3,4)	
19						14	
20						4	
24				(3,4)			
25				4			
39			(11,12)				
39				(11,12)			
40				12			
43				13			
47							6
47							16
50						(4,5)	
51						5	
52			12				
53						(5,6)	
55			13				
55			6				
55			(13,14)				
55				(4,5)			
56			14				
56				5			
56						6	
56						16	
58				(5,6)			
60			16				
61				6			

61				(13,14)			
62				14			
66				16			

Acest graf are patru trace-uri, generate pentru combinaţiile {CG, CG, CG, CG}. În acest graf, trace-ul pentru GC este cel mai lung. Prin urmare acesta este primul analizat. Pentru acest trace, nodul 2 are prioritate mică, astfel că nodurile 7, 8, 9 şi 10 sunt planificate înaintea lui. Execuţia lui 2 începe la momentul 11. Următorul trace prelucrat este cel pentru condiţiile GC. Conform lui, nodul 2 ar trebui lansat mai devreme, pentru că el are prioritate mare. Dar 2 a fost deja planificat în urma prelucrării trace-ului anterior, ceea ce face ca din punct de vedere al trace-ului GC, 2 să fi fost întârziat de procese mai puţin prioritare. Acest fapt alungeşte planificarea trace-ului GC, lungimea lui în urma "ajustării" fiind 66, care este cea mai mare dintre trace-urilor tuturor combinaţiilor posibile.

Rezultate experimentale

Experimentarea algoritmului de planificare prin ajustare şi potrivire foloseşte aceeaşi metodologie şi acelaşi set de experimente ca şi experimentarea paragrafului 5.6. Parcurgerea experimentelor şi prelucrarea statistică a rezultatelor a fost făcută automat, folosind un calculator PC486 DX2 la 80 MHz şi având sistemul de operare Linux.

Graficul din figura 5.11 prezintă alungirea relativă procentuală a planificărilor rezultate faţă de lungimea trace-urilor cele mai lungi ale grafurilor. Acest grafic caracterizează calitatea soluţiilor ce se obţin prin planificare.

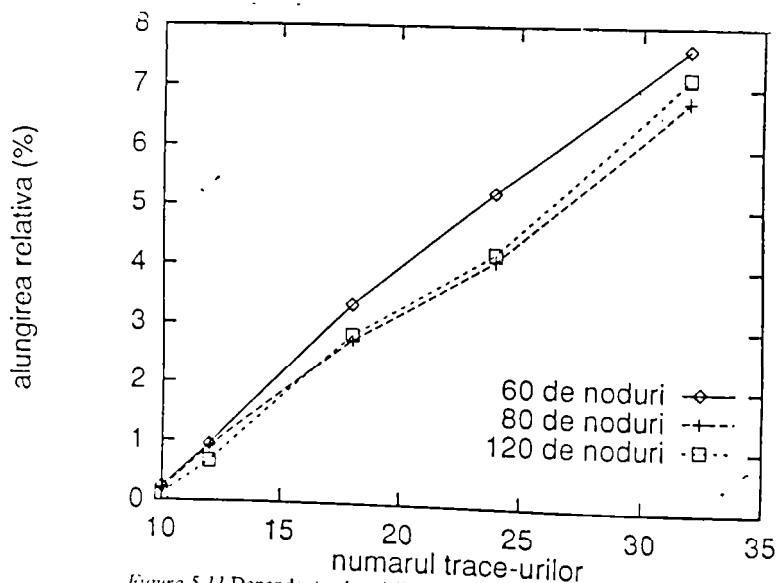


Figura 5.11 Dependenţa alungirii de numărul de trace-uri

Alungirile medii sunt sub 8% şi mai mici de cât cele pentru planificările bazate pe liste de priorităţi. Într-un număr important de cazuri planificările GDC-ului global sunt la fel de lungi ca şi planificarea individuală a trace-ului lor cel mai lung. Tabelul următor indică procentul exemplelor pentru care alungirea rezultată este 0. Liniile corespund numărului de noduri în graf, iar coloanele la numărul de trace-uri.

	10	12	18	24	32
60	89%	81%	56%	43%	32%
80	90%	81%	57%	49.7%	37.6%
120	90%	83%	58.6%	46.5%	29.5%

Alungirile maxime procentuale rezultate în experimente sunt sintetizate în tabelul următor.

	10	12	18	24	32
60	8.33%	42.47%	44.14%	44.14%	69.49%
80	10.12%	39.49%	32.56%	59.63%	84.48%
120	5.15%	18.96%	29.52%	36.76%	45.54%

Figura 5.12 prezintă timpii medii de execuţie necesari doar pentru algoritmul de ajustare şi potrivire, după ce priorităţile nodurilor au fost calculate.

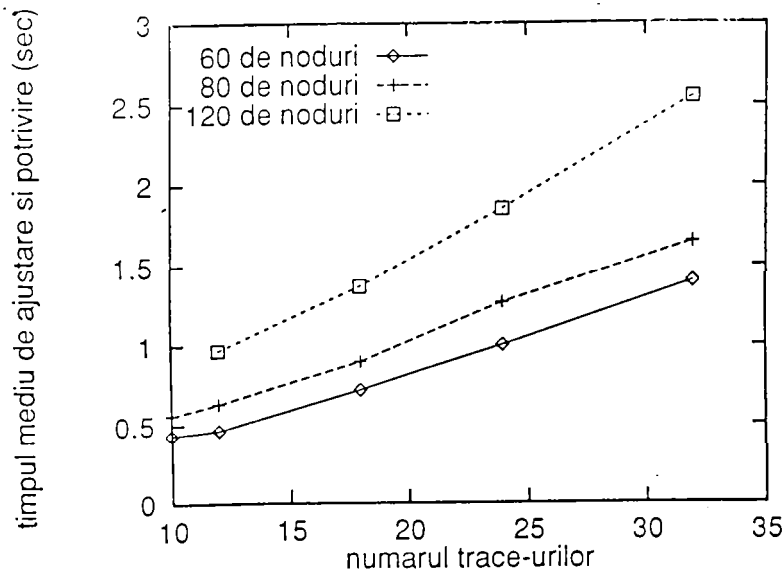


Figura 5.12 Dependenţa timpului mediu de ajustare şi potrivire de numărul de trace-uri

Figura 5.13 arată variaţia timpilor medii ai algoritmului de planificare prin ajustare şi potrivire (cuprinzând calculul priorităţilor, ajustarea şi potrivirea planificărilor) funcţie de numărul de trace-uri.

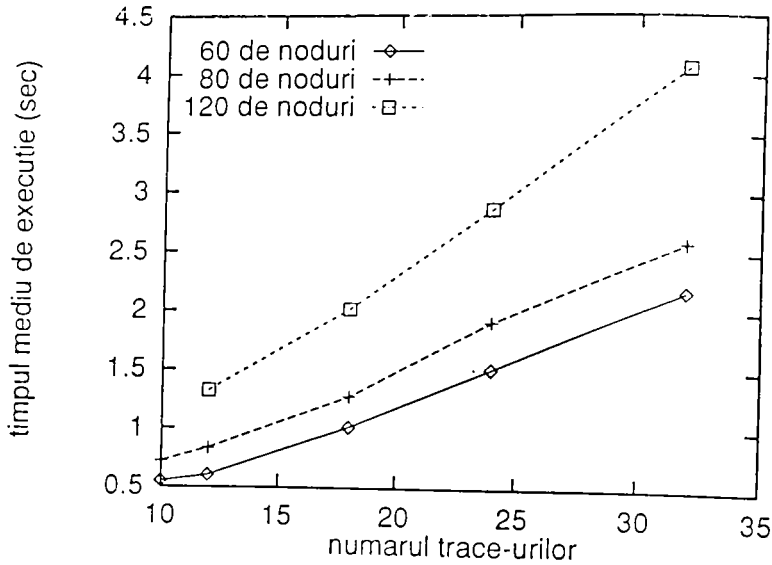


Figura 5.13 Dependența timpului total de planificare de numărul de trace-uri

5.8 Concluzii cu privire la planificarea euristică a GDC-urilor legate

Capitolul 5 discută problema planificării grafurilor legate, cu dependențe de date și de control. O constrângere practică este aceea că acești algoritmi trebuie să fie rapizi, pentru a putea rezolva într-un timp rezonabil grafuri cu mai multe zeci de noduri, grafuri ce corespund exemplelor din realitate. Euristicile de planificare bazate pe liste sunt algoritmi rapizi și care găsesc soluții apropiate optimelor. Calitatea soluțiilor lor depinde esențial de prioritățile folosite și pe baza rezultatelor experimentale ale capitolului 3 am decis să folosim CP/MISF-ML.

Capitolul detaliază arhitectura folosită pentru execuția grafurilor, deoarece există decizii de planificare ce sunt dependente de ea. Arhitectura cuprinde un număr arbitrar de procesoare programabile de diferite tipuri, pentru execuția componentei software, un procesor dedicat - ASIC sau FPGA, pentru funcționalitatea din hardware și circuite de memorie pentru păstrarea structurilor mari de date. Procesele își transferă date folosind un număr de magistrale, la care au acces toate componentele arhitecturii. Această structură particulară a arhitecturii influențează modul în care sunt transmise valorile condițiilor în arhitectură, restul deciziilor de planificare fiind însă neinfluențate, din cauză legării activităților de resurse.

Tabela de planificare este distribuită resurselor și folosită de planificatoare locale. Accesarea tabelei se face printr-un registru al momentului curent, ce selectează linia curentă și un registru al setului curent al condițiilor, care identifică coloana. Condiția de determinism logic a planificării impune ca între coloanele tabelei să nu apară situații de

conflict, adică noduri care să fie executate de mai multe ori pentru aceeaşi parcurgere a grafului. Pentru coloanele aflate în conflict am demonstrat că între expresiile logice ce le etichetează trebuie să existe o relaţie de incluziune.

Influenţa condiţiilor din graf asupra planificării unui nod se manifestă în două moduri. Condiţiile listei de gărzi a unui nod hotărăsc execuţia sau nu a nodului, în timp ce condiţiile din lista de influenţă nu condiţionează execuţia sa, dar influenţează momentul planificării. Condiţiile listei de influenţă sunt responsabile de apariţia conflictelor între coloane, deoarece întotdeauna în momentul planificării unui nod trebuie cunoscută garda sa. Capitolul prezintă algoritmii ce calculează static, înaintea planificării propriu-zise, listele de gărzi şi de influenţă ale nodurilor, parcurgând GDC-ul în adâncime.

Planificarea unui GDC impune ca orice condiţie să fie cunoscută la orice moment când pe o resursă ea este folosită în planificare. Consecinţele sunt că un algoritm de planificare a GDC trebuie să asigure că o condiţie este transmisă cel puţin către resursele arhitecturii unde ea este folosită în planificare şi pentru a întârzia cât mai puţin activităţile dependente de ea, cel târziu la primul moment în care ea este folosită în planificarea unui trace. Ultima restricţie asigură că tabela de planificare ce se construieşte este implicit fără conflicte. Transmiterea unei condiţii poate fi neselectivă, spre toate resursele arhitecturii sau selectiv, doar spre procesoarele ce o folosesc, adică cele ce cuprind procese având condiţia în lista lor de gărzi sau de influenţă. Momentul transmiterii unei condiţii poate fi cât mai devreme, adică pe prima magistrală găsită liberă la timpi ulteriori momentului când condiţia este calculată, sau cât mai târziu posibil, însă fără a întârzia nodurile ce o folosesc în planificarea lor. Prima metodă este simplă şi nu tinde să întârzie planificarea, dar din cauza importanţei maxime pe care o primesc automat comunicările de condiţii, ele pot întârzia alte comunicări prioritare. Totodată, este dificil de stabilit pentru o condiţie, dacă ea trebuie într-adevăr transmisă, astfel că sunt comunicate şi condiţii folosite doar pe procesorul ce le calculează. A doua strategie de transmitere a condiţiilor are tendinţa de a întârzia unele din activităţile ce o folosesc, dar în schimb prezintă două avantaje: se poate stabili dacă transmisia este necesară şi se încurajează transmiterea în acelaşi pachet de date a mai multor condiţii, calculate de aceeaşi resursă.

Fiecare GDC cuprinde setul trace-urilor executate pentru toate combinaţiile posibile de valori ale condiţiilor din graf. Un nod poate aparţine mai multor trace-uri şi să aibă importanţe diferite în cadrul planificării lor. De aceea, nodurile unui GDC trebuie caracterizate mai degrabă prin o mulţime de priorităţi, de cât prin o prioritate unică. În momentul când sunt planificate unele noduri, nu se poate distinge trace-ul curent, din cauză că unele din condiţiile grafului sunt încă necalculate. Aceste noduri vor fi planificate conform situaţiilor celor mai dezavantajoase, ce pot apare pentru setul de condiţii cunoscute în acel moment. Orice decizie de a planifica nodurile unui trace conform priorităţilor din alt trace (acest lucru ţine de natura relaţiilor de control) are efectul de a întârzia noduri prioritare şi potenţial, planificarea se poate alungi. Este bine să păstrăm nemodificate cu precădere trace-urile lungi şi să le potrivim pe cele scurte.

Capitolul propune doi algoritmi euristici pentru planificarea GDC-urilor legate. Algoritmul bazat pe liste de priorităţi fixează fiecărui nod o prioritate CP/MISF-ML unică, corespunzătoare situaţiei celei mai defavorabile în care el apare. Priorităţile sunt calculate folosind un graf simplificat, ce cuprinde doar dependenţele de date. Condiţiile

sunt transmise la cel mai târziu moment posibil, atât pentru a uşura transmiterea lor selectivă spre resurse, cât şi pentru facilitarea comunicării cumulate a mai multor condiţii în acelaşi pachet de date. Algoritmul este uşor de formulat şi de implementat, dar mai ales rapid. Rezultatele experimentale au folosit un număr foarte mare de grafuri de diferite feluri. Ele arată că planificările medii sunt puţin alungite faţă de trace-urile componente, mai ales pentru grafuri cu puţine structuri condiţionale încuibate. Cazurile defavorabile apar când nodurile unui trace sunt planificate conform priorităţilor unor trace-uri ce se exlud reciproc. Algoritmul poate fi folosit şi pentru grafuri cu construcţii ciclice, pentru că le planifică fără a le derula explicit.

Algoritmul bazat pe ajustarea şi potrivirea trace-urilor planifică nodurile unui trace fie după priorităţile sale sau după cele ale unor trace-uri mai prioritare. Trace-urile mai lungi sunt considerate mai prioritare. În ideea de a păstra trace-urile lungi neschimbate în cea mai mare măsură, ele sunt prelucrate primele. Trace-urile scurte sunt adaptate celor lungi, păstrându-se nemodificate nodurile planificate la momente anterioare cunoaşterii bifurcaţiilor pe resursele arhitecturii. Prin ajustare se evită apariţia conflictelor între coloanele tabelei de planificare. Complexitatea în timp a algoritmului este mai mare de cât cea pentru algoritmul bazat pe liste de priorităţi. În medie el găseşte planificări calitativ superioare, din cauza "capacităţii" sale de a folosi priorităţile potrivit trace-ului curent. Situaţiile dezavantajoase apar pentru noduri ce au atât priorităţi mici cât şi mari, pentru că orice alegere conform uneia din situaţii o dezavantajează pe cealaltă.

Noi privim acest studiu ca un pas de plecare în dezvoltarea unor euristici superioare pentru planificarea GDC-urilor legate. În acest moment considerăm că unele din aspectele problemei, ce pot fi studiate în continuare, sunt:

- Experimentarea pentru noduri a altor priorităţi, de cât CP/MISF-ML.
- Găsirea altor strategii de comunicare a condiţiilor, eventual ataşându-le priorităţi asemeni celorlalte activităţi din graf.
- Ierarhizarea trace-urilor după alte criterii de cât lungimea. Foarte interesantă pare folosirea unui factor, care să reflecte influenţa asupra trace-ului curent a planificării trace-urilor mai importante.
- Tratarea ciclurilor

Capitolul 6

Sinteza de nivel înalt a componentei hardware şi generarea codului pentru procesele software

Rezumat

Acest capitol descrie etapele "târzii" ale co-sintezei hardware/software: sinteza de nivel înalt a proceselor componentei hardware şi generarea codului C pentru procesele din software. Cele două etape sunt legate de faptul că folosesc reţelele ETPN ca şi reprezentare intermediară. Reţelele ETPN corespunzătoare specificărilor VHDL sunt obţinute aplicând un set de reguli ce păstrează semantica iniţială. În vederea sintezei, comunicările interprocese ale descrierilor VHDL pot fi organizate conform cu două metode, SAW şi SR. CAMAD este sistemul de sinteză la nivel înalt, integrat în mediul nostru de co-sinteză. Rezultatul produs de el este implementarea la nivel RTL a proceselor reprezentate ca reţele ETPN. Generatorul de cod C parcurge reprezentarea intermediară şi generează cod C pentru stările ce pot fi atinse (*reachable*) şi care sunt distincte. Fiecare stare produce secvenţa de instrucţiuni C echivalentă cu funcţionalitatea reprezentată de stare. Codul C generat poate fi subiectul optimizărilor de cod independente de arhitectură, menite să-i crească viteza de execuţie şi să-i scadă memoria necesară.

Soluţia şi rezultatele noastre legate de generarea codului C pentru componenta software au fost prezentate şi în [DE96].

6.1 Reprezentarea intermediară folosită pentru sinteza de nivel înalt şi generarea componetei software

ETPN (*Extended Timed Petri Nets*) [PE87] este *notaţia formală* folosită ca şi reprezentare intermediară pentru sinteza de nivel înalt a componentei hardware şi generarea codului pentru partea de software. ETPN descrie în mod natural acele aspecte ale sistemelor hardware, considerate ca dificil de reprezentat: execuţia concurentă şi sincronizarea operaţiilor. ETPN are şi caracteristicile unei notaţii pentru reprezentarea proceselor executate în software şi uşurează translatarea lor în forma finală, care este cod C. Cele două elemente justifică folosirea ETPN ca şi notaţie intermediară *unică* pentru descrierea şi manipularea proceselor unei specificări VHDL.

ETPN este derivat din teoria reţelelor Petri [JE88] şi particularitatea lui este că părţile de date şi de control sunt reprezentate separat, dar interacţionează între ele. *Partea de date* (*data path*) este un graf orientat compus din noduri şi arce. Nodurile descriu unităţile pentru manipularea datelor (unităţi de memorare, operatori aritmetici, etc.), iar arcele indică relaţiile ce există între noduri. *Partea de control* (*control part*) este o reţea Petri temporală, ce transmite semnale de control spre partea de date şi care primeşte semnale condiţionale de la partea de date. Separarea datelor de control este importantă pentru procesul de sinteză la nivel înalt. Descrierile comportamentale redau explicit maniera de prelucrare a datelor, iar partea de control este generată din maniera de secvenţiere a instrucţiunilor precum şi din instrucţiunile de control folosite în specificare. Sinteza de nivel înalt apare ca un proces de generare de informaţie de control nouă. Separarea datelor de control ajută şi implementarea structurilor sintetizate în circuite VLSI.

În continuare descriem semantica ETPN folosind exemplul VHDL din figura 6.1.

```
entity exemplu is
end exemplu;

architecture a of exemplu is
begin
   P1: process
      variable k, b: bit := '1';
      variable flag: boolean := true;
      begin
         if flag then          -- (1)
            k := b;            -- (2)
            b := '0';          -- (3)
            flag := NOT flag;  -- (4)
         end if;
         .....
      end process;
end a;
```

Figura 6.1 Exemplu de specificare VHDL

Reţeaua ETPN din figura 6.2 corespunde specificării VHDL din figura 6.1. Nodurile părţii de date sunt reprezentate prin dreptunghiuri, etichetate cu numele variabilelor corespunzătoare în specificarea VHDL sau cu tipul operaţiilor executate. Nodurile flag, b şi k sunt generate pentru a păstra cele trei variabile, nodul NOT realizează operatorul unar de negare, iar nodurile c şi c̲ memorează valorile condiţiei c, folosită pentru controlarea fluxului în partea de control a reţelei. Nodurile sunt legate prin arce orientate, ce realizează fluxul de date descris prin specificare. Fiecare arc este controlat de un *loc de control* (*control place*) al părţii de control, astfel că atunci când locul respectiv este marcat, arcul ataşat este deschis, iar informaţia nodului sursă "curge" în nodul ţintă. De exemplu, locul P2 marcat deschide arcul între nodurile b şi k, astfel că valoarea memorată în b este atribuită şi nodului k. Locurile de control sunt conectate prin *tranziţii* (*transitions*). În figura 6.2a, locurile sunt simbolizate prin cerculeţe, iar tranziţiile prin liniuţe orizontale. Funcţionarea reţelei ETPN foloseşte *marcaje*, care se deplasează prin partea de control funcţie de conexiunile între locuri şi tranziţiile care sunt *autorizate*. După cum deja am afirmat, un loc marcat deschide arcele ataşate lui în partea de date şi determină un flux de date. Partea de date controlează fluxul controlului prin *condiţii* care gardează tranziţiile. Un marcaj poate trece de la un loc spre altul, dacă garda tranziţiei este adevărată. O astfel de tranziţie este *autorizată*. În

figura 6.2a, locul P1 şi cele două tranziţii ale sale de ieşire realizează în reteaua ETPN semantica instrucţiunii *if* (instrucţiunea 1) din codul VHDL. Funcţie de valoarea lui flag, condiţiile c sau c̲ sunt adevărate, ceea ce autorizează una din cele două tranziţii de ieşire ale lui P1. Marcajul din P1 trece pentru c adevărat în locurile P2 şi apoi P3, iar pentru c̲ adevărat direct în P5. O tranziţie negardată este automat autorizată atunci când toate locurile sale de intrare cuprind un marcaj şi efectul ei este că toate locurile sale de ieşire primesc câte un marcaj. Legătura inversă spre P1 indică faptul că execuţia reţelei ETPN este un ciclu infinit, ceea ce corespunde semanticii unui proces VHDL.

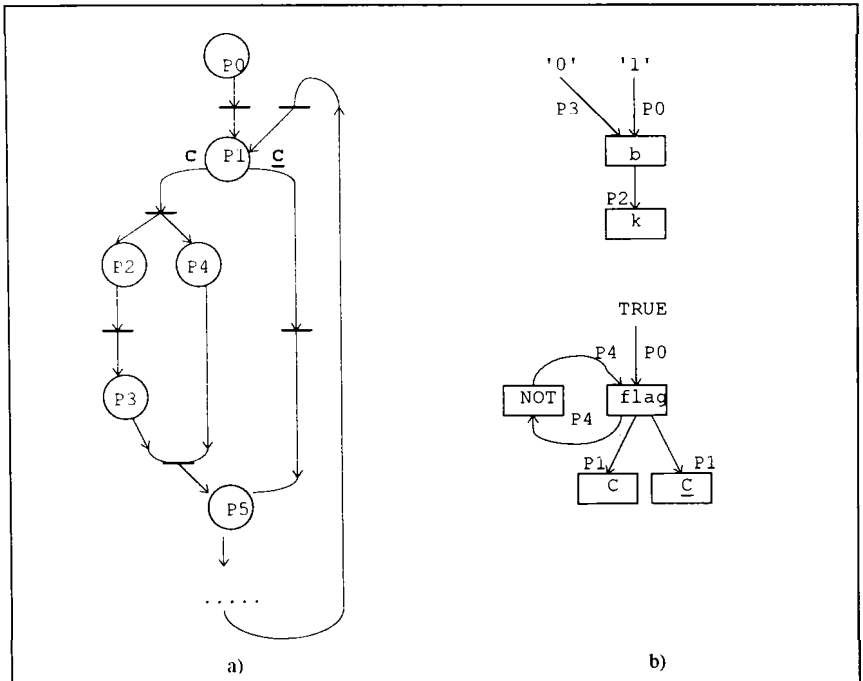


Figura 6.2 Reţeaua ETPN corespunzătoare codului VHDL din figura 6.1

Specificările ETPN sunt compilate automat în reţele ETPN corespunzătoare folosind compilatorul VHDL-ETPN al mediului nostru de proiectare [MI93].

Calitatea esenţială a reţelelor ETPN este că descriu explicit caracteristicile specificării, ceea ce permite luarea unor decizii de proiectare eficiente atât de componenta pentru sinteza de nivel înalt, cât şi de generatorul de cod C. De exemplu, ETPN indică explicit acele secvenţe de operaţii între care nu există relaţii de dependenţă şi care pot fi executate concurent. Locurile ataşate acestor operaţii pot cuprinde simultan marcaje. Între operaţiile lui P2-P3 şi P4 nu sunt relaţii de dependenţă, astfel că ele pot fi executate concurent. Acest lucru se datorează faptului că *iniţial*, între instrucţiunile VHDL (2)-(3) şi (4) nu apare nici o relaţie de dependenţă.

6.2 Generarea ETPN pe baza specificărilor VHDL

Aspectele VHDL cele mai delicate, referitoare la construirea reţelelor ETPN, sunt legate de tratarea comunicărilor interprocese. Descrierea interacţiunilor între procesele VHDL poate fi realizată prin una din următoarele două metode [EK97]: prima foloseşte instrucţiunile de *signal assignment* şi *wait* (metoda este denumită *SAW*), iar a doua apelează la primitivele sincrone *send/receive* pentru transmiterea mesajelor (metoda *SR*). Regulile de generare a reţelei ETPN pornind de la o specificare VHDL au fost descrise amănunţit în [EK97][EK94][EK92]. Exemplul din figura 6.3 şi figura 6.4 ilustrează caracteristicile de bază ale metodei SAW, iar exemplul din figura 6.5 şi reţeaua ETPN din figura 6.6 introduc elementele definitorii pentru SR.

SAW respectă fidel mecanismul standard de comunicare interprocese [IE88], conform căreia procesele interacţionează prin semnale şi actualizarea lor este operată abia după ce toate procesele specificării sunt în starea de *wait*. Consecinţa acestui fapt este că mecanismul de sincronizare specific ciclului de simulare VHDL trebuie sintetizat în hardware. Hardware-ul produs poate fi controlat fie de un singur sau mai multe automate cu stări finite. Figura 6.4 conţine ETPN-ul cu un singur automat finit produs pentru codul VHDL din figura 6.3.

```
signal s: integer;
....
P1: process
....
begin
      wait on s;
      ....
end process;
....
-- alte procese
```

Figura 6.3 Exemplu VHDL folosit pentru descrierea metodei SAW

Procesul P este supervizorul care asigură prin locul Q, actualizarea sincronizată a tuturor semnalelor din specificare. Locul Q devine marcat doar atunci când Q' conţine un marcaj şi garda C este adevărată. În figura 6.4b se observă că C este rezultatul operaţiei logice *şi* între valorile tuturor variabilelor x_i, ce controlează sincronizarea proceselor din specificare. Câte un x_i este generat pentru orice proces P_i din specificare şi el are lungimea de un bit. El este setat pe valoarea 1, atunci când locul W_{pi} este marcat, ceea ce se întâmplă doar când procesul ataşat este în starea de *wait*. x_i este resetat de locul P0 la începutul execuţiei sau la marcarea locurilor W'_{pi}, când toate procesele s-au sincronizat, Q din P este marcat şi s-a generat condiţia C_s adevărată. Figura 6.4c ilustrează partea de date creată pentru semnalul *s*. Ea cuprinde două noduri cu rol de memorare, *s* pentru păstrarea valorii curente a semnalului şi *s'* pentru ultima valoarea asignată lui *s*. Condiţia C_s arată apariţia unui *eveniment* pentru semnalul *s*, adică schimbarea valorii curente a semnalului. Generarea lui C_s şi actualizarea semnalului *s* este controlată de locul Q din P_0. Lucrul acesta este conform cu semantica ciclului VHDL de simulare, care obligă ca actualizarea semnalelor să aibă loc doar după ce toate procesele specificării ajung în starea de *wait*.

Părțile de control ale proceselor sunt disjuncte şi pentru fiecare este sintetizat un automat cu număr finit de stări distinct [PK94]. Sincronizarea colecției de automate se face prin automatul lui P_0 şi variabilele $x_1, x_2, \ldots$. Această metodă produce o mulțime de automate de complexitate redusă în locul unuia singur, dar de complexitate mare.

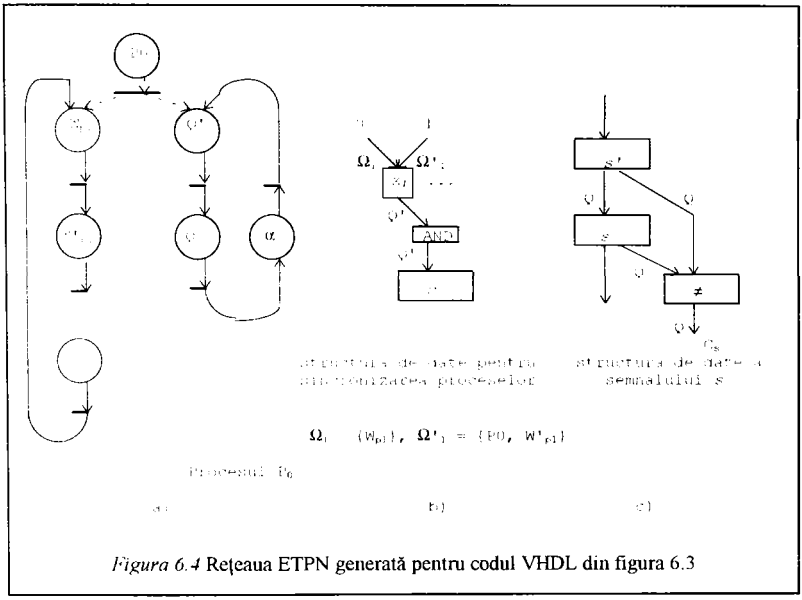


Figura 6.4 Rețeaua ETPN generată pentru codul VHDL din figura 6.3

Sincronizarea strânsă a proceselor sintetizate este dezavantajul major al metodei SAW, pentru că aceasta determină un grad scăzut de concurență al implementării. Metoda SR urmărește să reducă volumul sincronizărilor interprocese din implementare, păstrând în acelaşi timp semantica ei echivalentă cu cea a modelului de simulare. Aceste obiective sunt obținute impunând restricții asupra modului de descriere a specificărilor VHDL. Principiul metodei SR este să se înlocuiască sincronizarea *implicită* a proceselor prin ciclul de simulare, cu sincronizarea *explicită* obținută prin operațiile asupra semnalelor.

```
...
signal a: integer;
...
P1: process                          P2: process
...                                  ...
begin                                begin
      ...                                  ...
      send(a, x);                          receive(a);
      ...                                  ...
end process;                         end process;
...
```

Figura 6.5 Exemplu VHDL folosit pentru descrierea metodei SR

Procesele VHDL comunică prin mesaje, folosind primitivele *send/receive*. Oricâte procese pot fi conectate la un semnal, dar unul singur execută *send* asupra lui. Sinteza acestui mecanism de comunicare nu implică de cât procesele participante la comunicarea printr-un anume semnal, spre deosebire de SAW care adresează toate procesele specificării. Rezultatul practic este că hardware-ul sintetizat prin metoda SR

are un grad sporit de paralelism şi asincronism faţă de cel obţinut prin SAW. Figura 6.6 ilustrează ETPN-ul produs conform metodei SR pentru codul VHDL din figura 6.5.

Figura 6.6c arată că metoda SR reprezintă semnalele asemeni variabilelor, printr-un singur nod de date. Semantica primitivelor *send/receive* elimină asignarea unei valori noi unui semnal în doi paşi, astfel că un semnal îşi modifică automat valoarea în urma execuţiei lui *send*. Locurile $P1_1$, $P1_2$ şi $P1_3$ din procesul *P1* corespund apelului VHDL *send*(a,x), iar $P2_1$ şi $P2_2$ din *P2* lui *receive*(a) (figura 6.6a).

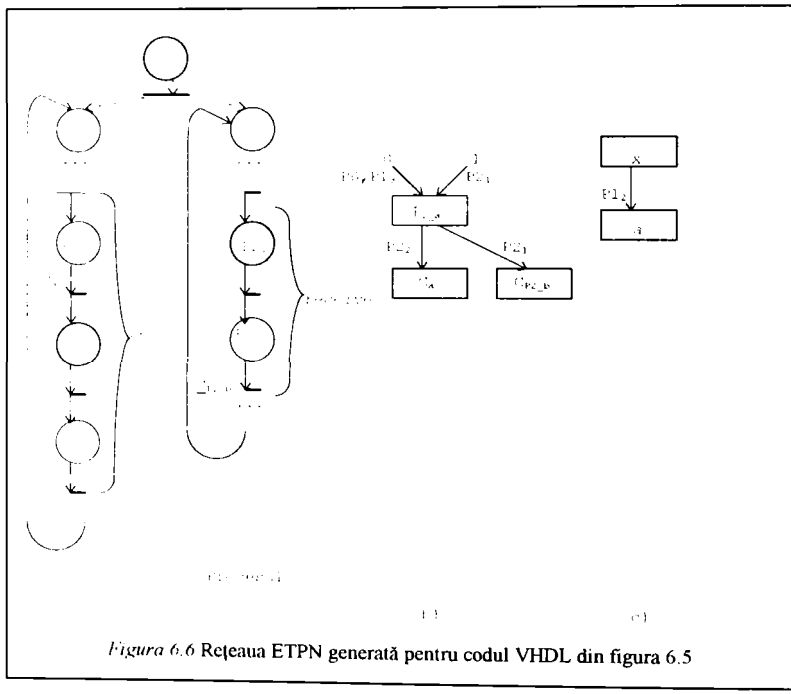


Figura 6.6 Reţeaua ETPN generată pentru codul VHDL din figura 6.5

Dacă locul $P1_1$ conţine un marcaj, tranziţia sa de ieşire este activată în momentul în care condiţia C_1 devine adevărată. Valoarea lui C_1 este stabilită de nodul P_{2_a}, care este setat pe 1 de către locul $P2_1$ marcat (figura 6.6b). $P2_1$ este marcat atunci când se execută primitiva *receive*(a) a procesului *P2*, după care marcajul se deplasează în $P2_2$, unde rămâne atâta timp cât condiţia C_{P2_b} este falsă. Autorizarea tranziţiei de ieşire a lui $P1_1$ determină curgerea marcajului în $P1_2$, care controlează asignarea valorii lui x lui a (figura 6.6c). Locul $P1_2$ resetează valoarea lui P_{2_a}, ceea ce modifică condiţia C_{P2_b} la adevărat şi autorizarea tranziţiei de ieşire a lui $P2_2$.

Părţile de control ale reţelelor ETPN generate prin metoda SR sunt implementate ca şi automate cu număr finit de stări independente, slab cuplate (*loosely coupled*) şi care lucrează în paralel. Metoda SR este mult mai simplă şi mai natural de folosit pentru specificări de sistem de cât SAW, iar rezultatele sintezei sunt superioare atât ca şi cost (număr mai mic de noduri, din cauza reprezentării simplificate a semnalelor), dar şi ca viteză de execuţie (grad sporit de concurenţă pentru implementarea finală).

6.3 Sistemul CAMAD de sinteză la nivel înalt

CAMAD [PK94] este sistemul de sinteză la nivel înalt pe care noi îl considerăm disponibil pentru sinteza proceselor atribuite componentei hardware. CAMAD este integrat cu restul instrumentelor de proiectare din mediul nostru de co-sinteză hardware/software, astfel că rezultatele produse de el sunt convertite automat în formate acceptate de celălalte instrumente şi invers.

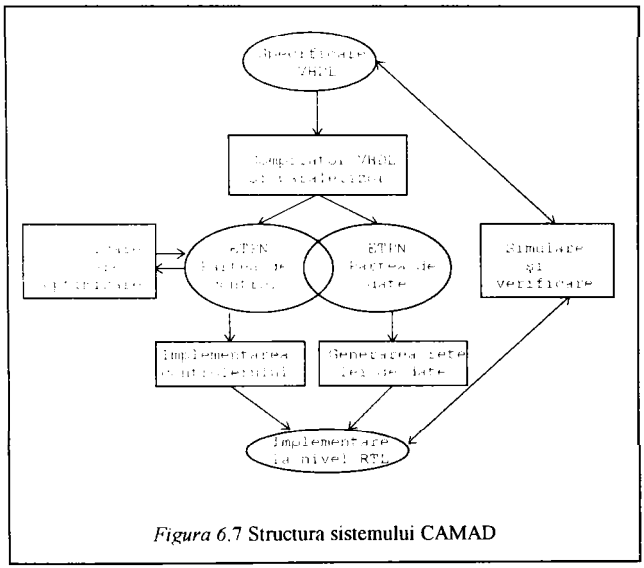


Figura 6.7 Structura sistemului CAMAD

CAMAD foloseşte ca informaţie de intrare reţeaua ETPN echivalentă unei specificări VHDL, care apoi este transformată printr-un proces iterativ, menit să îmbunătăţească calitatea implementării (*iterative improvement approach*). Reţeaua ETPN obţinută în urma compilării descrierii VHDL, este în acelaşi timp şi soluţia primitivă de implementare a sistemului. Acesteia îi sunt aplicate un set de transformări ce păstrează semantica şi corectitudinea reţelei (*correctness preserving transformations*), dar care cresc performanţele implementării. Paşii de transformare aplicaţi formează *procesul de sinteză*. CAMAD realizează planificarea operaţiilor (*operation scheduling*), alocarea părţii de date şi de control (*data path allocation* and *control allocation*) şi într-o oarecare măsură, legarea modulelor (*module binding*). Integrarea paşilor de proiectare este obţinută printr-un set de transformări de bază asupra reprezentării intermediare şi care tratează atât planificări parţiale, cât şi alocări parţiale ale părţilor de date şi de control. Un algoritm de optimizare analizează soluţia globală şi selectează transformările operate la fiecare pas de procesul iterativ de sinteză.

Figura 6.7 prezintă structura sistemului CAMAD. Specificările VHDL sunt compilate în reţele ETPN, după care paralelismul lor este îmbunătăţit automat. Prin paralelizare structurile ETPN iniţiale sunt transformate în structuri echivalente semantic, dar care să evidenţieze *toate* operaţiile concurente ce apar. După transformarea amintită, este

produsă implementarea componentei hardware la nivel RTL (*register transfer level*) şi care va cuprinde o reţea pentru date (*data path netlist*) şi un automat cu număr finit de stări (*finite state machine*) pentru controler. Implementarea finală RTL poate fi convertită automat într-o descriere VHDL structurală, care asemeni descrierii comportamentale iniţiale, poate fi simulată în vederea verificării corectitudinii ei [PE92].

6.4 Generarea codului pentru componenta software

Codul C al proceselor componentei software este produs pe baza reprezentării intermediare a specificării. Reprezentarea este formată din *stări*, fiecare stare fiind compusă din mulţimea locurilor de control ale reţelei ETPN care pot fi executate în acelaşi pas şi *mulţimea tranziţiilor* deschise la trecerea dintr-o stare precedentă în cea curentă. O stare este înlănţuită cu stările *următoare în ordinea execuţiei*, adică acele stări ce sunt executate imediat în continuarea celei curente. Deasemenea stările reprezentării sunt înlănţuite şi într-o *ordine liniară*, funcţie de secvenţa în care ele au fost create în etapa de generare a reprezentării intermediare. Fiecare proces al specificării are ataşată o listă distinctă de stări înlănţuite, a cărei prim element este indicat de pointerul *root*. Figura 6.8 ilustrează lista stărilor unui proces şi cele două moduri de înlănţuire a lor.

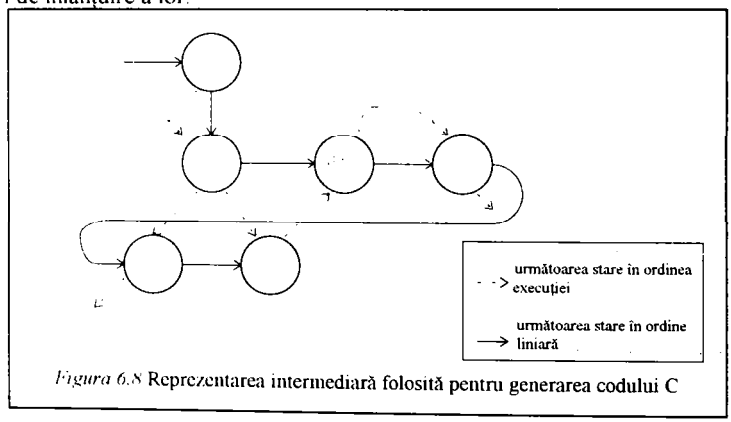


Figura 6.8 Reprezentarea intermediară folosită pentru generarea codului C

Este posibil ca un grup de stări să cuprindă aceeaşi mulţime de locuri de control şi după cum se va vedea în continuare, codul C corespunzător lor să fie identic. Secvenţa de cod C produsă pentru mulţimea comună a locurilor de control deserveşte toate stările ce cuprind această mulţime.

Generatorul de cod C pentru procesele software are pseudocodul detaliat în figura 6.9.

Stările procesului sunt parcurse în ordinea lor liniară, începând cu nodul indicat de `root`. Locurile de control ale lui `stare curentă` generează instrucţiunile C ale stării. Dacă `stare curentă` nu este urmată de o stare următoare în ordinea execuţiei, atunci ea este executată ultima şi suplimentar este generată instrucţiunea `goto stop`, ce încheie execuţia codului. Dacă `stare curentă` are o singură stare următoare în ordinea execuţiei, ce diferă de starea următoare în ordine liniară, atunci este produsă instrucţiunea `goto` la eticheta stării următoare în ordinea execuţiei. Astfel, la execuţia codului C generat execuţia lui `stare curentă` este urmată de cea a stării următoare.

Dacă *stare curentă* are mai multe stări următoare în ordinea execuției, atunci sunt produse instrucțiuni *if* pentru fiecare din stările următoare. Expresiile testate de instrucțiunile *if* sunt condițiile care gardează tranzițiile ce conectează *stare curentă* de cea următoare. Fiecare *if* cuprinde o instrucțiune `goto` la eticheta stării următoare corespunzătoare. La execuția codului C, funcție de condiția adevărată, este aleasă secvența de cod atașată. *stare curentă* produce cod C nou, dacă în pașii de generare anteriori nu au fost procesate stări, cărora să le corespundă aceiași mulțime de locuri de control ca și lui.

```
void C_code_generator (root) {
        point = root;
        do
           stare curentă = starea indicată de point;
           if (nu s-a generat cod C pentru stare curentă) {
              generează o etichetă unică pentru stare curentă;
              for (∀ s ∈ locurile de control ale lui stare curentă)
                 generează instrucțiunile C pentru s;
              case numărul stărilor următoare lui stare curentă în
              ordinea execuției este {
              0: generează goto stop;
              1: if (starea următoare lui stare curentă în ordinea
              execuției != starea următoare a lui stare curentă
              în ordine liniară)
                 generează goto eticheta stării următoare lui stare
                 curentă în ordinea execuției;
              mai multe stări următoare lui stare curentă în ordinea
              execuției:
                 for (∀ next point ∈ starea următoare lui stare
                 curentă în ordinea execuției) {
                    generează if;
                    generează condiția care gardează tranziția ce
                    conectează pe stare curentă la next point;
                    generează goto eticheta lui next point;
                 }
              }
           }
           point = pointerul la următoarea stare a lui stare curentă
           în ordine liniară;
        } while (point != NULL);
        generează eticheta stop;
     }
```

Figura 6.9 Pseudocodul generatorului de cod C

Fiecare loc de control al unei stări produce o anume instrucțiune C. Locurile ce reprezintă instrucțiuni de atribuire produc instrucțiuni C echivalente. Părțile stânga și dreapta ale instrucțiunii sunt identificate izolând subgraful părții de date controlat de locul curent. Locurilor pentru primitivele VHDL *send/receive* le sunt generate secvențe de cod C ce realizează funcționalitatea pentru transmiterea și recepția mesajelor.

Nodurile părții de date de tip întreg sunt translatate în variabile de tipul *int*. Asemănător, nodurile caracter sunt transformate în variabile *char*, iar tablourile ETPN generează tablouri C cu aceiași dimensiune. Lucrurile devin interesante pentru structurile de date ETPN, ce nu au un corespondent direct în C. ETPN cuprinde variabile de tip *bit* și *bitstring*, în timp ce limbajul C nu are cele două tipuri printre tipurile sale predefinite și nici nu oferă operatori pentru operanzii de aceste tipuri.

Generatorul nostru transformă o variabilă de tip bitstring, având lungimea de k biţi, într-un tablou de caractere cu $\lceil k/8 \rceil$ elemente, iar biţii sunt păstraţi începând cu elementul cel mai din stânga al tabloului. Operaţiile pentru bitstringuri sunt grupate într-o bibliotecă C specială, pe care am denumit-o *routinelib.lib*. Operaţiile bibliotecii atribuie o valoare unei porţiuni (*slice*) dintr-un bitstring, citesc o porţiune a unui bitstring, tratează constantele de tip bitstring şi implemetează operatori unari (negaţie) şi binari (şi, sau, sau-exclsiv) pentru operanzi de tip bitstring.

Generatorul de cod a fost implementat în C şi experimentat inclusiv pentru exemple complexe, descrise de specificări VHDL cu lungimea de ordinul sutelor de linii de cod.

Exemplu legat de generarea de cod C pentru o reţea ETPN

Acest exemplu ilustrează structura codului C produs de generatorul nostru pentru o reţea ETPN. Codul VHDL în cauză este descris în figura 6.1, reţeaua ETPN produsă pentru el este ilustrată în figura 6.2, iar codul C este cel din figura 6.10.

```
#include "routinelib.h"
char N5, N8, N9, N10, N11;
char N1[1], N3[1];
char _temp_0[1], _temp_1[1], _temp_2[1], _temp_3[1];

void P1() {
S1:
   put_bistring_const (_temp_0, "1");
   assign_bitstring (N1, 1, 18, _temp_0);
   put_bitstring_const (_temp_1, "1");
   assign_bitstring (N3, 1, 18, _temp_1, 1);
   N5 = 1;
S2:
   N8 = 1;
   N9 = !1;
   if (N8) goto S3;
   if (N9) goto S4;
S3:
   N10 = N5;
   N11 = !N5;
   if (N10) goto S5;
   if (N11) goto S6;
S4:
   goto stop;
S5:
   get_right_part (N3, 1, 1, 8, _temp_2);
   assign_bitstring (N1, 1, 1, 8, _temp_2, 8);
   goto S7;
S6:
   goto S2;
S7:
   put_bitstring_const (_temp_3, "0");
   assign_bitstring (N3, 1, 1, 8, _temp_3, 1);
   goto S8;
S8:
   N5 = !N5;
   goto S6
stop:
   ;
```

Figura 6.10 Codul C generat pentru reţeaua ETPN din figura 6.2

Codul C generat începe cu comanda preprocesor pentru includerea fişierului header `routinelib.h`. Acesta conţine declaraţiile funcţiilor menite lucrului cu bitstring-uri (funcţiile *`put_bitstring_const`*, *`assign_bitstring`* şi *`get_right_part`* din exemplul nostru). `N1` corespunde la nodul `k`, iar `N3` la `b`. Atât `k` cât şi `b` sunt noduri de tip bitstring şi sunt mapate la variabile de tip tablou de caractere cu un element ($\lceil 1/8 \rceil$). `N5` este generat pentru nodul `flag`, iar `N9`, `N10` şi `N11` sunt variabile auxiliare folosite în calcularea condiţiilor pentru instrucţiunea *if*. `S1` etichetează instrucţiunile ce iniţializează pe `N1` şi `N3` cu valoarea '1'. *`put_bitstring_const`* copiază valoarea celui de-al doilea argument în primul. *`assign_bitstring`* atribuie valoarea celui de-al cincilea argument primului. Al doilea şi al treilea parametru fixează porţiunea (*slice*) bitstring-ului atribuit. Codul etichetat cu `S2` verifică dacă execuţia procesului continuă sau nu. Reţeaua ETPN este executată într-un ciclu infinit şi de aceea condiţia testată (`N8`) este întotdeauna adevărată. Secvenţa etichetei `S3` corespunde *instrucţiunii 1* din codul VHDL.Condiţia testată pentru *if* este păstrată de variabila `N10`, iar negata ei în `N11`. Instrucţinile *if* generate realizează cele două variante de transfer al controlului. Codul C ce începe la eticheta `S5` corespunde *instrucţiunii 2* din codul VHDL. *`get_right_part`* ia o porţiune din `N3` şi o atribuie variabilei auxiliare `_temp_2`, care apoi este atribuită lui `N1`. Atribuirea de la eticheta `S8` inversează valoarea lui `N5` (`flag`) şi corespunde *instrucţiunii 4* a codului VHDL. Codul etichetei `S7` are acelaşi efect ca şi *instrucţiunea 3*. După ce este parcurs codul C al procesului, instrucţiunea `goto S2` al etichetei `S6`, declanşează o nouă iteraţie a procesului `P1`.

Generarea de cod optimizat

Optimizarea de cod îmbunătăţeşte atât timpul de execuţie, cît şi memoria ocupată de cod. Optimizările de cod pot avea loc la două nivele [OH94][GM94]. *Optimizările de cod independente de arhitectura gazdă* folosesc reprezentarea intermediară a codului şi cuprind: eliminarea codului mort (*dead code*), propagarea constantelor, evaluarea expresiilor, împachetarea variabilelor de tip bistring, expandarea apelurilor de funcţii, etc. *Optimizările de cod dependente de arhitectura gazdă* sunt specifice procesorului folosit şi cuprind alocarea regiştrilor pentru variabilele codului, planificarea şi selecţia instrucţiunilor, etc. Optimizările dependente de arhitectură sunt realizate de compilatoarele C uzuale, de aceea, discuţia următoare este concentrată asupra optimizărilor independente de arhitectură.

Lungimile nodurilor de tip bitstring nu sunt de regulă multiplii întregi ai lui 8, astfel că lungimea variabilelor C generate este mai mare de cât cea necesară. Împachetând aceste variabile în tablouri continue, se reduc spaţiile de memorie rămase nefolosite, iar volumul de memorie necesar pentru reprezentarea variabilelor scade. Acest procedeu este realizat la nivelul generatorului de cod şi nu implică regie de sistem suplimentară pentru execuţia codului. Prin propagarea constantelor (*constant folding*) şi evaluarea expresiilor în etapa generării de cod, programele rezultate sunt mai scurte şi mai rapide. O serie de variabile auxiliare sunt reduse şi sunt eliminate secvenţele de cod care tratează constantele şi calculează expresiile. O reţea ETPN poate cuprinde secvenţe ce nu sunt executate sub nici o circumstanţă. Aceste secvenţe corespund de regulă la alternative a căror condiţie de selecţie este întotdeauna falsă sau la cicluri care sunt executate de 0 ori. Secvenţele sunt denumite cod mort (*dead code*) şi nu produc

cod C corespunzător. Mecanismul *call/return* pentru apelul subrutinelor este costisitor atât din punct de vedere al memoriei folosite, cât şi al timpului lui de execuţie. Evităm folosirea mecanismului în situaţii în care el nu se impune expandând (înlocuind) apelurile de funcţii cu corpul lor. Funcţiile apelate de un număr redus de ori sunt candidatele pentru expandare. Codul C generat anterior (figura 6.10) arată maniera relativ ineficientă de tratare a condiţiilor ce gardează tranziţiile ETPN. Fiecare gardă produce o variabilă C nouă şi o instrucţiune ce îi atribuie o valoare. Este mult mai economic ca în locul acestei secvenţe să generăm expresia condiţiei ca făcând parte din instrucţiunea *if*. Deasemenea fiecărei condiţii îi este produsă şi secvenţa de cod ce calculează valoarea ei negată. Această strategie este foarte simplă şi uşurează implementarea generatorului de cod, dar uneori produce cod nefolosit. Optimizarea de "bun simţ" este să analizăm utilitatea fiecărei condiţii (inclusiv a negatelor lor) şi să generăm cod doar pentru cele ce sunt folosite în instrucţiuni *if*.

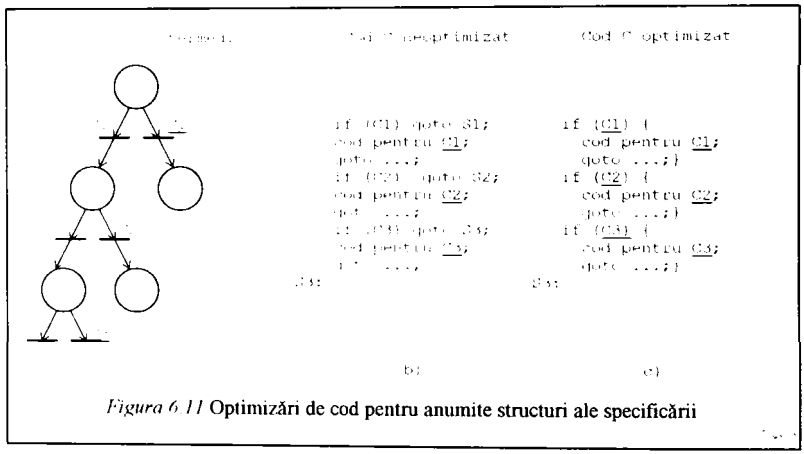


Figura 6.11 Optimizări de cod pentru anumite structuri ale specificării

ETPN nu conţine nici un fel de informaţie legată de structura specificării VHDL, astfel că nici codul C produs nu cuprinde această informaţie. Acest lucru nu este un dezavantaj major, din moment ce codul produs nu trebuie să fie lizibil. Singurul dezavantaj provine din faptul că se pierde noţiunea de domeniu al variabilelor, iar consecinţa practică este că toate variabilele unui program generat sunt globale şi trebuie păstrate pe toată durata lui de execuţie. Domeniul unei variabile se poate stabili prin analiza statică a reţelei ETPN, iar informaţia astfel găsită poate fi cuprinsă în codul C produs. În acest mod, variabilele sunt alocate în stivă la începutul domeniului lor şi dezalocate atunci când acesta se termină.

Alte optimizări vizează construcţii ce au o structură (*pattern*) particulară. De exemplu, în urma compilării unei porţiuni VHDL rezultă secvenţa ETPN din figura 6.11a. Codul C neoptimizat este redat în figura 6.11b, iar cel optimizat în figura 6.11c.

Optimizări foarte interesante se pot opera la nivelul proceselor ce comunică. Foarte des, specificările VHDL cuprind perechi de procese aflate într-o relaţie *master-slave* (figura 6.12). Procesul master transmite date procesului slave, după care aşteaptă ca el să-i returneze rezultatele procesării sale. Dacă cele două procese software sunt mapate

aceluiaşi procesor, ele pot fi contopite într-un proces unic (figura 6.12). Acest fapt creşte calitatea codului generat (viteză, memorie ocupată), pentru că se elimină două comunicări interprocese, care de obicei sunt mari consumatoare de regie de sistem. În general putem face afirmaţia că dependenţele de date între procesele mapate aceluiaşi procesor pot fi folosite pentru remodelarea structurii proceselor şi eliminarea unor comunicări interprocese redundante.

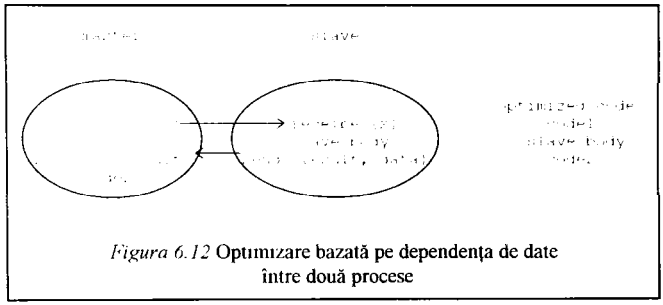


Figura 6.12 Optimizare bazată pe dependenţa de date între două procese

6.5 Concluzii

Capitolul 6 descrie etapele "târzii" ale co-sintezei hardware/software, adică sinteza de nivel înalt a proceselor componentei hardware şi generarea codului C pentru procesele din software. Cele două etape sunt înrudite prin faptul că folosesc reţelele ETPN ca şi reprezentare intermediară. ETPN are particularitatea că partea sa de date şi de control sunt reprezentate ca şi părţi separate, dar aflate în interacţiune, ceea ce ajută atât procesul de sinteză, cât şi pe cel de generare de cod C. ETPN descrie în mod natural aspecte ale specificării de sistem considerate dificile, cum sunt de exemplu execuţia concurentă şi sincronizarea operaţiilor.

Reţelele ETPN sunt construite pentru specificări VHDL, aplicând un set de reguli ce păstrează semantica iniţială. Comunicările interprocese pot fi descrise în VHDL (în vederea sintezei) prin două metode, SAW şi SR. SAW respectă fidel mecanismul standard de comunicare, conform căruia procesele comunică prin instrucţiunile *wait* şi *signal assignment*. Consecinţa acestui fapt este că ciclul de simulare VHDL trebuie implementat în hardware. Metoda SR descrie comunicarea proceselor VHDL prin mesaje şi cu folosirea primitivelor *send/receive*. Acest lucru permite reducerea volumului de sincronizări din implementare, păstrând echivalenţa ei semantică cu modelul de simulare. Hardware-ul sintetizat este mai ieftin (partea de date este mai simplă) şi mai rapid (implementarea are un grad sporit de concurenţă). CAMAD este sistemul de sinteză la nivel înalt, integrat în mediul nostru de co-sinteză. Rezultatul produs de el este implementarea proceselor hardware la nivel RTL.

Generatorul de cod C parcurge reprezentarea intermediară şi generează cod pentru stările ce pot fi atinse (*reachable*) şi care sunt distincte. Fiecare stare produce secvenţa de instrucţiuni C, care este echivalentă cu funcţionalitatea reprezentată prin stare. Nodurile de tip *bit* şi *bitstring* sunt translatate în tablouri de caractere cu lungime acoperitoare, iar operaţiile asupra bitstring-urilor sunt cuprinse într-o bibliotecă

dedicată. Codul C generat poate fi subiectul optimizărilor de cod independente de arhitectură, care sunt menite să-i crească viteza de execuţie şi să-i scadă memoria necesară. Optimizările realizează împachetarea variabilelor de tip bitstring, evaluarea expresiilor, eliminarea codului mort, expandarea apelurilor de funcţii, procesarea eficientă a expresiilor condiţionale şi restructurarea codului. A doua categorie de optimizări foloseşte dependenţele de date între procesele mapate aceluiaşi procesor, pentru contopirea anumitor procese şi eliminarea comunicărilor lor interprocese.

Capitolul 7

Co-sinteza hardware/software a blocului F4 din circuitul ATM

Rezumat

Acest capitol prezintă experienţa dobândită prin experimentarea metodologiei şi a uneltelor noastre software de co-sinteză hardware/software, pentru blocul F4 al circuitului ATM. Specificul lui F4 este că are o funcţionalitate foarte complexă şi complicată, motiv pentru care paşii co-sintezei sale sunt orientaţi în special pe rafinări şi reorganizări ale specificării, pe măsură ce informaţii noi devin disponibile. În capitol sunt sintetizate principiile de bază ale protocolului de comunicare ATM, structura şi funcţionarea circuitelor ATM şi sunt detaliate funcţionalitatea blocului F4 vis-a-vis de gestiunea erorilor şi monitorizarea funcţionării circuitelor ATM. Prima specificare dezvoltată pentru F4 descrie *doar* funcţionalitatea lui şi nu conţine *nici* un fel de constrângeri de proiectare sau detalii de implementare. Ea este organizată încât să uşureze activităţile de modelare, simulare şi verificare. Specificarea este rafinată şi reorganizată în vederea paşilor următori ai co-sintezei, astfel încât ea să uşureze şi să eficientizeze deciziile de proiectare Procesele sunt restructurate pentru ca ele să descrie o funcţie unică, iar detaliile de arhitectură adăugate au un caracter *generic*, în sensul că se referă la folosirea bufferelor şi a memoriilor. După ce sunt fixate resursele disponibile pentru sinteza arhitecturii, sunt estimaţi timpii de execuţie ai activităţilor din specificarea rafinată. Deoarece nu am dispus de nici o referinţă privind corectitudinea acestor valori, am estimat timpii prin două metode ortogonale: executând codul şi măsurând timpii şi calculând-i pe baza manualelor de procesor. Timpii operaţiilor de lucru cu memoria au fost estimaţi ţinându-se cont de volumul de informaţii implicat şi de maniera de organizare a memoriei. Dispunând de o imagine a timpilor necesari diverselor activităţi, precum şi de constrângerile de timp impuse prin standard, specificarea este reorganizată pentru a doua oară, astfel încât să putem căuta implementarea ce îndeplineşte constrângerile de timp la un cost cât mai mic. Reorganizarea se face prin comasarea proceselor identice în exemplare unice şi respectiv, multiplicarea unor procese în mai multe exemplare pentru asigurarea vitezei de lucru necesare. Specificarea transformată este folosită pentru generarea grafurilor de date şi de control folosite la căutarea implementării celei mai eficiente. Co-sinteza hardware/software continuă cu aplicarea *iterativă* a paşilor de alocare a resurselor, legarea activităţilor pe resurse, planificarea lor şi folosirea informaţiilor rezultate în vederea îmbunătăţirii soluţiei. Când o soluţie nu satisface cerinţele de timp ale standardului se încearcă o arhitectură mai performantă şi pornind de la o soluţie care îndeplineşte constrângerile de timp, se poate încerca o arhitectură mai ieftină.

Experimentele noastre au studiat patru variante de arhitecturi posibile, ce se disting prin performanțe și cost. Arhitecturile performante permit desfășurarea în paralel a activităților concurente și aceasta poate scade timpul total al grafurilor. Se poate alege una din cele patru variante de arhitecturi funcție de performanțele de timp pe care le dorim pentru implementarea finală.

7.1 Principiile de bază ale ATM

Caracteristicile de bază ale protocolului ATM sunt prezentate detaliat în literatura de specialitate [RC94][SV94][LL93][PR91]. Ele se referă în special la următoarele:

- ATM folosește celule de lungime fixă (53 de octeți). O celulă este formată din header (5 octeți) și câmpurile de informații (48 de octeți). Header-ul conține informații explicite pentru rutare, astfel că celule aparținând la canale diferite pot fi transmise succesiv într-o ordine arbitrară pe aceeași legătură fizică. Principala diferență între ATM și STM (*Synchronous Transfer Mode*) este existența header-ului pentru rutarea celulelor.
- ATM este orientat pe conexiune. Conexiunile ATM sunt inițializate înainte ca informația utilizator să fie transmisă și rămân nemodificate pe întreaga durată a transmisiei. Resursele alocate unei conexiuni sunt eliberate când comunicarea pe ea se încheie.
- Câmpurile de informație dintr-o celulă ATM sunt transmise în mod transparent în rețea, în sensul că ele nu sunt procesate (de exemplu, pentru ele nu se verifică apariția erorilor de transmisie).

Figura 7.1 ilustrează schema bloc a unui circuit ATM (*ATM switch*). Circuitul are *n*

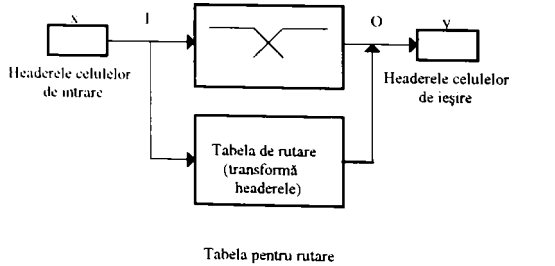


Tabela pentru rutare

Conexiunea de intrare	Header	Conexiunea de ieșire	Header
I	x	O	y

Figura 7.1 Principiul de rutare al protocolului ATM

intrări (I_1, I_2, ... I_n), care aduc celulele spre switch. Acolo, ele sunt rutate pe baza informației cuprinse în header-ele lor, iar apoi transmise către ieșirile (O_1, O_2, ..., O_q).

Header-ul unei celule de intrare şi conexiunea pe care vine ea sunt folosite pentru generarea unui header nou, respectiv pentru identificarea conexiunii de ieşire.

Conexiunile necesare între utilizatori sunt stabilite înainte de a transmite datele în reţea. Conexiunile pot fi de două tipuri: *Virtual Path Connections* (VPC) şi *Virtual Channel Connections* (VCC). VPC-urile sunt identificate folosind *Virtual Path Identifier* (VPI), iar VCC-urile sunt adresate prin *Virtual Channel Identifier* (VCI).

VCC este mijlocul principal de conectare între două puncte. Figurile 7.2 şi 7.3 arată modul în care VCC-urile între aceleaşi două puncte sunt grupate într-un VPC. VPC-urile sunt folosite pentru a creşte numărul de VCC-uri, care pot fi rutate prin nodurile unei reţele ATM.

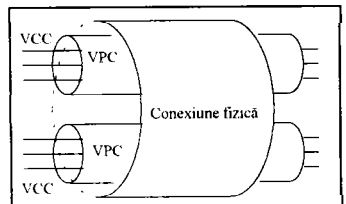


Figura7.2 Relaţia între VCC, VPC şi conexiunea fizică

[PR91] detaliază următoarele aspecte ce compun fincţionalitatea protocolului ATM:

- Translatarea (convertirea) VPI/VCI.
- Adăugarea la celulă a informaţiei de rutare.
- Eliminarea celulelor cu VPI/VCI eronat.
- Tratarea conexiunilor (*Connection Handling*).
- Gestiunea variaţiilor în fluxul asincron de celule.
- Gestiunea erorilor (*Error Handling*).
- Monitorizarea performanţelor (*Performance Monitoring*).
- Managementul securităţii (*Security Management*).
- Charging Management.
- Gestiunea configuraţiilor (*Configuration Management*).
- Funcţii de sistem (*System Functions*).

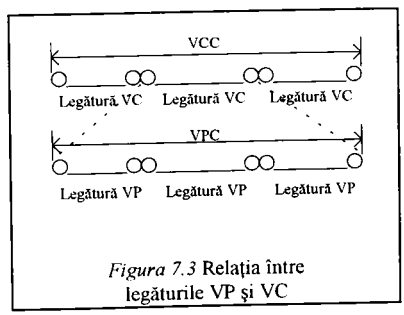


Figura 7.3 Relaţia între legăturile VP şi VC

7.2 Blocul F4 al switchului ATM

Blocul F4 acoperă funcționalitatea OAM (*Operation and Maintenance*) pentru nivelul F4. Nivelul F4 cuprinde funcționalitatea OAM pentru VP-uri, iar F5 pe cea pentru VC-uri. Nivelele F1, F2 și F3 tratează funcționalitatea OAM pentru nivelul fizic (*Physical Layer*) al protocolului ATM.

Conform standardului [BE93], blocul F4 are următoarea funcționalitate de bază:

- *Gestiunea erorilor*: Dacă nivelul fizic semnalizează către blocul F4 apariția unei erori, atunci acesta generează celule speciale, denumite *celule OAM*. Acestea sunt destinate conexiunii pe care a apărut eroarea. Dacă eroarea persistă, atunci sistemul pentru management (*Management System*) este informat despre eroarea apărută.
- *Monitorizarea performanțelor*: Funcționarea normală a rețelei este monitorizată prin verificarea continuă sau periodică a fluxului de celule transmise.
- *Localizarea erorilor*: În continuare, la apariția unei erori este necesară, localizarea ei. Localizarea este realizată prin celule OAM speciale, denumite *celule loop back* (*loop back cells*)
- *Activare dezactivare*: Activarea și dezactivarea funcțiilor OAM pentru care blocurile F4 sunt implicate în mod activ (de exemplu, monitorizarea performanețelor) se realizează printr-un protocol special.

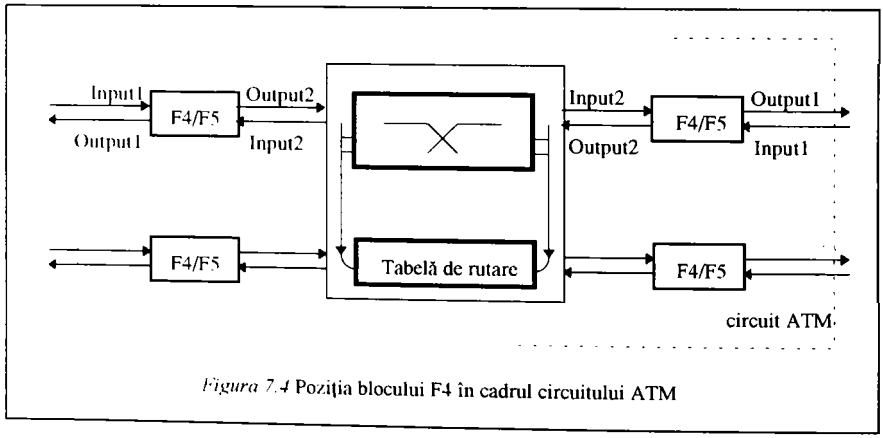


Figura 7.4 Poziția blocului F4 în cadrul circuitului ATM

Un bloc F4 este prezent pentru fiecare conexiune fizică a unui circuit ATM (figura 7.4). Pentru că este bidirecțional, blocul F4 are două intrări și două ieșiri.

F4 își realizează funcționalitatea procesând celule ATM speciale. Aceste celule sunt denumite celule OAM (*Operation and Maintenance*) și se deosebesc de *celulele utilizator* prin valorile atribuite lui VCI (nivelul F4) și respectiv PTI (nivelul F5).

Standardul ATM [BE93] definește trei tipuri de celule OAM:

- *Celulele pentru activare dezactivare* (*Activation Deactivation cells*) activează și dezactivează generarea celulelor OAM de către blocul F4 și respectiv funcțiile sale

de procesare a VPC şi VCC-urilor. Celulele pentru activare/dezactivare sunt create ca urmare a unei cereri venite din partea sistemului pentru management.

- *Celulele pentru monitorizarea performanţelor* (*Performance Monitoring cells*). Acestea sunt transmise în mod regulat între punctele terminale ale unei conexiuni sau unui segment de conexiune (figura 7.3). Celulele PM poartă informaţii referitoare la celulele utilizator transmise pe o anumită conexiune VP sau VC. Informaţia purtată de celulele PM este folosită la calcularea a două mărimi:
  1) *Diferenţa* între numărul celulelor utilizator trimise şi numărul celor recepţionate.
  2) *Numărul nepotrivirilor* între codurile de detecţie ale erorilor (*block error detection codes*) calculate local şi cele salvate în celulă.
  Mărimile calculate pe baza celulelor PM sunt folosite pentru:
  1) Verificarea calităţii transmisei şi analiza măsurii în care funcţionarea ei pentru VP sau VC-uri corespunde celei dorite.
  2) Diagnoza şi analiza problemelor legate de VC-uri şi VP-uri.
- *Celulele pentru managementul erorilor* (*Fault Management cells*). Aceste celule indică o situaţie de eroare, cum ar fi apariţia discontinuităţilor pentru VP-uri sau VC-uri. Totodată, celule din această categorie testează continuitatea VP-urilor sau VC-urilor, pentru a identifica locul unde s-a produs un defect. Nodul care a sesizat o eroare, trimite către succesorii săi celule OAM, menite să le raporteze apariţia erorii. Sistemul pentru management este cel care iniţiază procedura pentru verificarea continuităţii unei anume conexiuni.

7.3 Modelarea blocului F4

7.3.1 Dezvoltarea modelului de simulare pentru funcţionalitatea blocului F4

Figura 7.5 prezintă modelul construit de noi pentru blocul F4. Dezvoltarea modelului cu acestă structură este bazată pe următoarele argumente:

- Blocul F4 produce celule de ieşire ca urmare a *doi* factori. *Primul factor* se referă la receptarea unui "stimul" de intrare, care poate fi atât o celulă ATM, cât şi un semnal de la nivelul fizic sau sistemul pentru management. *Al doilea caz* apare atunci când blocul se găseşte pentru o conexiune, într-o stare pentru care trebuie produse periodic celule de ieşire. De exemplu, dacă conexiunea este în starea AIS, atunci la intervale regulate de timp sunt generate celule speciale AIS. Generarea celulelor PM, a celor pentru activare/dezactivare şi a semnalelor de răspuns către sistemul pentru management au loc ca urmare a unor informaţii de intrare. Trecerea unei conexiuni dintr-o stare în alta (de exemplu, trecerea din starea normală în starea AIS) are loc tot pentru anumite informaţii de intrare. Pe de-altă parte, celulele FM sunt produse periodic, ca urmare stării în care este o conexiune a nodului.

  Ca atare, ni s-a părut normal ca modelul pentru F4 să cuprindă două procese distincte. Un proces acoperă funcţionalitatea blocului legată de generarea celulelor şi a semnalelor de ieşire ca un răspuns la "stimuli" (celule şi semnale) de intrare. Al doilea proces realizează funcţionalitatea reprezentând producerea periodică a celulelor specifice stărilor curente în care sunt conexiunile ce traversează blocul F4.

În figura 7.5, procesul *InputHandle* tratează intrările blocului, iar *FMCellGenerator* crează celule de ieşire periodice, ataşate stărilor.

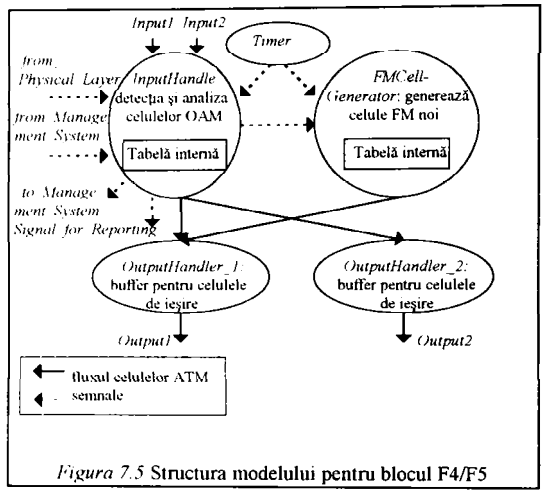


Figura 7.5 Structura modelului pentru blocul F4/F5

- Modelul cuprinde două procese (*buffere*) pentru stocarea celulelor de ieşire. În figura 7.5 aceste procese sunt denumite *OutputHandler_1* şi *OutputHandler_2*, câte unul pentru fiecare ieşire a modelului. Suplimentar, *OutputHandler_2* prelucrează priorităţile celulelor de ieşire. Standardul ATM [BE93] defineşte trei priorităţi pentru celulele ATM.
  1) Celulele FM şi cele pentru activare/dezactivarea au prioritatea cea mai mare.
  2) Celulele utilizator au prioritate medie.
  3) Celulele PM au cea mai mică prioritate.

  În mod obişnuit, celulele PM sunt intârziate atâta timp cât există în buffer şi celule de alt tip şi care trebuie transmise. Totuşi, această întârziere este păstrată sub o limită specificată în standard, forţându-se (când e necesar) celulele PM la ieşire.

  Blocul F4 gestionează fluxuri de celule, care corespund la o mulţime de VPC-uri şi VCC-uri, ce traversează blocul. Fiecare din aceste căi se poate găsi într-o stare specifică, pentru care activităţile executate să fie caracteristice. Informaţia legată de stările în care se găsesc VPC şi VCC-urile blocului este păstrată în tabelele de memorie, locale proceselor *InputHandle* şi *FMCellGenerator* (figura 7.5).

- Interfaţa circuitului F4 cu exteriorul cuprinde canalele pentru celulele de intrare şi de ieşire, precum şi semnalele cu nivelul fizic şi sistemul pentru management. În modelul nostru (figura 7.5), următoarele porturi realizează această interfaţă:
  - *Input1* este intrarea pentru celulele ce vin din exterior spre blocul F4, iar *Input2* primeşte celulele ce vin dinspre circuitul ATM spre blocul F4 (figura 7.4).
  - *Output1* conectează F4 cu exteriorul, iar *Output2* trimite celulele dinspre bloc spre circuitul ATM (figura 7.4).
  - Portul *from Physical Layer* recepţionează semnale la momentele când pe un anumit VP a apărut sau a fost eliminată o eroare. *from_Management_System*

este folosit de sistemul pentru management pentru a cere servicii de monitorizare a performanțelor sau verificarea continuității pentru o anumită conexiune sau de informare a nodului ATM despre stabilirea unei conexiuni noi.

- Rezultatele culese în urma monitorizării performanțelor sunt raportate prin portul *Signal_for_Reporting*, iar prin *to_Management_System*, F4 comunică sistemului pentru management rezultatele serviciilor inițiate prin *from_Management_System*.

Dacă procesul *InputHandle* primește de la nivelul fizic un semnal de eroare, atunci VPC sau VCC-ul implicat intră în starea de eroare. Deasemenea, procesul semnalizează *FMCellGenerator* că trebuie să genereze celule specifice stării de eroare (celule AIS sau RDI). Astfel de celule sunt generate până la momentul când eroarea descoperită la nivelul fizic este eliminată. La rezolvarea erorii fizice, *InputHandle* recepționează de la nivelul fizic (prin portul *from_Physical_Layer*) un semnal specific, părăsește starea de eroare fixată pentru VPC-ul sau VCC-ul implicat și încearcă să revină în starea existentă înainte de apariția erorii. Activitățile parcurse de blocul F4 la eliminarea unei erori fizice cuprind și transmiterea unui semnal către *FMCellGenerator*, în urma căruia acesta încetează să mai producă celule specifice stării de eroare.

Procesul *Timer* generează semnale pentru *InputHandle* și *FMCellGenerator*, menite să indice scurgerea unei cuante de timp. La receptarea acestor semnale, cele două procese verifică tabelele lor locale, pentru a stabili dacă este momentul să execute pentru un VPC sau VCC o acțiune dependentă de timp. Astfel de acțiuni sunt părăsirea stării de eroare pentru o conexiune (pentru *InputHandle*) sau generarea unei celule de ieșire (pentru *FMCellGenerator*).

InputHandle poate recepționa semnale de la sistemul pentru management, prin care i se cere efectuarea anumitor servicii (de exemplu, verificarea continuității, activarea sau dezactivarea monitorizării performanțelor), iar ca efect, *InputHandle* produce celulele de ieșire indicate în standard.

Dacă *InputHandle* găsește la *Input1* o celulă de intrare, el o procesează, executând activitățile specificate de tipul celulei:

- Dacă celula de intrare este o *celulă FM* (celulă AIS sau RDI), aceasta indică apariția unei erori fizice pentru o anumită direcție. Drept urmare, procesul stabilește starea de eroare pentru VPC-ul sau VCC-ul implicat și semnalizează procesului *FMCellGenerator* că trebuie să producă și să transmite celule RDI în direcție opusă.
- *Celulele loop back*, care ajung la nodurile capăt ale unei conexiuni sau segment de conexiune, sunt întoarse în direcția opusă. Când celule întoarse ajung la nodul unde au fost generate, procesul *InputHandle* semnalizează printr-un semnal special sistemul pentru management, că serviciul de loop back a reușit. Acest fapt îi arată acestuia că direcția verificată este continuă.
- *Celulele utilizator* sunt transmise prin blocul F4 fără a se opera nici un fel de procesare asupra lor. Când un VPC sau un VCC este monitorizat, atunci celulele utilizator de pe acea direcție sunt numărate, pentru a putea apoi calcula mărimile statistice de monitorizare.

- Când *InputHandle* primeşte o *celulă pentru activare/dezactivare*, el verifică dacă cererea transmisă prin celulă este suportată de nod. În caz afirmativ, conexiunea specificată în cerere intră în starea de monitorizare şi *InputHandle* produce o celulă de ieşire de tip PM. Dacă însă cererea nu este suportată, atunci *InputHandle* transmite sistemului pentru management un semnal special de refuz al cererii.
- *Celulele pentru monitorizarea performanţei* transportă informaţii statistice spre nodurile de monitorizare. *InputHandle* culege în aceste noduri informaţia statistică purtată de celulele pentru monitorizare, pe care apoi o foloseşte în calculul mărimilor ce descriu gradul în care funcţionarea reţelei se abate de la cea dorită.

Celulele care ajung la *InputHandle* prin *Input2* parcurg blocul F4, fără a se opera nici un fel de procesare asupra lor. Pentru conexiuni monitorizate, celulele de intrare sunt numărate şi se calculează informaţia statistică corespunzătoare. Celule PM sunt generate şi trimise la ieşire, după fiecare 1024 de celule de intrare recepţionate.

Funcţionalitatea de bază a blocului F4 este cuprinsă în procesele *InputHandle* şi *FMCellGenerator* [DH94] conţine specificările celor două procese, descrise într-un pseudocod apropiat limbajului VHDL.

Celulele transmise de *InputHandle* şi *FMCellGenerator* sunt gestionate de cele două buffere de ieşire (*OutputHandler_1* şi *OutputHandler_2*). Celulele de ieşire sunt transmise în continuare porturilor de ieşire, într-o ordine ce respectă prioritatea lor.

7.3.2 Implementarea şi testarea modelului

Modelul a fost dezvoltat în VHDL şi simulat pe staţii SUN SPARC ELC folosind simulatorul VHDL MINT 2.5 [SY93]. Codul ce descrie funcţionalitatea modelului conţine aproximativ 1600 de linii de program, iar partea dezvoltată pentru testarea şi validarea sa (figura 7.6) cuprinde aproximativ 1400 de linii de cod VHDL. Componenta pentru testarea şi validarea modelului [SV94] implementează următorul principiu: la intrarea blocului F4 sunt generate celule de diferite tipuri şi se compară celulele şi semnalele de ieşire produse de model cu cele care ar trebui create conform standardului ATM.

Controler-ul (figura 7.6) iniţiază generarea celulelor de un anumit tip, transmiţând către *Sender1* şi *Sender2* semnalele corespunzătoare. Ca urmare a semnalelor primite de la *Controler*, *Sender1* produce celule pentru portul *Input1*, iar *Sender2* pentru portul *Input2*. Totodată, cele două sendere crează şi celulele de ieşire, care ar trebui produse pentru o funcţionalitate F4 corectă. "Mostrele" de celule corecte sunt transmise celor două receivere (*Receiver1* şi *Receiver2*), câte unul pentru fiecare ieşire a blocului. În urma recepţionării celulelor de intrare, modelul F4 produce celule de ieşire, pe care în continuare, le comunică celor două receivere. Receiverele primesc celule atât de la blocul F4, cât şi de la sendere, le compară, iar rezultatul comparaţiei este raportat proiectantului.

Prin această schemă, testarea şi validarea modelului s-a făcut practic automat. Procesul de testare şi validare a cuprins generarea şi experimentarea a numeroase situatii de test, precum şi verificarea rezultatelor produse pentru ele.

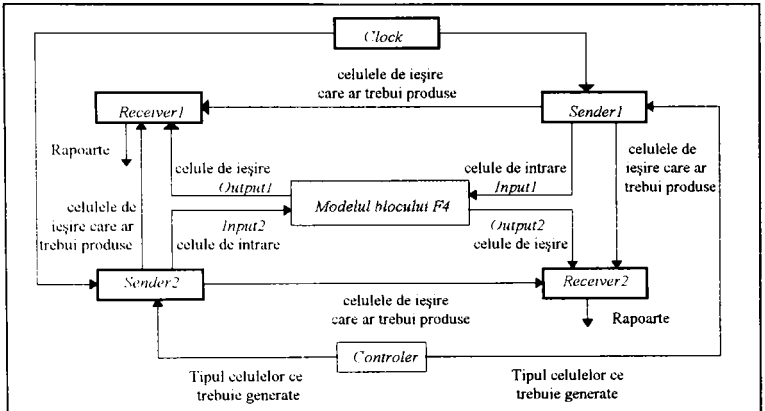


Figura 7.6 Structura folosită pentru testarea şi validarea modelului lui F4

7.3.3 Descrierea constrângerilor de timp pentru F4

Pasul următor în realizarea modelului pentru F4 are scopul de a descrie constrângerile de timp prevăzute de standardul ATM [BE93]. Deciziile de proiectare hardware/ software (*sinteza* arhitecturii finale, *legarea* funcţionalităţii proceselor pe resurse, *transformarea* specificării) sunt luate folosind aceste constrângeri împreună cu cele pentru cost, numărul şi tipul resurselor disponibile.

La această oră, constrângerile de timp *nu* sunt descrise în cadrul specificării modelului. Proiectantul le culege, sistematizează şi gestionează manual. Lucrul acesta este perfect rezonabil pentru metodologia noastră curentă de proiectare, în care deciziile sunt luate de proiectant, iar calitatea lor este estimată automat prin uneltele noastre de estimare (de exemplu planificatorul). Dacă calitatea implementării nu corespunde restricţiilor de timp ale standardului, proiectantul încearcă alte decizii de proiectare.

Într-un mediu de proiectare ce parcurge automat etapele spre implementarea finală, este necesară o notaţie pentru descrierea constrângerilor de timp. Literatura de specialitate oferă două căi pentru aceasta: adnotarea specificărilor ce descriu funcţionalitatea modelului sau descrierea lor într-un formalism dedicat.

Standardul ATM [BE93] defineşte pentru blocul F4 următoarele constrângeri de timp:

- Viteza de transmisie a *celulelor utilizator* poate fi până la 150 Mbs pentru fiecare canal serial de intrare. Aceasta înseamnă că un octet apare la intrarea în medie la

aproximativ 50 ns, iar o celulă ATM (53 de octeți) este recepționată în medie la fiecare 2.6 μs.

- Pentru fiecare direcție monitorizată, la fiecare 1024 de celule utilizator este generată o *celulă PM* nouă. Aceasta înseamnă creșterea traficului de celule pe acea direcție cu aproximativ 0.097%.
- *Celulele AIS* și *RDI* sunt produse pentru conexiunile eronate cu o frecvență de 1 celulă pentru fiecare secundă. Dacă blocul F4 are *k* intrări, la limită (dacă apar erori pentru toate conexiunile nodului) numărul maxim de celulele AIS și RDI ce traversează blocul într-o secundă este *k*.
- *Celulele loop back* apar pentru o direcție la maxim 3 secunde. Pentru cele *k* căi de intrare, blocul F4 este traversat în medie într-o secundă, în cazul cel mai defavorabil (când se verifică continuitatea tuturor conexiunilor), de *k/3* celule loop back.
- *Protocolul de activare și dezactivare* folosește 2 celule specifice pentru fiecare cerere formulată de sistemul pentru management. Standardul ATM nu face nici o referire la cât de des pot apare în realitate cererile de la sistemul pentru management. Pentru 1 cerere/secundă/conexiune, numărul maxim de celule de activare și dezactivare ce traversează nodul într-o secundă este de $2 \times k$ celule (pentru fiecare direcție avem o celulă cerere și una răspuns).

Aceste informații descriu constrângerile temporale definite de standardul ATM. Ele sunt folosite în conducerea procesului de proiectare și totodată, în evaluarea măsurii în care o soluție de implementare satisface cerințele temporale impuse.

7.4 Rafinarea modelului în vederea co-sintezei hardware/software

Specificarea de sistem detaliată în paragraful 7.3.1 are menirea de a descrie *doar* funcționalitatea blocului F4. Ea nu cuprinde informații despre constrângerile de timp, de arhitectură, de tehnologie, etc. pe care implementarea trebuie să le satisfacă în final. Putem afirma că, modul în care am conceput structurarea specificării urmărește să ușureze munca de modelare, simulare și verificare, ceea ce nu este de loc trivial, din cauză că funcționalitatea lui F4 este complexă și complicată. În continuare, modelul este rafinat prin decizii de proiectare și introducerea treptată a elementelor de nivel din ce în ce mai scăzut (memorii, buffere, magistrale, logică de interfațare) și care aduc specificarea tot mai aproape de implementarea finală. Nivelul specificării este coborât pe măsură ce informația necesară devine disponibilă în întregime.

Deasemenea, introducerea în specificarea de un anumit nivel a unor aspecte *relevante* ce aparțin rafinărilor de nivel mai scăzut, are darul de a mări acuratețea și calitatea deciziilor de proiectare și a metodelor de estimare. Planificarea proceselor operează asupra grafului dependențelor de date și de control, a cărui noduri corespund proceselor sau secvențe de instrucțiuni. Pentru că timpul lor de execuție este comparabil cu cel al operațiilor de lucru cu memoria, are sens să introducem în graf informații de nivel mai scăzut, în efortul nostru de a obține o planificare cât mai scurtă. Operațiile de lucru cu memoria cuprind activități ce folosesc resurse diferite (circuite de memorie, magistrale de date). Tratarea separată a acestor activități în graful proceselor este o sursă potențială pentru creșterea calității planificării. Secțiunea 7.5.2

detaliază acest aspect. Prin urmare, este benefic ca proiectantul să aibă în vedere la luarea deciziilor de nivel ridicat şi anumite elemente ce aparţin nivelelor mai coborâte.

Prima decizie de rafinare a specificării lui F4 îi adaugă primele informaţii referitoare la arhitectura implementării: modulele pentru operaţiile de intrare-ieşire şi memoria pentru păstrarea tabelelor locale de date. Această rafinare se referă la elemente arhitecturale *de principiu*, ce nu ţin de aplicaţia particulară proiectată (de exemplu, păstrarea structurilor mari de date în memorii şi nu în regiştrii) şi mai puţin la decizii specifice aplicaţiei (de exemplu, folosirea anumitor tipuri de resurse hardware - procesoare, circuite de memorie, magistrale, etc.). Deciziile luate şi elementele adăugate specificării au un caracter generic (de principiu). La acest nivel, elementele introduse în rafinare nu sunt suficient de detaliate pentru a putea fi implementate direct. De exemplu, pentru memorii nu s-a stabilit modul lor de organizare (lungimea cuvântului de memorie, numărul circuitelor de memorie, distribuirea datelor pe modulele de memorie) şi nici tipul circuitelor folosite pentru implementarea lor. Aceste aspecte vor fi decise în faze ulterioare ale procesului de proiectare hardware/software, pe măsură ce toate informaţiile necesare luării acestor decizii devin disponibile.

Datele de intrare sunt transmise serial spre bloc, unde sunt "împachetate" în octeţi, iar apoi procesate. Conversia serial-paralel, precum şi toate verificările legate de corectitudinea recepţiei biţilor de intrare sunt operate de două buffere de intrare (unul pentru fiecare direcţie). Celulele de ieşire sunt convertite paralel-serial de către două buffere de ieşire. Bufferele de intrare şi de ieşire au şi rolul de adaptare a vitezei de funcţionare a lui F4 la vitezele de recepţie şi de transmitere a celulelor. Bufferele pentru intrări şi ieşiri sunt adăugate modelului din 7.3, iar noua structură este redată în figura 2.15.

Procesele *InputHandle* şi *FMCellGenerator* cuprind tabele locale de dimensiuni mari. Dacă aceste tabele ar fi sintetizate, atunci ele ar folosi un număr foarte mare de regiştrii. O serie de factori (preţ de cost foarte mare, arie mare de integrare, număr de pini, etc.) fac însă această soluţie de nefolosit din punct de vedere practic. Structurile de date mari, proprii celor două procese le păstrăm în memorii. Aceasta impune modelarea funcţionării memoriilor prin procese noi. Figura 2.15 conţine procesul dedicat, ce păstrează ambele tabele locale.

Modelul descris în paragraful 7.3.1 cuprinde procese organizate astfel încât să reflecte fluxul de date, dar el nu face nici un fel de referire la posibilitatea de a implementa eficient funcţionalitatea sa în hardware şi/sau în software. Modelul cuprinde două procese mari (*InputHandle* şi *FMCellGenerator*), cu o funcţionalitate complexă şi complicată. În continuare, justificăm şi prezentăm modul de descompunere a acestor procese mari în procese mai mici, care să fie uşor de implementat.

InputHandle procesează celulele şi semnalele de intrare, iar *FMCellGenerator* execută activităţile periodice, corespunzătoare stărilor în care sunt conexiunile nodului. *InputHandle* recepţionează celule de intrare cu tipuri diferite: celule utilizator, celule FM, celule PM şi celule pentru activare/dezactivare, iar procesările executate sunt specifice fiecărui tip. Semnalele de intrare pot fi generate de nivelul fizic sau sistemul pentru management şi deasemenea, ele implică activităţi diferite funcţie de natura lor. Prin urmare, *InputHandle* are o funcţionalitate foarte heterogenă. De aceea, este "de

bun simţ" decizia noastră de a distribui funcţionalitatea lui *InputHandle* pe procese mici, dar care să modeleze un singur tip de activitate şi care să fie mai uşor de implementat. Decizia noastră este justificată şi de următoarele argumente "formale":

- Instrucţiunile ce realizează o anumită funcţionalitate sunt legate între ele prin relaţii de precedenţă şi operează de regulă asupra aceloraşi date proprii. Din cauza relaţiilor de precedenţă, instrucţiunile unei funcţionalităţi sunt în mare parte secvenţiale (deşi nu în totalitate) şi executarea lor de către acelaşi procesor este justificată. Procesele rezultate în urma rafinării stabilesc granularitatea de lucru şi sunt punctul de pornire pentru construirea grafului dependenţelor de date date şi de control. Această concluzie este întărită de faptul că majoritatea datelor asupra cărora operează instrucţiunile (principiul localizării datelor) pot fi păstrate în memoria locală procesorului (doar procesorul curent are acces la ea).
- Un proces cuprinde de regulă şi instrucţiuni de control, astfel că funcţie de valorile expresiilor de test folosite, sunt executate diferite secvenţe de instrucţiuni. Dacă timpii de execuţie pentru alternativele complementare, ce aparţin aceluiaşi proces, sunt diferiţi, atunci apar inexactităţi de modelare. Acestea apar pentru că în graful dependenţelor de date şi de control fiecare nod este caracterizat printr-un timp de execuţie unic. Reducerea inexactităţilor se poate face scăzând granularitatea de lucru prin încapsularea alternativele complementare în procese distincte. Dezavantajul acestei soluţii provine din faptul că numărul nodurilor în graf poate creşte foarte mult, iar odată cu el (chiar dincolo de limite acceptabile din punct de vedere practic) şi timpul de execuţie pentru algoritmii de proiectare. În mod fericit pentru F4, dacă grupăm procesele după criteriul funcţionalităţii lor, atunci alternativele din componenţa lor au timpi de excuţie apropiaţi. Acest lucru a permis să reducem (fără a deteriora acurateţea modelării) numărul nodurilor din graf.

Revenind la proiectarea lui F4, pasul următor de rafinare duce la generarea următoarelor procese noi. Procesul *InputHandle* este descompus astfel (noua structură a modelului apare în figura 2.15):

- *cell-handler-1* "culege" celulele din bufferul de intrare şi le direcţionează spre procesele unde are loc prelucrearea lor propriu-zisă. Celulele utilizator sunt direcţionate spre *output-handler-1*, iar restul spre procesele ce execută procesarările specifice tipului lor. Totodată, dacă se recepţionează o celulă corectă pentru o conexiune în stare de eroare, atunci conexiunea revine în starea normală. Asemănător, celulele ce vin pe direcţia opusă (prin portul *Input2*) sunt preluate de *cell-handler-2* şi trimise în continuare bufferului *output-handler-2*.
- Circuitul ATM este unul foarte rapid, deoarece pentru o conexiune viteza sa de transmisie ajunge până la 150 Mbs. Pentru a "acomoda" viteza la care funcţionează *cell-handler-1*, respectiv *cell-handler-2*, cu cea a lui *output-handler-1* şi respectiv *output-handler-2*, am introdus buffere între ele (figura 2.15). Astfel, funcţionarea acestor procese este decuplată una de cealaltă.
- Funcţionalitatea lui *InputHandle* pentru celulele OAM (celule AIS, RDI şi loop back) este acoperită de procesul *FMC-handler*. Dacă acceptăm faptul că rata de apariţie a erorilor este mică, atunci acest proces este executat foarte rar. Procesul *act deact* realizează funcţiile pentru protocolul de activare şi dezactivare, *iar prf-mon-1* asigură funcţionalitatea pentru monitorizarea performanţelor pe direcţia 1. *prf-mon-2* este cel ce asigură activitatea de monitorizare a performanţelor pe direcţia opusă.

- *InputHandle* prelucrează semnalele primite de la nivelul fizic şi de la sistemul pentru management. Modelul rafinat acoperă această funcţionalitate prin procesele *PhL-interface* şi *MS-interface*.

Funcţionalitatea lui *FMCellGenerator* este destinată unui singur gen de activităţi, acela de a genera celule periodice potrivit stărilor în care sunt conexiunile blocului. *FMCellGenerator* nu trebuie descompus după criteriul funcţionalităţii şi el apare denumit în figura 2.15 ca şi *FMC-gen*

Figura 2.15 descrie structura întregului modelului, obţinut în urma pasului de rafinare. Modelul descrie funcţionalitatea lui F4, este compus din procese organizate după criteriul funcţionalităţii lor (un proces asigură o singură funcţionalitate) şi include informaţii generice despre arhitectura folosită (memorii şi buffere). Paşii următori ai co-sintezei hardware/software folosesc acest model.

7.5 Construirea grafului dependentelor de date şi de control pentru blocul F4

7.5.1 Estimarea timpilor de execuţie pentru nodurile grafului construit pentru blocul F4

Timpii de execuţie ai activităţilor reprezentate prin nodurile grafului depind fundamental de resursele pe care acestea urmează să fie executate. De aceea, tipurile resurselor ce pot intra în componenţa arhitecturii finale (biblioteca resurselor), trebuie stabilite înaintea pasului de estimare a timpilor de execuţie asociate nodurilor. Acest lucru presupune stabilirea tipurilor de procesoare, memorii, magistrale, circuite combinaţionale (multiplexoare, demultiplexoare, decodificatoare, porţi logice, etc.), circuite secvenţiale (bistabile, regiştrii, etc.), etc. ce pot fi folosite pentru realizarea arhitecturii finale. După ce s-a stabilit biblioteca resurselor disponibile, pentru fiecare nod se determină resursele, ce îi pot executa funcţionalitatea. Nu toate resursele unei biblioteci pot executa pe oricare din nodurile grafului, din cauză că nodurile modelează activităţi heterogene. De exemplu, nodurile grafului construit pentru blocul F4 reprezintă atât procesări de informaţie (descrise printr-o secvenţă de instrucţiuni), cât şi activităţi de lucru cu memoria (citiri sau scrieri în memoria globală a sistemului). Nodurile din prima categorie sunt executate folosind procesoare programabile, iar cele din a doua clasă operează asupra circuitelor de memorie. În continuare, timpul de execuţie al unui nod este estimat pentru fiecare din resursele cărora le poate fi legat.

Procesoarele programabile pe care le-am experimentat pentru realizarea arhitecturii noastre sunt 486DX2 [in90] la 80 MHz şi Pentium la 120 MHz. Ambele procesoare sunt de uz general, foarte populare la ora aceasta (bine cunoscute şi prezentate în documentaţia din domeniu), ieftine şi având o putere de calcul suficientă pentru necesarul aplicaţiei noastre.

Nodurile mapata procesoarelor programabile corespund la secvenţe de instrucţiuni. Am estimat timpul necesar pentru execuţia unei secvenţe, compilând secvenţa, executând-o şi măsurând timpul ei de execuţiei. Ca să avem o estimare cât mai apropiată realităţii,

trebuie să eliminăm influenţele a două elemente, ce pot "perturba" major măsurătorile noastre. Acestea sunt:

- *Optimizările introduse de compilator*. Fiecare compilator execută un număr de optimizări ale codului sursă (de exemplu propagarea constantelor - *constant folding*, eliminarea invarianţilor din cicluri, eliminarea variabilelor de inducţie, etc. [AS86]). Acestea sunt dependente atât de codul compilat, dar şi de tipul compilatorului folosit. Deoarece nu este evident că, compilatorul folosit la estimarea timpilor de execuţie este disponibil şi pentru procesoarele folosite în construirea implementării finale, am preferat să eliminăm influenţele compilatorului asupra codului generat. De aceea, secvenţele de instrucţiuni ce reprezintă nodurile grafului le-am compilat fără opţiune de optimizare.
- Influenţa pe care o are *prezenţa memoriei cache*. Studiul şi metoda noastră de proiectare nu consideră existenţa în arhitectura finală a memoriei cache. Dezvoltarea unor procedee pentru sinteza sistemelor cu memorie cache, poate fi un subiect al cercetării noastre viitoare.

Pentru a stabili măsura în care am eliminat influenţele sistemului gazdă (calculator, sistem de operare, compilator), am validat timpii ataşaţi nodurilor prin măsurarea directă a timpului lor de execuţie, comparându-i cu valorile calculate pe baza manualelor de procesor [in90]. Instrucţiunile fiecărui nod au fost compilate în limbaj de asamblare. După aceea, timpul asociat unui nod a fost aproximat cu suma timpilor de catalog pentru instrucţiunile de asamblare generate. Timpii de catalog au fost aleşi pentru situaţiile în care operanzii instrucţiunilor nu apar în memoria cache (*cache miss*), pentru că am dorit să eliminăm (să diminuăm cât mai mult) influenţa memoriei cache asupra timpilor de execuţie. În urma acestei validări, concluzia noastră este că timpii obţinuţi prin adunarea valorilor de catalog şi cei obţinuţi prin măsurare sunt foarte apropiaţi. Dacă acceptăm folosirea memoriei cache, respectiv folosim codul optimizat produs de compilator, atunci timpii reali de execuţie ai nodurilor sunt mai mici de cât cei estimaţi. De aceea, putem afirma că estimarea noastră este una acoperitoare celei ce apare în realitate.

A doua categorie de noduri în graf sunt cele care reprezintă operaţiile de lucru cu memoria globală a sistemului. Tabelele ce păstrează starea conexiunilor (tabelele locale lui *InputHandle* şi lui *FmCellGenerator*) sunt memorate în memoria globală a arhitecturii. Din punctul nostru de vedere, fiecare din procesoarele componente ale unei arhitecturi multiprocesor poate accesa memoria globală. Pentru a stabili timpii necesari operaţiunilor cu memoria globală (citiri şi scrieri din/în memorie), trebuie să decidem cantitatea de informaţie transferată de o operaţie cu memoria (lungimea cuvântului de memorie) şi tipul circuitelor de memorie folosite.

Lungimea unui cuvânt de memorie am stabilit-o analizând lungimea datelor implicate în transferurile cu memoria globală. Pentru că acestea sunt în general de 1 octet şi rar 2, 3, 4 sau 5 octeţi, am luat decizia ca un cuvânt de memorie să aibă lungimea de 1 octet.

În ceea ce priveşte circuitele de memorie folosite, putem alege între memorii rapide, scumpe şi cu capacitate de stocare mai mică şi memorii mai lente, ieftine şi cu capacitate de memorare mare [MA91]. Din cauză că specificarea lui F4 nu impune constrângeri de timp severe pentru funcţionalităţile ce lucrează cu memoria globală, am

ales module de memorii comune (ieftine şi cu capacitate mare de stocare). De exemplu, am folosit în proiectarea noastră circuitul CMOS SRAM HM6116 [MA91], a cărui timp de acces este de 90 ns.

După cum argumentat la începutul paragrafului 7.4, este benefică pentru creşterea calităţii proiectării, utilizarea la un anumit nivel a informaţiilor semnificative, ce aparţin unui nivel mai coborât. Astfel, lucrul cu memoria globală poate fi reprezentat din două perspective:

- Prima (pe care am numit-o *de nivel înalt*) tratează operaţiunile de scriere şi citire în/din memorie ca fiind atomice (indivizibile). Astfel, un nod corespunzător unei citiri din memorie descrie punerea adreselor pe magistrala de adrese, generarea semnalelor pentru selectarea chipului (*CS*) şi validarea ieşirii (*OE*) şi plasarea datelor pe magistrala de date. Timpul de 90 ns cuprinde toate aceste activităţi. Asemănător, nodurile pentru operaţiunile de scriere în memoria globală descriu punerea adreselor pe magistrala de adrese, generarea semnalelor *CS* şi *W* şi plasarea datelor pe magistrala de date. Între aceste activităţi nu există doar relaţii de secvenţiere şi ele folosesc fie circuite de memorie, fie magistrala de date.
- A doua abordare (pe care am numit-o *de nivel scăzut*) modelează prin noduri separate activităţile ce compun operaţiile de citire şi scriere din/în memoria globală. Aceste noduri sunt astfel create încât ele să grupeze activităţile ce folosesc aceeaşi resursă fizică.

Secvenţa tipică de semnale pentru operaţiunea de citire din memorie este [MA91]:

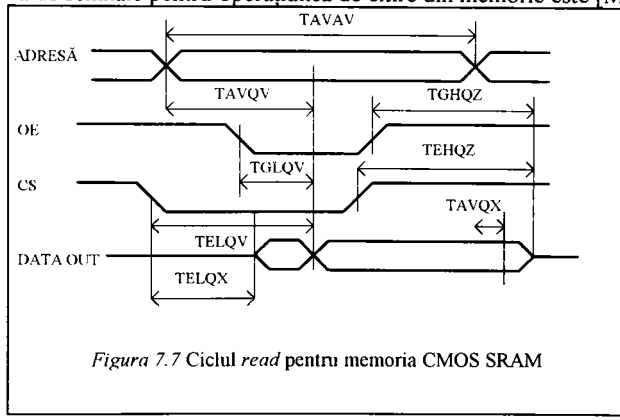


Figura 7.7 Ciclul *read* pentru memoria CMOS SRAM

Această secvenţă de operaţii am modelat-o la nivel scăzut prin următorul subgraf:

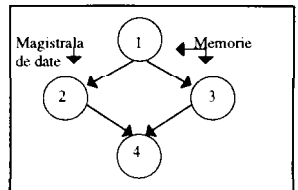


Figura 7.8 Modelarea operaţiei *read*

Nodurile 1 şi 3 se referă la operaţiile ce folosesc circuitul de memorie şi magistrala de adrese, iar nodul 2 utilizează magistrala de date. Nodul 4 este fictiv şi a fost introdus pentru a marca sfârşitul operaţiei de citire din memorie. Nodul 1 reprezintă toate operaţiile ce sunt cuprinse până în momentul în care apar primele date valide pe magistrala de date şi se referă la semnalele ADRESĂ, CS şi OE. Timpul asociat lui este TAVQV. Nodul 2 modelează punerea şi menţinerea datelor pe magistrala de date, iar timpul asociat lui este TAVAV - TAVQV + TAVQX. Nodul 3 corespunde operaţiilor cu memoria, semnalele ADRESĂ, CS şi OE, ce au loc după ce datele devin disponibile El are timpul TAVAV - TAVQV Nodurile 2 şi 3 sunt executate în paralel, după terminarea lui 1.

Operaţia de scriere în memorie este descrisă de următoarea diagramă [MA91]:

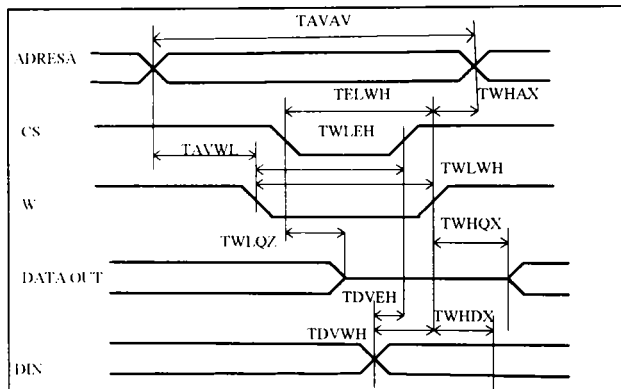


Figura 7.9 Ciclul *write* pentru memoria CMOS SRAM

Aceste activităţi le modelăm prin următoarea secvenţă de graf:

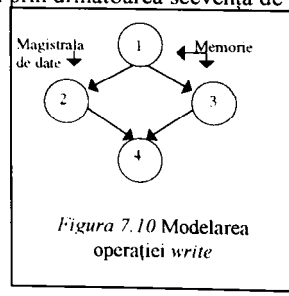


Figura 7.10 Modelarea operaţiei *write*

Nodurile 1 şi 3 descriu operaţii care folosesc circuitele de memorie, iar 2 modelează folosirea magistralei de date. Nodul 4 este un nod fictiv pentru a indica sfârşitul operaţiei *write*. Timpul pentru 1 este TAVWL + TWLWH + TWHDX - (TDVWX + TWHHDX), cel pentru 2 şi 3 este TDVWX + TWHDX.

Modelarea de nivel scăzut este utilă atunci când timpii nodurilor din graf sunt de ordinul a $n \times 10$ ns, care este ordinul de mărime al activităților ce au loc pentru scrie și citire. De exemplu, pentru o arhitectură cu două circuite de memorie care folosesc aceeași magistrală de date, are sens să detaliem secvența în care două citiri din circuite de memorie diferite folosesc magistrala comună de date. Modelarea de nivel scăzut a operațiilor de lucru cu memoria oferă o granularitate de lucru mai mică și deci, posibilitatea de obținere a unor soluții superioare celor pentru abordarea în care citirea și scrierea cu memoria sunt operații atomice.

Nodurile mapate la procesoare diferite, noduri ce interacționează între ele, își comunică datele folosind memoria globală a sistemului. Timpii necesari pentru accesul la aceste date sunt fixați de timpii de lucru cu memoria. Suplimentar, nodul sursă al transmisiei generează un semnal spre nodul țintă, pentru a-i indica că poate să-și înceapă execuția. Timpul necesar transmiterii acestui semnal este aproximat cu 30 ns, adică timpul cât este necesar ca semnalul să fie stabil pe magistrală, pentru a putea fi sesizat de către procesorul țintă [in80].

7.5.2 Alte decizii arhitecturale. Construirea grafului

Viteza de transmisie serială este de 150 Mbs pentru *fiecare* conexiune a unui circuit ATM. Aceasta înseamnă că 1octet sosește la un bloc F4 în medie la 50 ns, iar o celulă utilizator la 2.6 μs.

Părțile cele mai constrânse din funcționalitatea lui F4 sunt cele pentru tratarea celulelor utilizator și colectarea informațiilor statistice de monitorizare. Viteza de procesare de 2.6 μs/celulă/conexiune poate fi asigurată, doar dacă există câte un bloc funcțional pentru fiecare conexiune (figura 7.11). Acesta tratează celulele utilizator și calculează informațiile statistice de monitorizare. Blocul testează tipul celulelor ATM de intrare, iar cele utilizator sunt transmise mai departe bufferelor de ieșire. Celulele OAM sunt puse într-un buffer dedicat, de unde sunt luate de procesele destinate prelucrării lor. Blocurile sunt sintetizate ca ASIC-uri pentru a putea asigura viteza de lucru necesară. Dacă ne referim la figura 2.15, blocurile sintetizate acoperă funcționalitatea descrisă de *cellhandler1*, *output-handler-1*, *statistics* din *prf-mon-1*, *cell-handler-2*, *statistics* din *prf-mon-2* și *output-handler-2*. Bufferele sunt implementate folosind memorii rapide (circuitul M10E422 [MA91]), având un timp de acces de aproximativ 10 ns.

Specificările temporale ale standardului ATM indică faptul că celulele OAM apar foarte rar în fluxul de celule ATM. Din acest motiv, este acoperitor dacă realizăm printr-un singur "exemplar" al proceselor respective funcționalitățile pentru procesarea celulelor OAM ale tuturor blocurile F4 dintr-un circuit ATM (figura 7.11). Datorită cerințelor de timp ale standardului, aceste procese sunt cuprinse în componenta software și executate pe procesoare programabile. În figura 7.11 componenta software este denumită *funcționalitate OAM* Dacă ne referim la figura 2.15, procesele software sunt: *FMC handler*, *PhL-interface*, *MS-interface*, *prf. mon.* din *prf-mon-1*, *act/deact*, *FMC-gen*, *prf. mon.* din *prf-mon-2*, *clr.error status* și *inspect table*. Structura blocurilor F4 ale unui circuit ATM este cea din figura 7.11.

Co-sinteza se încheie cu implementarea componentei *funcționalitate OAM*.

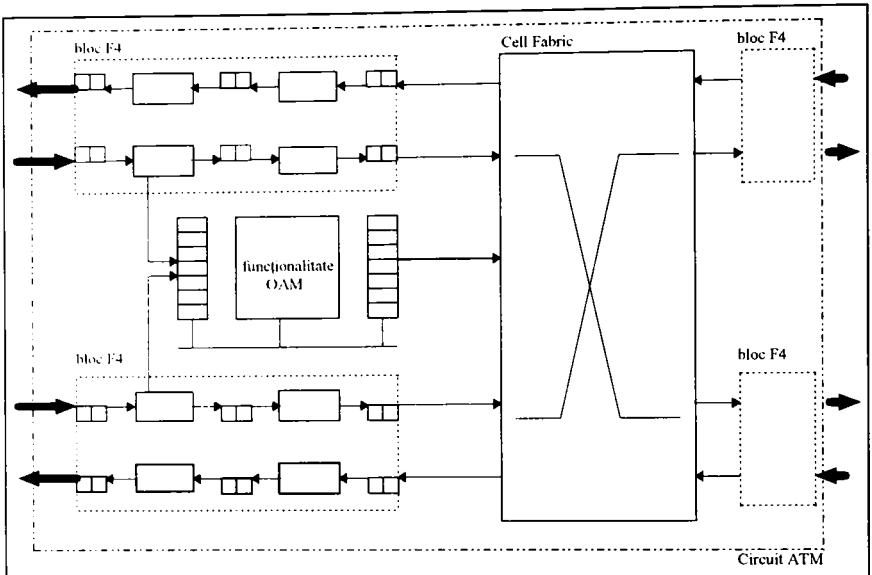


Figura 7.11 Structura blocului F4 din cadrul circuitului ATM

Pentru procesele componentei software am construit 3 grafuri, denumite *funcțional.AIS RDI*, *PhL-interf.&MS-interf.* și *Perf.Monit&Act/Deact*. Ele au 23, 32 și respectiv 42 de noduri. Paragraful următor discută variantele experimentate pentru arhitectura destinată execuției software-ului și modul de legare a nodurilor pe resurse. Calitatea fiecărei arhitecturi și felul de legare este evaluat planificând automat nodurile și comparând timpii rezultați pentru cele 3 grafuri planificate cu cerințele temporale ale standardului ATM.

7.6 Experimentarea soluțiilor de implementare.

În urma etapelor precedente de proiectare, am construit graful dependențelor de date și de control pentru F4 și am stabilit biblioteca de resurse folosită pentru realizarea arhitecturii. Nodurile grafului modelează activitățile ce compun funcționalitatea lui F4, iar arcele descriu relațiile între noduri. Fiecare nod are atașată mulțimea timpilor de execuție pentru toate resursele de care el poate fi legat.

Biblioteca noastră de resurse cuprinde procesoare 486DX2 și Pentium, circuite de memorie CMOS SRAM HM6116 și M10E422, magistrale și logică de interfațare. Numărul de componente din fiecare tip de resurse este practic nelimitat.

Co-sinteza hardware/software continuă cu luarea deciziilor de către proiectant privind sinteza arhitecturii şi legarea nodurilor de resursele sale. Cei doi paşi sunt parcurşi succesiv: sinteza arhitecturii stabileşte numărul, tipul şi topologia de interconectare a resurselor din arhitectură, iar apoi legarea nodurilor identifică pentru fiecare nod resursa care-l execută. Pentru fiecare arhitectură şi mod particular de legare, planificăm automat graful şi obţinem astfel estimările timpilor săi de execuţie pentru toate valorile posibile ale condiţiilor din graf. Dacă timpul cel mai lung satisface constrângerile standardului, atunci se încearcă o arhitectură mai ieftină. Aceasta înseamnă fie o arhitectură cu mai puţine resurse sau una cu resurse mai ieftine, dar cu performanţe mai slabe. Dacă însă timpul nu se încadrează în constrângerile specificate, atunci se încearcă fie o legare a nodurilor mai favorabilă, fie o arhitectură cu mai multe resurse sau cu resurse mai performante. Proiectarea decurge iterativ prin paşi succesivi de sinteză a arhitecturii, legare a nodurilor, planificare şi evaluarea calităţii proiectării, până când calitatea implementării este cea dorită. Structura lui F4 face ca numărul variantelor de arhitectură şi de legare să fie relativ mic şi să poată fi explorate manual. În schimb, planificarea foloseşte un număr de câteva zeci de noduri şi este greu de realizat în absenţa unui planificator automat.

Legarea nodurilor pe resurse urmăreşte să favorizeze obţinerea unui timp de execuţie a grafului cât mai scurt. Acest lucru îl realizăm prin următoarele obiective ale legării:

- *Să minimizăm volumul comunicărilor între procesoare.* De aceea, grupurile de noduri care comunică intens între ele sunt executate de acelaşi procesor. Graful blocului F4 este format, în general, din secvenţe independente (lanţuri) de noduri. Nodurile ce compun acelaşi lanţ sunt executate de preferinţă de acelaşi procesor.
- *Să maximizăm gradul de concurenţă în sistem.* Activităţile concurente sunt plasate în mod preferenţial pe resurse diferite. În cazul lui F4, lanţurile sale independente sunt legate la resurse distincte. Acest fapt măreşte potenţialul ca anumite secvenţe concurente din graf să fie executate în paralel. Legarea lanţurilor de resurse este astfel încât încărcarea resurselor să fie cât mai apropiată, eliminându-se eventualele "gâtuiri" ale execuţiei datorită resurselor încărcate puternic.
- *Nodurile cu complexitate computaţională mare sunt legate resurselor rapide.* Dacă luăm decizia să mutăm anumite noduri de pe resursa unde sunt executate restul lanţului căreia el aparţine, atunci verificăm dacă câştigul în timpul său de execuţie, datorat procesorului mai rapid, compensează timpul suplimentar din cauza comunicărilor cu alte procesoare.

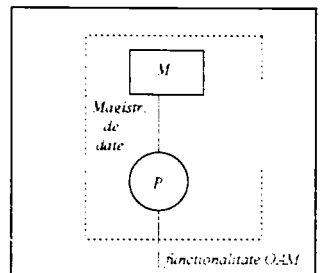


Figura 7.12 Varianta I de arhitectură pentru *functionalitate OAM*

Structura grafului pentru blocul F4 este relativ simplă, fiind formată din lanțuri independente de noduri. Majoritatea acestor lanțuri reprezintă alternative complementare, astfel că execuția lor se exclude reciproc. Numărul lanțurilor independente, executate pentru aceeași mulțime de condiții, este mic, ceea ce face ca numărul alternativelor de legare pentru o arhitectură să fie deasemenea mic. Pentru fiecare arhitectură experimentată este relativ simplu să stabilim care este acea legare ce duce la timpul minim de execuție a grafului. Indicăm legarea folosită pentru toate arhitecturile prezentate în continuare.

Pentru a ne face o imagine despre timpul global de execuție al grafului, dar și pentru a culege informații utile sintezei de arhitectură și legării, primul experiment folosește pentru blocul *funcționalitate OAM* varianta de arhitectură cea mai simplă și mai puțin performantă. Structura acesteia apare în figura 7.12.

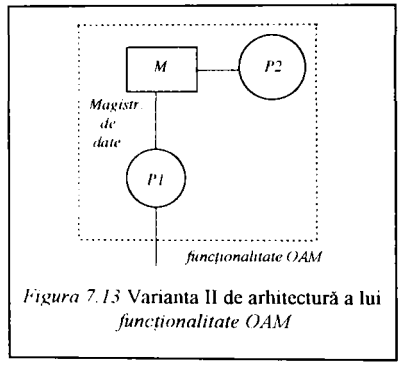


Figura 7.13 Varianta II de arhitectură a lui *funcționalitate OAM*

Arhitectura cuprinde următoarele componente: procesorul programabil *P*, căruia sunt mapate procesele *FMC handler*, *PhL-interface*, *MS-interface*, *prf. mon.* din *prf-mon-1*, *act deact*, *FMC-gen*, *prf. mon.* din *prf-mon-2*, *clr.error status* și *inspect table*. Memoria *M* păstrează ambele tabele de date, iar *P* comunică cu *M* folosind o magistrală de date. *P* este 486DX2.

Cele 3 grafuri ale dependențelor de date și de control sunt actualizate pentru arhitectura și legarea curentă. Ele sunt planificate automat și pentru fiecare reținem timpii de execuție cei mai lungi. Apoi, pentru aceeași arhitectură, considerăm că *P* este Pentium și reluăm planificarea celor trei grafuri. Tabelul următor sistematizează timpii cei mai lungi (în unități de timp) ai celor 3 grafuri, obținuți pentru cele două variante de procesor. Timpul de execuție al grafurilor scade când folosim un procesor mai puternic.

	486DX2	Pentium
PhL-interf. & MS-interf.	4471 ns	2701 ns
funcțional. AIS/RDI	1732 ns	1167 ns
Perf. Monit & Act/Deact	5852 ns	3548 ns

Experimentul identifică pentru cele 3 grafuri atât setul de condiții ce activează trace-ul cel mai lung, cât și trace-ul respectiv și furnizează timpii necesari execuției lor. Trace-urile cele mai lungi sunt analizate pentru stabilirea secvențelor independente de noduri,

care ar putea fi executate în paralel Secvențele concurente ale grafurilor reprezintă atât procesări ce folosesc procesoare programabile, cât şi operații de lucru cu memoria. Arhitecturii i se alocă resurse noi destinate execuției activităților concurente, ceea ce poate reduce timpii totali ai celor 3 grafuri. Strategia noastră pentru sinteza arhitecturii alocă şi/sau procesoare programabile, module de memorie şi magistrale de date noi.

A doua variantă de arhitectură surprinde impactul pe care introducerea de procesoare programabile noi îl are asupra celor 3 timpi de execuție. În primul pas determinăm pentru fiecare din trace-urile cele mai lungi numărul maxim de activități de procesare (care folosesc procesoare programabile) ce se pot desfăşura concurent. Apoi, alocăm procesoare noi pentru a executa aceste activități în paralel. *funcțional.AIS/RDI* nu conține activități concurente, iar *PhL-interf.&MS-interf.*şi *Perf.Monit&Act/Deact* au cîte o pereche. De aceea, adăugăm 1 procesor arhitecturii şi legăm fiecare lanț independent unui procesor diferit. Tabele sunt păstrate în continuare în acelaşi modul de memorie, la care au acces ambele procesoare din sistem. Arhitectura care se obține este în figura 7.13.

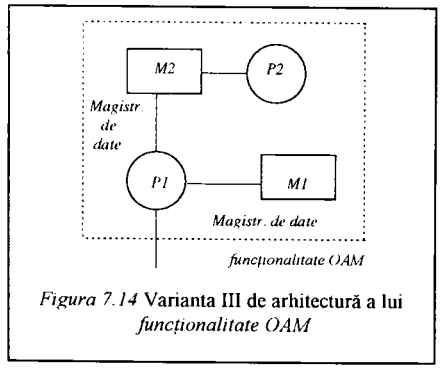


Figura 7.14 Varianta III de arhitectură a lui *funcționalitate OAM*

În primul caz *P1*şi *P2* sunt Pentium, în al doilea *P1* este Pentium, iar *P2* este 486DX2, iar în al treilea, atât *P1* cât şi *P2* sunt 486DX2. Cele trei 3 grafuri sunt reactualizate noii arhitecturi şi legări, după care sunt planificate automat. Timpii care rezultă pentru ele în cele trei situații sunt cuprinşi în tabelul următor.

	2 × Pentium	Pentium + 486DX2	2 × 486DX2
PhL-interf.&MS interf.	2131 ns	2532 ns	2932 ns
funcțional.AIS/RDI	1167 ns	1167 ns	1732 ns
Perf. Monit & Act/Deact	3548 ns	3548 ns	5033 ns

Timpii lui *funcțional.AIS/RDI* nu se modifică față de primul experiment, din cauză că el nu conține procesări concurente care să beneficieze în urma procesorului suplimentar. Timpii lui *PhL-interf.&MS interf.* scad prin folosirea unui procesor suplimentar. O arhitectură cu un procesor Pentium este mai rapidă de cât una cu două 486DX2, dar mai lentă de cât una cu un 486DX2 şi un Pentium sau una cu două Pentium-uri. În cazul lui *Perf.Monit&Act/Deact* cea mai bună variantă este să folosim un singur Pentium, din cauză că timpul care se obține este identic pentru un 486DX2 şi un

Pentium sau două Pentium-uri. Două 486DX2 duc la un timp mai mic de cât unul singur, dar sunt mai lente de cât o arhitectură cu un singur Pentium.

Al treilea experiment încearcă să îmbunătățească în continuare timpii pentru cele 3 grafuri. Posibilitatea stă în alocarea de resurse noi la arhitectura precedentă, care să permită operații concurente cu modulele de memorie. Analizăm trace-urile cele mai lungi pentru cele 9 situații ale experimentului 2 și constatăm că *PhL-interf.&MS interf.* are o pereche de operații cu memoria, ce s-ar putea executa în paralel. Alocăm un modul nou de memorie, pentru ca tabele să fie pastrate în circuite separate. Astfel rezultă arhitectura cea mai performantă, dar și cea mai scumpă. Ea cuprinde 2 procesoare, ce permit execuția paralelă a lanțurilor independente, 2 module de memorie și 2 magistrale de date pentru accesul concurent la cele două tabele. Arhitectura care rezultă este prezentată în figura 7.14.

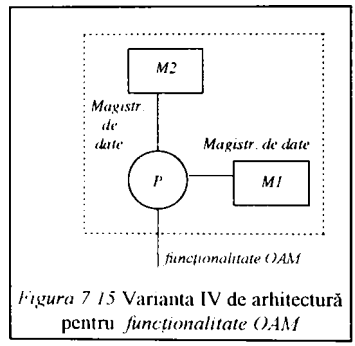


Figura 7.15 Varianta IV de arhitectură pentru *funcționalitate OAM*

Pentru această variantă de arhitectură, experimentăm cazul când *P1* și *P2* sunt Pentium, cel în care *P1* este Pentium și *P2* este 486DX2 și deasemenea, când *P1* și *P2* sunt ambele 486DX2. Timpii care se obțin în urma experimentelor sunt sintetizați în următorul tabel.

	2 × Pentium	Pentium + 486DX2	2 × 486DX2
PhL-interf.&MS interf.	1932 ns	2532 ns	2932 ns
funcțional AIS/RDI	1167 ns	1167 ns	1732 ns
Perf. Monit & Act/Deact	3548 ns	3548 ns	5033 ns

Timpul se îmbunătățește doar pentru *PhL-interf.&MS interf.* executat pe o arhitectură cu două Pentium-uri, deoarece aceasta este singura situație în care trace-ul cel mai lung cuprinde operații concurente, care folosesc modul suplimentar de memorie. În restul situațiilor timpii sunt identici cu cei din experimentul 2, astfel că din punctul de vedere al acestora nu se justifică al doilea modul de memorie.

Al patrulea experiment stabilește maniera în care se modifică timpii totali ai grafurilor, atunci când arhitectura permite doar operații concurente de lucru cu memoria. Arhitectura este mai simplă de cât cea de la experimentul trei. Adăugăm la arhitectura experimentului unu 2 module de memorie, câte unul pentru fiecare tabelă, și 2 magistrale distincte de date, astfel încât aceste operații să poată fi desfășurate în paralel

(figura 7.13). În primul experiment *P* este 486DX2, iar în al doilea Pentium. Tabelul următor prezintă timpii care se obțin pentru cele 3 grafuri.

	486DX2	Pentium
PhL-interf. & MS-interf.	4471 ns	2701 ns
funcțional. AIS/RDI	1732 ns	1167 ns
Perf. Monit & Act/Deact	5852 ns	3548 ns

Dintre cele 3 grafuri, doar *PhL-interf.&MS-interf.* conține o pereche de operații ce accesează tabele diferite și care pot fi executate în paralel. Aceste operații nu aparțin însă celui mai lung trace din această situație, astfel că timpii ce rezultă sunt identici celor din experimentul 1. Concluzia foarte interesantă este că pentru o arhitectură cu un singur procesor nu se justifică introducerea unui modul suplimentar de memorie, din cauză că trace-urile cele mai constrânse nu beneficiază de pe urma noi resurse.

Pentru F4 în mod particular, introducerea unor aspecte de nivel coborât nu a dus la scăderea timpilor totali pentru cele trei grafuri.

7.7 Concluzii

Capitolul 7 aplică metodologia noastră de co-sinteză hardware/software unui exemplu din industrie, blocul F4 al circuitului ATM. Acest exemplu arată că este dificil de indicat la începutul co-sintezei, structura necesară și particularizată a ciclului de proiectare, pe motiv că sunt greu de precizat informațiile absente, dar determinante în luarea deciziilor de proiectare. Acestea trebuie calculate înaintea deciziilor unde sunt folosite, astfel că ele stabilesc secvența de parcurgere a etapelor de proiectare și determină "forma" ciclului de proiectare. F4 este caracterizat de o funcționalitate complexă, constrângeri de timp reduse și un număr mic al variantelor de arhitecturi pentru implementare, respectiv de legare a activităților de resurse. De aceea, co-sinteza cuprinde în special pași de rafinare a specificărilor VHDL spre implementare și mai puține decizii privind structura arhitecturii și modul de legare. În final sunt propuse patru variante de arhitecturi, care se deosebesc prin numărul, tipul și maniera de interconectare a resurselor, și se evaluează calitatea acestor soluții.

Capitolul debutează cu prezentarea principiilor de bază ale protocolului ATM, funcționarea circuitului ATM și cu detalierea funcționalității blocului F4. Acesta este responsabil cu gestiunea și localizarea erorilor, monitorizarea funcționării circuitului și realizarea protocolului de activare/dezactivare a monitorizării performanțelor.

Model VHDL construit inițial pentru F4 descrie doar funcționalitatea sa, fără a conține constrângerile de timp ale implementării sau detalii arhitecturale. Organizarea lui ușurează modelarea, simularea și verificarea, ceea ce nu este de loc trivial pentru o funcționalitate complexă și complicată ca cea a lui F4. Verificarea modelului construit este automată și funcționează după următorul principiu: se generează automat atât celule de intrare de diferite tipuri, cât și celulele de ieșire care ar trebui produse la o funcționare corectă a lui F4. Celulele de ieșire sunt comparate cu cele produse de model, iar eventualele nepotriviri sunt raportate proiectantului. Prima etapă de

modelare este completată de sistematizarea constrângerilor de timp introduse de standardul ATM. Acestea sunt culese şi gestionate manual, acest mod fiind adaptat la metodologia curentă de proiectare, în care deciziile de proiectare sunt luate manual, iar calitatea lor este analizată folosind uneltele noastre software pentru estimare.

Specificării iniţiale îi sunt introduse elemente de nivel mai scăzut (memorii, buffere, circuite de interfaţare), pe măsură ce informaţia necesară acestor decizii devine disponibilă. Prima rafinare a specificării grupează funcţionalitatea lui F4 în procese, astfel încât fiecare proces să realizeze o singură funcţie de procesare. Gruparea funcţiilor distincte în procese diferite scade granularitatea de lucru şi ajută punerea în evidenţă a dependenţelor de date şi de control între procese. Totodată, specificării îi sunt adăugate elemente *generice* de arhitectură, cum sunt păstrarea structurilor mari de date în memorii şi folosirea bufferelor pentru a acomoda vitezele diferite de funcţionare ale proceselor. Specificarea rafinată este mai apropiată de implementare de cât cea iniţială şi este destinată deciziilor următoare de sinteză.

Următoarea etapă de proiectare produce grafurile dependenţelor de date şi de control, folosite în sinteza arhitecturii, legarea funcţionalităţii de resurse şi planificarea activităţilor. Înainte de a estima timpii de execuţie ai activităţilor lui F4, am stabilit toate resursele ce pot intra în structura arhitecturii finale (procesoare programabile, circuite de memorie, logică de interfaţare). Pentru fiecare activitate am identificat acele resursele ce o pot realiza, după care am estimat timpul lor de execuţie. Din cauză că nu am dispus de o "imagine" a acestor valori, am folosit două metode ortogonale de estimare şi am confruntat rezultatele lor. Fiecare secvenţă de instrucţiuni a fost compilată fără optimizări de cod, iar timpul ei de execuţie a fost stabilit pe baza manualului de procesor. Influenţa memoriei cache a fost eliminată considerând în permanenţă că operanzii nu sunt în cache. A doua metodă a fost cea directă, de a compila şi executa codul şi de a înregistra explicit timpul lui de execuţie. Timpii rezultaţi prin cele două metode ortogonale sunt apropiaţi şi sunt o aproximare acoperitoare pentru valorile din realitate. Timpii operaţiilor de lucru cu memoria i-am aproximat după ce am stabilit modul de organizare a memoriei şi am determinat volumul de date implicat în fiecare operaţie. Specific blocului F4 este faptul că activităţile care-l compun au timpul de execuţie de ordinul zecilor de ns, ceea ce corespunde ordinului de mărime al timpilor pentru operaţiile de citire şi scriere din/în memorie. Din acest punct de vedere, este interesant să descopunem operaţiile de lucru cu memoria în primitivele lor (generarea adreselor, semnale de selecţie, plasarea datelor pe magistrale, etc.) şi să le tratăm separat în procesul de proiectare. Din punct de vedere al modelării, acestea sunt operaţii de nivel scăzut, dar din punctul de vedere al timpului lor de execuţie ele sunt absolut comparabile cu restul activităţilor din graf.

Având la dispoziţie imaginea privind timpii de execuţie ai activităţilor din F4, precum şi pe cea a constrângerilor de timp ale standardului ATM, am restructurat pentru a doua oară specificarea lui F4 astfel încât satisfacerea constrângerilor de timp să devină posibilă. Această restructurare comasează funcţionalitatea identică şi rar folosită, ce aparţine blocurilor F4 ale circuitului ATM, într-un singur set de procese destinat componentei software. Acestea produce trei grafuri ale dependenţelor de date şi de control, folosite mai departe pentru deciderea tuturor detaliilor componentei software. Funcţiile F4 a căror viteză hotăresc viteza de funcţionare ATM sunt mapate componentei hardware şi sintetizate ca ASIC-uri.

Metoda de proiectare este în continuare o aplicare iterativă a paşilor de alocare a resurselor arhitecturii, de legare a nodurilor de resurse şi de evaluare a calităţii implementării. Calitatea este evaluată planificând automat cele trei grafuri alocate resurselor şi comparând timpii lor totali cu cerinţele standardului ATM. Maniera de legare urmăreşte să minimzeze volumul de comunicări între procesoare, să maximizeze gradul de concurenţă în sistem şi să lege nodurile cu complexitate computaţională mare la resursele rapide. Aceste cerinţe au darul de a conduce spre timpi de execuţie ai grafurilor cât mai mici. Primul experiment foloseşte arhitectura cea mai simplă şi cea mai uşor de decis, din cauză că informaţia utilizată în sinteza arhitecturii este relativ săracă. În urma experimentului, identificăm trace-urile cele mai lungi, timpii lor de execuţie precum şi cauzele care determină alungirea timpilor. A doua arhitectură experimentată cuprinde două procesoare, pentru a permite execuţia în paralel a secvenţelor concurente. Examinând rezultatele obţinute, am observat că un potenţial de îmbunătăţire a timpilor de execuţie stă în folosirea mai multor module de memorie, care să permită lucrul în paralel cu memoria. În urma acestei decizii de alocare se obţine arhitectura cea mai scumpă, care conţine două procesoare şi două module de memorie. Timpii rezultaţi pentru cele trei grafuri sunt cei mai scurţi. A patra variantă de arhitectură încearcă o soluţie mai ieftină, ce cuprinde un procesor şi două module de memorie. Concluzia acestui experiment a fost că această arhitectură nu creşte performanţele implementării faţă de varianta cea mai ieftină. Fiecare din cele patru variante de arhitectură a fost experimentată când procesoarele programabile sunt 486DX2 sau Pentium. Ultimele experimente au încercat să obţină eventuale creşteri de performanţe folosind aspecte de nivel coborât, în cazul nostru operaţiile de lucru cu memoria. Pentru F4 în mod particular, introducerea unor aspecte de nivel coborât mi a dus la scăderea timpilor totali pentru cele trei grafuri.

Capitolul 8

Concluzii şi subiecte de cercetări viitoare

Subiectul acestei teze aparţine domeniului foarte actual al proiectării automate a sistemelor numerice. În acest domeniu foarte generos, noi ne-am concentrat cercetarea asupra proiectării automate hardware/software a sistemelor numerice multiprocesor heterogene. În fine, particularizând şi mai mult, conţinutul tezei este axat pe două probleme centrale în proiectarea de sistem: partiţionarea hardware/software şi planificarea activităţilor sistemului.

Fiecare capitol al tezei se încheie cu prezentarea concluziilor noastre desprinse pentru capitolul respectiv. De aceea evităm ca în acest capitol să spunem încă o dată în plus ceea ce a fost deja spus într-un loc mult mai potrivit. Capitolul 8 sistematizează contribuţiile originale ale acestei teze şi scoate în faţă subiectele pentru cercetări viitoare, pe care noi în acest moment le considerăm interesante.

Capitolul 2 discută problema partiţionării hardware/software. Principalele contribuţii originale desprinse din acest capitol sunt:

- O abordare nouă a partiţionării automate hardware/software a specificărilor de sistem. Factorii pe care noi îi considerăm de bază în obţinerea unor implementări cu performanţe ridicate sunt: creşterea gradului de paralelism al sistemului şi minimizarea comunicărilor între hardware şi software.
- Metrici noi pentru analiza specificării şi estimarea costurilor hardware şi software. Controlarea procesului de partiţionare foloseşte o funcţie de cost, care înglobează aceste metrici.
- Definirea problemei partiţionării hardware/software ca şi o problemă de partiţionare a unui graf. Algoritmii experimentaţi sunt tabu search şi simulated annealing şi după cunoştinţele noastre, aceasta este prima cercetare ce foloseşte tabu search în partiţionare.
- Punerea la punct a două experimente care folosesc exemple din industrie: coprocesorul reţelei Ethernet şi blocul F4 al circuitului ATM.

Capitolul 3 adresează problema planificării activităţilor unui sistem, legate resurselor arhitecturii şi între care există dependenţe de date. Rezultatele originale conţinute în acest capitol sunt:

- Instanţierea mărimilor ASAP, ALAP şi drum critic pentru grafuri legate. Relaţiile noi pe care le propunem ţin cont de faptul că activităţile legate aceleiaşi resurse, mai puţin procesorul dedicat, trebuie secvenţializate.

- O relație nouă pentru limita inferioară a timpului de execuție a unui graf legat. Relația reface raționamentul folosit în stabilirea formulei "clasice" a lui Fernandez, dar folosește suplimentar și informația de legare a nodurilor pe resurse.
- O euristică nouă pentru planificarea grafurilor legate. Ea încurajează execuția nodurilor de pe drumul critic, redefinit ținând cont de maniera de legare a nodurilor pe resurse, și urmărește să crească gradul de concurență din implementarea finală.

Capitolul 4 definește formal un graf legat, cu dependențe de date și de control (GDC) și formulează formal problema planificării acestor grafuri. Rezultatele acestui capitol sunt următoarele:

- Reprezentarea unui GDC ca un cvint-uplu ce modelează activitățile sistemului, relațiile de precedență între acestea și structurile condiționale ale grafului.
- Definirea formală a planificării unui GDC legat. Ea prezintă constrângerile și setul de informații ce trebuie cunoscute în momentul în care un nod poate fi planificat pentru execuție.
- Un set de reguli pentru construirea GDC-ului atașat unei specificări de sistem.

Capitolul 5 prezintă soluția noastră pentru planificarea grafurilor legate, cu dependențe de date și de control. După cunoștințele noastre, această problemă nu a fost abordată anterior în literatura de specialitate. Principalele contribuții originale aduse de acest capitol sunt:

- Descrierea arhitecturii folosite pentru execuția grafurilor cu dependențe de date și de control. Prezentarea structurilor de date, inclusiv a organizării lor, folosite în planificarea execuției nodurilor unui GDC.
- Studiul modului în care condițiile din graf influențează planificarea activităților. Condițiile sunt grupate în lista de gărzi, care condiționează execuția unui nod, și lista de influență, care influențează momentul când o activitate poate fi lansată în execuție. Capitolul prezintă algoritmii pentru calculul listelor de gărzi și de influență ale nodurilor unui graf.
- Stabilirea constrângerilor legate de cunoașterea condițiilor pe parcursul planificării. Condițiile trebuie transmise resurselor, cel târziu la primul moment când ele trebuie folosite în planificarea unui nod. Capitolul formulează două strategii de comunicare a condițiilor în graf, astfel încât să se evite întârzierea activităților ce depinde de ele. Prima comunică condițiile după ce ele au fost calculate, cât se poate de repede pe prima magistrală găsită disponibilă. A doua metodă "întârzie" transmiterea până când o activitate ce folosește condiția în planificare devine pregătită pentru execuție.
- Formularea a două euristici de planificare a GDC-urilor. Euristica bazată pe liste de priorități caracterizează fiecare nod printr-o prioritate unică, corespunzătoare situației celei mai dezavantajoase în care el apare. Aceste priorități sunt folosite în selecția următorului nod planificat. Al doilea algoritm este cel de planificare prin ajustare și potrivire. El calculează fiecare trace posibil din graf, le planifică individual, după care le potrivește, lăsând trace-urile lungi nemodificate și adaptându-le lor pe cele scurte.
- Punerea la punct a unui set de experimente, menite să evidențieze calitatea soluțiilor găsite și complexitatea în timp a algoritmilor.

Capitolul 6 este orientat asupra fazelor târzii ale co-sintezei hardware/software. El descrie sinteza de nivel înalt a componentei hardware şi generarea codului C pentru procesele din software. Capitolul aduce următoarele contribuţii originale:

- Prezentarea principiului după care se poate genera cod C, folosind ca şi reprezentare intermediară reţele Petri. Sunt discutate regulile după care sunt produse structurile de date şi instrucţiunile echivalente semantic cu o reţea Petri.
- Sunt detaliate un set de optimizări pentru reducerea timpului de execuţie şi a dimensiunii codului produs. Aceste optimizări realizează împachetarea datelor, evaluarea expresiilor, eliminarea codului mort, expandarea apelurilor de funcţii, restructurarea codului după anumite pattern-uri şi reducerea numărului de comunicări interprocese.

Capitolul 7 încheie teza de doctorat. El aplică metodolgia şi instrumentele noastre de proiectare unui exemplu din industrie, blocul F4 al circuitului ATM. Acest capitol este gândit ca el însuşi să fie o concluzie, în sensul aplicabilităţii practice a cercetării noastre pentru exemple din industrie. El are un caracter foarte pragmatic, totuşi, din conţinutul său rezultă următoarele elemente generale, originale:

- Construirea unui model pentru o aplicaţie cu funcţionalitate complexă. Modelul este independent de arhitectură şi este organizat astfel încât să grupeze activităţi similare.
- Estimarea timpilor de execuţie pentru activităţile componente şi construirea GDC-ului său.
- Experimentarea a diferite soluţii arhitecturale şi variante de legare a activităţilor din sistem.

După cum am afirmat în mai multe rânduri pe parcursul tezei, noi considerăm rezultatele obţinute ca fiind doar "etape" în obţinerea unor algoritmi superiori de proiectare. Această muncă fiindu-ne foarte proaspătă în memorie, este greu să propunem acuma subiecte viitoare de cercetare, care să fie structural diferite de abordările acestei teze. Să sperăm că timpul le va aduce şi pe ele. Mai degrabă, în urma experimentării metodelor noastre, am observat aspecte perfectibile. Acestea le enumerăm în continuare.

Referitor la planificarea grafurilor legate, cu dependenţe de date:

- Introducerea de strategie pentru micşorarea influenţelor negative observate în planificarea cu CP/MISF-ML
- Definirea unei priorităţi asemănătoare forţei [PK89] pentru nodurile grafurilor legate
- Experimentarea altor euristici de planificare, de cât cele bazate pe liste.

Referitor la planificarea grafurilor legate, cu dependenţe de date şi de control:

- Experimentarea pentru noduri a altor priorităţi, de cât CP/MISF-ML.
- Găsirea altor strategii de comunicare a condiţiilor, eventual ataşându-le priorităţi asemeni celorlalte activităţi din graf.
- Ierarhizarea trace-urilor după alte criterii de cât lungimea. Foarte interesantă pare folosirea unui factor, care să reflecte influenţa asupra trace-ului curent a planificării trace-urilor mai importante.
- Tratarea ciclurilor

- Punerea la punct a unui algoritm de planificare optimă a GDC-urilor.

Pentru a întregii imaginea despre aplicabilitatea instrumentelor noastre de proiectare în situații practice, s-ar impune dezvoltarea și experimentarea altor exemple din industrie.

Instrumentele dezvoltate de noi trebuie completate cu componentele software pentru legarea activităților de resurse și sinteza arhitecturii pentru GDC-uri. Reunirea acestor instrumente într-un ciclu de proiectare coerent presupune însă realizarea unui mediu software pentru configurarea și gestiunea ciclului de proiectare.

Bibliografie

[AC'95] Hans Achatz, "Extended 0/1 LP Formulation for the Scheduling Problem in High Level Synthesis", în *Proceedings of the European Design Automation Conference*, Hamburg, IEEE CS Press, 1993, pp. 226-231

[AS83] G. R. Andrews, F. B. Schhneider, "Concepts and notations for concurrent programming", *ACM Computing Surveys*, 15, 1, 1983

[AS86] Alfred Aho, Ravi Sethi, Jeffrey Ullman, "Compilers. Principles, Techniques, and Tools", Addison-Wesley Publishing Company, 1986

[AS93] Peter Athanas, Harvey Silverman, "Processor Reconfiguration Through Instruction-Set Metamorphosis", *Computer*, March 1993, pp. 11-18

[AT95] Jay K. Adams, Donald E. Thomas, "Multiple-process behavioral synthesis for mixed hardware-software systems", în *Proceedings of the International Symposium on System Synthesis*, IEEE CS Press, 1995, pp.10-15

[AT96] Jay K. Adams, Donald E. Thomas, "The Design of Mixed Hardware/Software Systems", în *Proceedings of the 33-rd Design Automation Conference*, Las Vegas, NV, IEEE CS Press, 1996, pp. 515-520

[BC'96] Gaetano Boriello, Pai Chou, Ross Ortega, "Embedded system co-design towards portability and rapid integration", în *Hardware/Software Co-Design*, NATO ASI 1995, G. De Micheli şi M. G. Sami (editori), Kluwer Academic Publishers, 1996

[BE92] M. Berkelaar, "Unix manual page of `lp_solve`", Eindhoven University of Technology, Design Automation Section, 1992

[BE93] Bellcore, "Generic Requirements for Operations of Broadband Switching Systems", TANWT-001248 issue 2, October 1993

[BE95] Thomas Benner, Rolf Ernst, A. Osterling, "Scalable Performance Scheduling for Hardware-Software Cosynthesis", în *Proceedings of the European Design Automation Conference*, Brighton , IEEE CS Press, 1995, pp. 164-169

[BE96] Armin Bender, "Design of an Optimal Loosely Coupled Heterogeneous Multiprocessor System", în *Proceedings of the ED & TC*, Paris, March 1996

[BR94] E. Barros, Wolfgang Rosenstiel, X. Xiong, "A Method for Partitioning UNITY Language in hardware and software", în *Proceedings of the European Design Automation Conference EURO-DAC'VHDL*, IEEE CS Press, 1994, pp.220-225

[BS96] Klaus Buchenrieder, A. Sedlemeier, C. Veith, "Industrial hardware/software co-design", în *Hardware Software Co-Design*, NATO ASI 1995, G. De Micheli şi M. G. Sami (editori), Kluwer Academic Publishers, Boston , 1996

[CB94] Pai Chou, Gaetano Boriello, "Software scheduling in the co-synthesis of reactive real-time systems", în *Proceedings of the 31st DAC*, June 1994

[CG72] E. G. Coffman, R. L. Graham, "Optimal scheduling for two-processor systems", *Acta Informatica*, vol. 1, 1972, pp. 200-213

[CG94], Massimilliano Chiodo, Paolo Giusto, Attila Jurecska, Harry C. Hsieh, Alberto Sangiovanni-Vincentelli, Luciano Lavagno, "Hardware-Software Codesign of Embedded Systems", *IEEE Micro*, August 1994, pp. 26-36

[CL] Chen. Liu, "ATM Switching Systems",

[CO76] E. G. Coffman, "Computer and Job-Shop Scheduling Theory", New York: Wiley, 1976

[COB95] Pai Chou, Ross Ortega, Gaetano Boriello, "The Chinook Hardware/Software Co-Synthesis System", *în Proceedings of the 7th International Symposium on System Synthesis*, 1995

[CW94] Pai Chou, Elizabeth Walkup, Gaetano Boriello, "Scheduling for Reactive Real-Time Systems", *IEEE Micro*, August 1994, pp. 37-47

[DE96] Alexa Doboli, Petru Eleş,"Software Generation in a Hardware/Software Co-synthesis Environment", în *Buletinul Ştiinţific al Universităţii "Politehnica" din Timişoara*, Tomul 41 (55), 1996, pp. 185-195

[DH94] Alexa Doboli, Jonas Hallberg, Petru Eleş, "A Simulation Model for the Operation and Maintenance Functionality in ATM Switches", în *Proceedings of the International Conference on Technical Informatics CONTI'94*, 1994, vol. 4, pp. 31-40

[DO95] Alexa Doboli, "VHDL Based Hardware/Software Partitioning of Complex Systems", *Referatul 2 pentru doctorat*, Universitatea "Politehnica" Timişoara, 1995

[ED97] Petru Eleş, Alexa Doboli, Paul Pop, "Process Scheduling for Performance Estimation and Co-Synthesis of Embedded Systems", Raport tehnic al Departamentului de Calculatoare, Universitatea "Politehnica" Timişoara, 1997

[EF94] Marty Edwards, John Forrest, "A development environment for the cosynthesis of embedded software/hardware systems", *în Proceedings of the European Design Automation Conference EDAC*, IEEE CS Press, 1994, pp. 469-473

[EHB93] Rolf Ernst, Jorg Henkel, Thomas Benner, "Hardware-Software Cosynthesis for Microcontrollers", *IEEE Design & Test of Computers*, December 1993, pp. 64-75

[EK92] Petru Eleş, Krizysztof Kuchcinski, Zebo Peng, Marius Minea, "Compiling VHDL into a High-Level Synthesis Design Representation", în *Proceedings of the EURO-DAC/EURO-VHDL'92*, IEEE CS Press, 1992, pp.604-609

[EK94] Petru Eleş, Krzysztof Kuchcinski, Zebo Peng, Marius Minea, "Synthesis of VHDL Concurrent Processes", în *Proceedings of the EURO-DAC/EURO-VHDL'94*, IEEE CS Press, 1994, pp.540-545

[EP94] Petru Eleş, Zebo Peng, Alexa Doboli, "VHDL System-Level Specification and Partitioning in a Hardware/Software Co-synthesis Environment", în *Proceedings of the 3rd International Workshop on Hardware/Software Codesign*, IEEE CS Press, 1994, pp. 49-55

[EP96] Petru Eleş, Zebo Peng, Krzysztof Kuchcinski, Alexa Doboli, "Hardware-Software Partitioning with Iterative Improvement Heuristics", în *Proceedings of the International Symposium on System Synthesis*, IEEE CS Press, 1996

[EP97] Petru Eleş, Zebo Peng, Krzysztof Kuchcinski, Alexa Doboli, "System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search", în *Design Automation for Embedded Systems*, 2, Kluwer Academic Publishers, 1997, pp.5-32

[FB73] Eduardo B. Fernandez, Bertram Bussel, "Bounds on the Number of Processors and Time for Multiprocessor Optimal Schedules", *IEEE Transactions on Computers*, vol. c-22, no. 8, August 1973, pp. 745-751

[GJ79] M. R. Garey, D. S. Johnson, "Computers and Intractability: a Guide to the Theory of NP-Completness", W. H. Freeman and Company, 1979

[GD92] Daniel Gajski, Nikil Dutt, A. Wu, Stan Liu, "High Level Synthesis, Introduction to Chip and System Design", Kluwer Academic Publishers, 1992

[GM93] Rajeh K. Gupta, Giovanni De Micheli, "Hardware-Software Cosynthesis for Digital Systems", *IEEE Design & Test of Computers*, September 1993, pp. 29-41

[GM94] Rajeh K. Gupta, Giovanni De Micheli, "Program Implementation Schemes for Hardware-Software Systems", *COMPUTER*, January 1994, pp. 50-55

[GO79] Michael J. Gordon, "The Denotational Description of Programming Languages-An Introduction", Springer Verlag, 1979
[GR95] Peter Grun, Translatarea programelor VHDL în C, într-un mediu pentru co-sinteza hardware/software, Lucrare de dizertație, Universitatea "Politehnica" din Timișoara, 1995
[GT93] F. Glover, E. Taillard, D. de Werra, "A users guide to tabu search", *Annals of Operators Research*, 41, 1993, pp. 3-28
[GV95] Daniel D. Gajski, Frank Vahid, "Specification and Design of Embedded Hardware - Software Systems", *IEEE Design & Test of Computers*, Spring 1995, pp. 53-67
[HU61] T. C. Hu, "Parallel sequencing and assembly line problem", *Operational Research*, vol. 9, November 1961, pp. 841-848
[IEEE88] IEEE, IEEE Standard VHDL Language Reference Manual, 1988
[IJ95] Terek Ben Ismail, Ahmed A. Jerraya, "Synthesis steps and design models for codesign", *Computer*, February 1995, pp. 44-52
[in80] intel, "Component Data Catalog", 1980
[in90] intel, "i486™ Processor Programmer's Reference Manual", Mc Graw-Hill, 1990
[JE92] K. Jensen, Coloured Petri Nets, Springer Verlag, 1992
[KA96] Yu-Kwong Kwok, Ishfaq Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 5, Maz 1996, pp. 506-521
[KG83] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, "Optimization by Simulated Annealing", *Science* 220 (4598), 1983, pp. 671-680
[KL70] B. W. Kernighan, S. Lin, "An efficient heuristic procedure for partitioning Graphs", *Bell System Technical Journal*, 49 (2), 1970, pp. 291-307
[KL93] Asawaree Kalavade, Edward Lee, "A Hardware-Software Codesign Methodology for DSP Applications", *IEEE Design & Test of Computers*, September 1993, pp. 16-28
[KL94] Asawaree Kalavade, Edward Lee, "A Global Criticality/ Local Phase Driven Algorithm for the Constrained Hardware/Software Partitioning Problem", în *Proceedings of the 3rd International Workshop on Hardware/Software Codesign*, IEEE CS Press, 1994, pp. 42-48
[KN84] Hironori Kasahara, Seinosuke Narita, "Practical Multiprocessor Scheduling Algorithms for Eficient Parallel Processing", *IEEE Transactions on Computers*, vol. c-33, no 11, November 1984, pp. 1023-1029
[KP96] Michael J. Knieser, Christos A. Papachristou, "COMET: A Hardware-Software Codesign Methodology", în *Proceedings of the European Design Automation Conference*, 1996, pp. 178-183
[KO74] W.H. Kohler, "Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems", *Journal of Association of Computer Machineries*, vol. 21, January 1974, pp. 140-156
[KS75] W.H. Kohler, K. Steiglitz, "Exact, approximate, and accuracy algorithms for the flow-shop problem n/2/F/F", *Journal of Association of Computer Machineries*, vol. 22, January 1975, pp. 106-114
[LA83] Stephen Lavenberg, "Computer Performance Modeling Handbook", Academic Press, 1983
[LL93] M. Larsson, M. Ljungberg, J. Rooth, "The ATM Switch Concept and the ATM Pipe Switch", *Ericsson Review*, No.1, 1993

[LM88] Peter D. Lawrence, Konrad Mauch, "Real-Time Microcomputer System Design: an Introduction", McGraw-Hill, 1988
[LM95] Yau-Tsun Steven Li, Sharad Malik, "Performance Analysis of Embedded Software Using Implicit Path Enumeration", în *Proceedings of the 32nd Design Automation Conference*, 1995, pp. 456-461
[LW82] J. Y. T. Leung, J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks", *Performance Evaluation*, 2, 1982
[MA91] Matra, Memories, Matra MHS, 1991
[MB95] Peter Marwedel, Steven Bashford, Rainer Domer, Birger Landwehr, Ingolf Markhof, "A Technique for Avoiding Isomorphic Netlists in Arhitectural Synthesis", *Technical Report #95-28*, University of California, Irvine, 1995
[MI90] Giovanni De Micheli, "The Olympus Synthesis System", *IEEE Design and Test of Computers*, vol. 7, no.5, October 1990, pp. 37-53
[MI93] Marius Minea, "A VHDL Compiler for a High Level Synthesis System", Technical Report LiTH-IDA-R-93-23, Department of Computer and Information Science, Linkoping University, June 1993
[MI94] Giovanni De Micheli, "Computer-Aided Hardware-Software Codesign", *IEEE Micro*, August 1994, pp. 10-16
[MI94b] Giovanni De Micheli, "Synthesis and optimization of Digital Circuits", McGraw-Hill, 1994
[MO90] Mayez A. Al-Mouhamed, "Lower Bound on the Number of Processors and Time for Scheduling Precedence Graphs with Communication Costs", *IEEE Transactions on Software Engineering*, vol. 16, no. 12, December 1990, pp. 1390-1401
[NM96] Ralf Niemann, Peter Marwedel, "Hardware/Software Partitioning using Integer Programming", în *Proceedings of the ED & TC Conference*, IEEE CS Press, March 1996, pp.473-479
[OH94] Kunle Olukotun, Rahid Helaihel, Jeremy Levitt, Ricardo Ramirez, "A Software-Hardware Cosynthesis Approach to Digital System Simulation", *IEEE Micro*, August 1994, pp. 48-58
[RE93] Collin Reeves, "Modern Heuristic Techniques for Combinatorial Problems", Blackwell Scientific Publications, 1993
[PE87] Zebo Peng, "A Formal Methodology for Automated Synthesis of VLSI Systems", teză de doctorat no.170, Department of Computer and Information Science, Linkoping University, 1987
[PE92] Zebo Peng, "Digital System Simulation with VHDL in a High-Level Synthesis System", *Microprocessing and Microprogramming, the EUROMICRO Journal*, 34 (1-5), 1992, pp. 263-269
[PK89] Pierre G. Paulin, John P. Knighht, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", *IEEE Transactions on Computer-Aided Design*, vol.8, no.6, June 1989, pp.661-679
[PK93] Zebo Peng, Krzysztof Kuchcinski, "An algorithm for partitioning of application specific systems", în *Proceedings of the European Design Automation Conference EDAC*, IEEE CS Press, 1993, pp. 316-321
[PK94] Zebo Peng, Krzysztof Kuchcinski, "Automated transformation of algorithms into register-transfer level implementation", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13 (2), February 1994, pp. 150-166
[PP92] Shiv Prakash, Alice Parker, "SOS: Synthesis of Application-Specific Heterogeneous Multiprocessor Systems", *Journal of Parallel and Distributed Computing*, 16, 1992, pp. 338-351

[PR91] M. de Prycker, "Asynchronous Transfer Mode", Ellis Horwood, 1991
[RC94] R. Rooholamini, V. Cherkassy, M. Garver, "Finding the Right ATM Switch for the Market", *Computer*, April 1994
[SA89] V. Sarkar, "Partitioning and Scheduling Parallel Programs for Multiprocessors", Cambridge, Massachussets, MIT Press, 1989
[SC88] David Schmidt, "Denotational Semantics-A Methodology for Language Development", Wm.C. Brown Publishers, 1988
[SE76] Ravi Sethi, "Scheduling Graphs on Two Processors", *SIAM Journal Computing, vol.5*, nr.1, March 1976, pp. 73-82
[SP94] Erik Stoy, Zebo Peng, "A design representation for hardware/software co-synthesis", în Proceedings of the Euromicro Conference, IEEE CS Press, 1994, pp. 192-199
[SV93] P. Staxen, C. Vestin, "The Telecom Evolution in the Broadband Era", *Ericsson Review*, 1993
[SY93] Synthesia, "The SYNT/MINT VHDL Design System", Doc. No: SYNTHESIA 93 011 A, 1993
[SV94] Bengt Svantesson, "A Testbench for the F4/F5 Block of the ATM Switch", Master Thesis, Royal Technical University Stockholm, 1994
[SW88] R. Sommerhalder, S. C. van Westrhenen, "The Theory of Computability, Programs, Machine Effectiveness and Feasibility", Addison-Wesley Publishing Company, 1988
[TAS93] Donald E. Thomas, Jay K. Adams, Herman Schmit, "A Model and Methodology for Hardware-Software Codesign", IEEE Design & Test of Computers, September 1993, pp. 6-15
[VG94] Frank Vahid, Jie Gong, Daniel D. Gajski, "A Binary-Constraint Search Algorithm for Minimizing Hardware durinmmg Hardware/Software Partitioning", în Proceedings of the European Design Automation Conference, IEEE Computer Science Press, 1994, pp. 214-219
[VN95] Frank Vahid, Sanjiv Narayan, Daniel D. Gajski, "SpecCharts: A VHDL Front-End for Embedded Systems", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 14, no. 6, June 1995, pp. 694-706
[WB94] Elizabeth Walkup, Gaetano Boriello, "Interface timing verification withh application to synthesis", în Proceedings of the 31st DAC, June 1994
[WG90] M. Y. Wu, Daniel D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems", IEEE Transactions on Parallel and Distributed Systems, vol.1, no. 3, July 1990, pp. 330-343
[YC95] C. W. Yeh, C. K. Cheng, T. T. Y. Lin, "Optimization by Iterative Improvement An Experimental Evaluation on Two-Way Partitioning", IEEE Transaction on CADICS, vol. 14, no. 2, February 1995, pp. 145-153
[YE93] W. Ye, "Fast Timing Analysis for Hardware-Software Cosynthesis", în Proceedings of the International Conference on Computer Design, IEEE CS Press, 1993, pp. 452-457
[YW95] Ti-Yen Yen, Wayne Wolf, "Sensitivity-Driven Co-Synthesis of Distributed Embedded Systems", în Proceedings of the 8th Symposium on System Synthesis, Cannes, 1995, pp. 4-9
[YW96] Ti-Yen Yen, Wayne Wolf, "Performance Estimation for Real-Time Distributed Embedded Systems", submitted to IEEE Transactions on Parallel and Distributed Systems, 1996