Calculatoare ne-universale: implementarea programelor în hardware

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

mai 2001

Subiect: sintetizarea automată de calculatoare specifice unei aplicaţii

Cunoştinţe necesare: cunoştinţe elementare despre arhitectura calculatoarelor şi limbaje de programare, elemente de C

Cuvinte cheie: hardware, hardware reconfigurabil, compilare, sinteza de circuite

Cuprins

1 Un excurs istoric **1**

2 Modelul spaţial de calcul **2**
2.1 Hardware reconfigurabil . . . 3
2.2 Limbaje de descriere a hardware-ului (HDL) . . . 3
2.3 Hardware din limbaje de nivel înalt . . . 4
2.3.1 O instrucţiune . . . 4
2.3.2 Un bloc de instrucţiuni . . . 4
2.3.3 Instrucţiuni de control . . . 4
2.3.4 Accese la memorie . . . 5
2.3.5 Bucle . . . 6
2.4 Apeluri de procedură . . . 7

3 Rezultate preliminare **7**
3.1 Un program desfăşurat . . . 7
3.2 Modelul computaţional . . . 8
3.3 Duplicarea codului . . . 9
3.4 Performanţă . . . 9
3.5 Cercetări viitoare . . . 9

4 Alte surse de informaţie **10**

1 Un excurs istoric

Primele calculatoare construite erau foarte diferite de cele de astăzi. Nu mă refer aici la dimensiuni şi capacitate (care depind doar de tehnologie), ci la structura lor fundamentală. Acele calculatoare erau construite pentru a rezolva o *singură problemă*; nu erau universale. Ele constau dintr-o colecţie de unităţi funcţionale, care puteau face calcule simple. "Programatorii" aveau sarcina de a conecta unităţile funcţionale între ele cu sîrme, pe care le înfigeau manual în tot felul de mufe. De exemplu, dacă vroiau să calculeze $(a+b)^2$, programatorii luau o unitate care făcea adunări şi una care făcea înmulţiri şi le cuplau ca în figura 1.

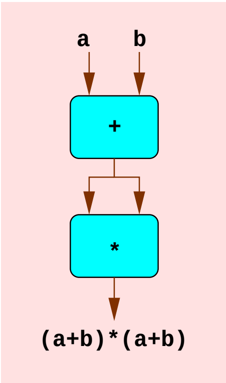

Figura 1: Implementarea în hardware a expresiei $(a+b)^2$. Variabilele devin sîrme, iar operaţiile sunt executate de unităţi funcţionale.

În anii '40 în domeniul calculatoarelor intră în scenă un genial matematician, pe nume John von Neumann. von Neumann analizează starea de fapt a calculatoarelor şi scrie în 1945 un raport intitulat "First Draft of a Report on the EDVAC" (Prima ciornă a unui raport despre EDVAC), în care sugerează o arhitectura revoluţionară. În această arhitectură, programul nu mai este reprezentat de felul în care sunt cuplate unităţile funcţionale, ci este stocat în memorie, fiind descris folosind un limbaj numit cod-maşină. În cod-maşină, operaţiile de executat sunt codificate sub forma unor numere numite *instrucţiuni*. Programul de executat este descris printr-un şir de instrucţiuni, care se execută consecutiv.

Pe lîngă unităţile funcţionale care fac operaţii aritmetice, calculatorul mai are o unitate de control, care citeşte instrucţiunile programului şi care trimite semnale între unităţile funcţionale pentru a executa aceste instrucţiuni. Rezultatele intermediare sunt stocate în memorie. Această arhitectură se numeşte "von Neumann". Dacă nu sunteţi impresionaţi de această viziune radicală, este pentru că marea majoritate a calculatoarelor din ziua de azi sunt bazate pe această arhitectură; noţiunea de limbaj-maşină, şi cea înrudită, de limbaj de programare, folosite pentru descrierea programelor, sunt concepte foarte naturale pentru toţi cei care manipulează calculatoarele.

Se cuvine menţionat faptul că ideea de a descrie un program folosind un limbaj (şi nu prin conexiuni între unităţi funcţionale) este mai veche; în 1936 Alan Turing folosise noţiunea de "maşină Turing universală" pentru a descrie un calculator universal, care poate executa orice program. Programele erau stocate în memoria calculatorului, reprezentate ca şiruri de numere.

2 Modelul spaţial de calcul

Spuneam că marea majoritate a calculatoarelor digitale construite astăzi au o arhitectură von Neumann. Dar nu toate; există încă "calculatoare" bazate pe arhitectura veche, în care unităţile funcţionale sunt legate între ele prin sîrme şi nu există un program stocat în memorie.

Modelul von Neumann are o virtute excepţională: flexibilitatea. Folosind acelaşi hardware putem executa oricîte programe, toate diferite între ele. Preţul care trebuie însă plătit este performanţa: unele operaţiuni nu se pot descrie foarte eficient prin instrucţiunile disponibile, şi s-ar putea implementa mult mai bine direct în hardware; figura 2 prezintă un astfel de exemplu extrem.

```
for (i=0; i < 8*sizeof(a); i++) {
    b |= a & 1;
    b <<= 1;
    a >>= 1;
}
```

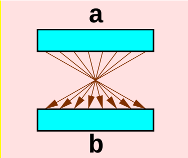

Figura 2: Implementarea unui program care pune biţii dintr-un octet în ordine inversă, în software (în limbajul C) şi respectiv hardware. Execuţia în software necesită cîteva zeci de instrucţiuni, pe cînd cea în hardware se face practic instantaneu.

Ca atare, atunci cînd este nevoie de performanţă deosebită, sau cînd flexibilitatea nu este foarte importantă, inginerii construiesc în continuare calculatoare specializate. De exemplu, ruterele de mare viteză de pe coloana vertebrală a Internetului trebuie să proceseze pachetele cu informaţii la viteze uluitoare (au la dispoziţie doar cîteva zeci de nano-secunde pentru fiecare pachet). Ca atare, unele din funcţiile de clasificare şi manipulare ale pachetului sunt implementate direct în hardware, într-un algoritm fixat. Chiar dacă performanţa nu este critică, atunci cînd algoritmul de procesare este foarte bine standardizat, este adesea mai ieftin de folosit un astfel circuit specializat, numit ASIC (Application-Specific Integrated Circuit). Chiar şi într-un calculator obişnuit se găsesc de obicei multe astfel de circuite pentru manipularea perifericelor: plăci de reţea, controlere de magistrală, etc.

2.1 Hardware reconfigurabil

În articolul meu anterior din NetReport am discutat despre o posibilă evoluţie a tehnologiilor electronice, nanotehnologiile, care ne vor permite să construim circuite digitale cu miliarde de porţi logice. În acel articol am sugerat că aceste circuite imense vor putea fi folosite cel mai eficient sub formă de hardware reconfigurabil. Hardware-ul reconfigurabil este un model de calcul foarte asemănător cu cel de acum cincizeci de ani: constă dintr-o "mare" de porţi logice universale, care pot fi programate individual pentru a implementa orice funcţie logică, şi două reţele: una care leagă porţile între ele, şi care poate fi configurată în felurite moduri pentru a lega şi dezlega porţile, şi una de configuraţie, care este folosită pentru a transporta informaţiile de configurare spre porţi şi spre cealaltă reţea.

Folosind hardware reconfigurabil putem sintetiza circuite specifice fiecărei aplicaţii; de data asta, însă, în loc de a conecta unităţile funcţionale manual, le cuplăm în mod electronic, schimbînd felul în care sunt cuplate porţile logice.

În articolul de faţă voi arăta discuta mai pe larg cum se poate transforma un program într-un circuit şi ce fel de proprietăţi are acest nou model de execuţie.

2.2 Limbaje de descriere a hardware-ului (HDL)

Ideea de a descrie hardware folosind programe nu este de loc nouă; de fapt circuitele integrate digitale sunt construite chiar în acest fel: circuitul de construit este descris într-un limbaj special de programare, numit Hardware Description Language. Circuitul astfel descris este apoi prelucrat de o suită complicată de programe CAD (Computer-Aided Design), care optimizează circuitul, îl transformă în operaţii elementare, le plasează pe o suprafaţă plană şi le conectează prin sîrme. În faza de proiectare, circuitul descris într-un HDL poate fi *simulat*, pentru testare şi depanare.

La ora actuală există două limbaje foarte importante de descriere a hardware-ului, care au fiecare susţinătorii lor aprigi; unul dintre ele se numeşte VHDL iar celălalt Verilog. VHDL este prescurtarea a două prescurtări: Vhsic HDL, unde VHSIC înseamnă "Very High Speed Integrated Circuits". Verilog vine de la "VERifying LOGic", dar limbajul este folosit nu numai pentru a simula şi verifica circuite logice (cum a fost conceput iniţial, şi după cum arată numele său), ci şi pentru a le proiecta şi implementa (*sinteza de hardware*; hardware synthesis).

Cele două limbaje sunt relativ similare ca putere de expresie, dar incompatibile între ele. Sunt de asemenea destul de diferite de celelalte limbaje obişnuite de programare: în limbajele HDL programatorul exprimă un circuit ca o colecţie de sub-circuite care operează în paralel (paralelismul este explicit în program). Variabilele sunt semnale electrice, iar operaţiile descriu unităţi funcţionale. Nu există funcţii recursive, structuri de date complicate sau manipulare dinamică a memoriei (`malloc/new/free`).

2.3 Hardware din limbaje de nivel înalt

Verilog şi VHDL sunt foarte potrivite pentru a descrie circuite, însă sunt puţin potrivite pentru felul în care gîndesc oamenii. În general, oamenii au probleme în a-şi imagina mai multe procese care se petrec concurent şi comunică între ele; din cauza asta calculatoarele paralele nu au avut prea mult succes, şi tot din cauza asta programarea aplicaţiilor cu mai multe fire de execuţie (threads) este un exerciţiu de disciplină şi nervi de fier.

Ca atare, de multă vreme unele grupuri de cercetare încearcă să folosească limbaje mai simple şi mai naturale (de genul Pascal/C) pentru a descrie hardware. (Dacă limbajele acestea nu vi se par naturale, înseamnă că nu aţi încercat să descrieţi un sistem complex în Verilog!).

Tentativa de a genera hardware din limbaje obişnuite de programare se loveşte de mai multe obstacole. În primul rînd, tehnologia pentru extragerea paralelismului dintr-un program scris într-un limbaj "secvenţial" (ca C sau Fortran) este nesatisfăcătoare; ori paralelismul este principalul avantaj al unui model de execuţie în hardware.

În al doilea rînd, limbajele de genul C sau Fortran folosesc tot felul de construcţii care nu au un corespondent direct în hardware, cum ar fi funcţii recursive, memorie alocată dinamic, sau structuri de date complicate. În ce fel se pot traduce acestea în hardware într-un mod eficient este un subiect activ de cercetare.

În cele ce urmează voi ilustra cum se pot traduce în hardware operaţiile fundamentale ale unui limbaj de programare.

2.3.1 O instrucţiune

Pentru a traduce o operaţiune simplă în hardware, asociem variabilele cu "sîrmele", şi operaţiile cu unităţi funcţionale, după cum am anticipat şi mai devreme. Figura 1 ilustrează o astfel de transformare.

2.3.2 Un bloc de instrucţiuni

În limbajele de nivel înalt valorile sunt stocate în variabile; în codul maşină ele sunt atribuite unor regiştri, sau sunt stocate în memorie. Cînd avem de-a face cu un bloc de instrucţiuni consecutive care manipulează doar regiştri, le putem traduce în hardware transformînd fiecare registru într-o sîrmă, ca în figura 3.

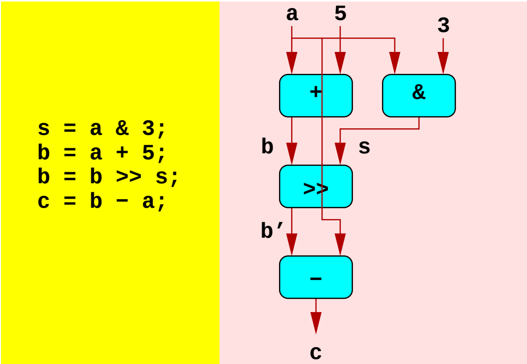

Figura 3: Un bloc de instrucţiuni şi circuitul corespunzător. Observaţi cum variabilele devin sîrme. Variabila `b` este folosită de două ori, deci devine două sîrme diferite, `b` şi `b'`. Observaţi de asemenea cum operaţiile `&` şi `+` sunt executate simultan.

2.3.3 Instrucţiuni de control

În limbajele de nivel înalt avem construcţii care selectează între diferite instrucţiuni care urmează să se execute, cum ar fi `if-then-else`. Pentru a le traduce pe acestea în hardware avem mai multe alternative. În figura 4 ilustrăm o posibilitate care exploatează la maximum paralelismul dintre instrucţiuni, executînd simultan părţile `then` şi `else`, şi selectînd doar rezultatele necesare.

Acest gen de execuţie se numeşte *speculativă*, pentru că execută mai mult decît este strict necesar şi selectează numai ceea ce e necesar la sfîrşit. Şi microprocesoarele moderne folosesc o formă de execuţie speculativă, pe care am descris-o mai demult într-un articol din PC Report despre predicţia salturilor.

2.3.4 Accese la memorie

Una dintre cele mai puternice trăsături ale limbajelor de nivel înalt este capacitatea de a manipula matrici, care se bazează pe posibilitatea de a adresa memoria în mod *indirect*. Cu alte cuvinte, citim conţinutul unei celule de memorie a cărei adresă a fost calculată.

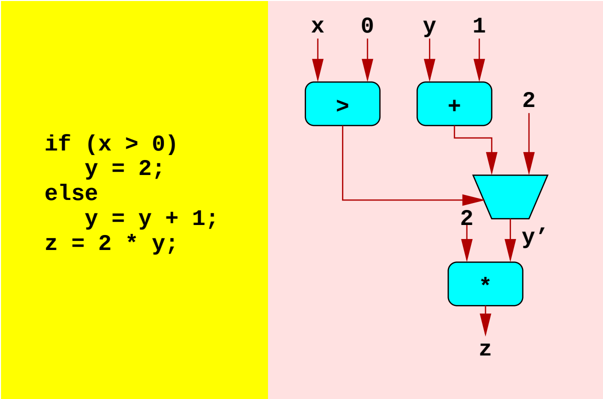

Figura 4: Implementarea instrucţiunilor de control al execuţiei în mod speculativ: executăm simultan ramurile `if`, `then` şi `else` şi păstrăm numai rezultatul corect. Trapezul este un multiplexor, cu trei intrări: un selector, desenat în lateral, şi două intrări între care se selectează. Ieşirea multiplexorului este intrarea din stînga dacă selectorul este 0 şi intrarea din dreapta dacă selectorul este 1.

Accesele indirecte nu pot fi transformate pur şi simplu în sîrme; trebuie să folosim memorii hardware pentru a stoca matrici. În plus, atunci cînd avem cod executat în mod speculativ, trebuie să fim atenţi să nu executăm decît accesele la memorie corecte, pentru că altfel vom "strica" conţinutul memoriei.

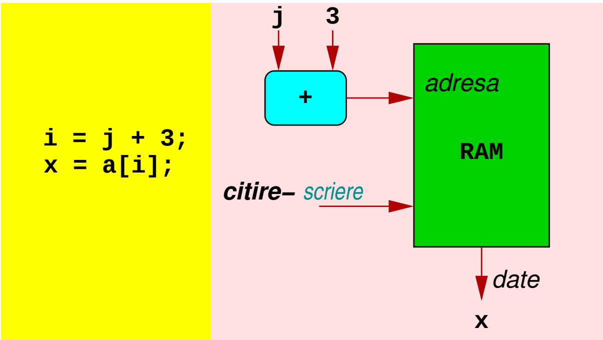

Figura 5: Pentru a implementa matrici folosim memorii în care adresăm cu indicele din matrice. Citirea din matrice este o citire din memorie, iar scrierea în matrice este o scriere în memorie.

Figura 5 ilustrează implementarea unei bucăţi de cod care conţine accese la memorie.

2.3.5 Bucle

Lucrurile devin mai interesante atunci cînd avem de executat bucle. Circuitul pe care-l implementăm nu mai este un simplu circuit *combinatorial*, în care semnalul se propagă într-un singur sens. El devine un circuit *secvenţial*, cu o buclă de feed-back. Bucla de feed-back trebuie să fie întreruptă de un registru controlat de un ceas, care sincronizează propagarea tuturor semnalelor din buclă.

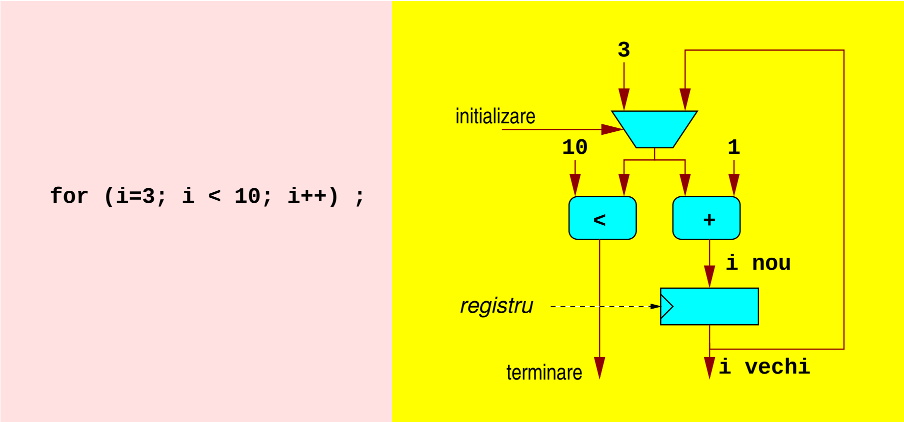

Figura 6: Implementarea (uşor simplificată) a unei bucle. Un *registru* citeşte noua valoare a lui `i` la fiecare tact de ceas şi o memorează pînă la următorul tact. Bucla este pornită în execuţie punînd semnalul de iniţializare pe 0 (după aceea semnalul trebuie să rămînă 1 pînă la terminarea buclei). Semnalul de terminare a buclei indică momentul cînd toate iteraţiile s-au terminat.

Figura 6 ilustrează o posibilă implementare a unei bucăţi de cod care conţine o buclă.

2.4 Apeluri de procedură

Cînd avem un program compus din mai multe proceduri, fiecare procedură poate fi implementată ca un circuit separat. Procedura curentă, în curs de execuţie, corespunde circuitului activ. Controlul este transmis între proceduri prin semnale care activează procedura chemată.

O problemă ceva mai complicată este că atunci cînd procedura chemată se termină, trebuie să sară la o bucată de cod diferită de fiecare dată (la chemătorul ei). Acest lucru poate fi realizat în mai multe feluri; de exemplu, fiecare procedură primeşte un parametru suplimentar, care este chemătorul său; pentru terminarea procedurii, se poate sintetiza o bucată de hardware care selectează dintre toţi chemătorii posibili bazat pe valoarea acestui parametru şi-i trimite un mesaj cînd procedura curentă şi-a terminat execuţia.

Această implementare nu funcţionează pentru procedurile recursive, care mai au nevoie şi de o stivă; deocamdată nu avem la-ndemînă o soluţie eficientă pentru a trata astfel de cazuri.

3 Rezultate preliminare

Pentru a estima impactul unui asemenea model de calcul asupra performanţei am făcut un studiu preliminar, care face o mulţime de asumpţii simplificatoare. Sperăm că modelul real de calcul nu va da rezultate substanţial diferite de idealizarea noastră.

Cel mai "idealizat" aspect al analizei noastre constă în cantitatea de informaţie pe care am folosit-o despre program şi zonele de memorie pe care le foloseşte: am presupus că ştim exact ce adrese de memorie vor fi accesate de fiecare instrucţiune de citire şi scriere din memorie. Această informaţie a fost colectată executînd programul modificat pentru a colecta informaţii despre propria sa comportare (un astfel de program se numeşte *instrumentat*).

3.1 Un program desfăşurat

Figura 7 arată un astfel de program desfăşurat şi aşezat în plan.

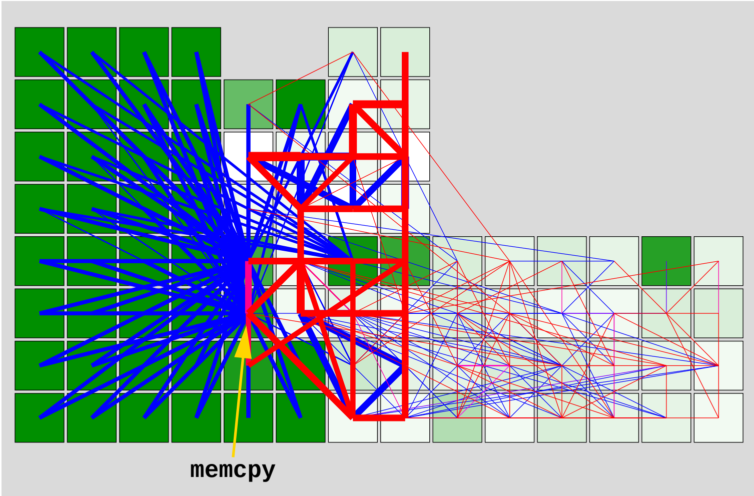

Figura 7: Un program aşezat în plan. Fiecare pătrat reprezintă 100 de unităţi, unde o instrucţiune aritmetică simplă sau un cuvînt de memorie ocupă 1 unitate. Un pătrat verde conţine numai memorie, iar un pătrat alb conţine numai instrucţiuni. Nuanţe intermediare indică mixturi de instrucţiuni şi memorie. Liniile roşii indică transfer de control între blocuri consecutive de instrucţiuni, iar liniile albastre indică accese la memorie. Grosimea unei linii este proporţională cu logaritmul frecvenţei cu care acea linie este utilizată. De exemplu, o linie de 10 "pixeli" a fost utilizată de program de $2^{10} = 1024$ de ori. Liniile groase sunt deci mult mai importante decît cele subţiri.

Fiecare program este reprezentat sub forma unui *graf*, în care nodurile sunt instrucţiuni sau memorii şi în care muchiile sunt transfer de execuţie între instrucţiuni sau accese la memorie. Toate programele pe care le-am analizat[1] au trăsături foarte similare:

- Numărul de muchii este proporţional cu cel de noduri (constanta de proporţionalitate este sub 10)[2].
- Gradul unui nod (numărul de vecini) urmează o distribuţie foarte inegală: cîteva noduri au foarte multe muchii.
- În particular, există un număr foarte mic de noduri care "atinge" aproape toată memoria folosită de program. În figură sunt două astfel de pătrate, din care radiază foarte multe muchii albastre. Numim astfel de noduri "stele".

Analizînd graful din figură am descoperit că una dintre cele două stele este în corpul funcţiei `memcpy`. Dacă ne gîndim puţin, vom realiza că acest lucru este foarte natural: toate structurile de date ale programului care sunt copiate vor fi atinse de această funcţie.

[1]Am analizat o serie de programe reprezentative scrise în limbajul C.
[2]Numărul de muchii într-un graf poate fi pătratic în cel de noduri.

3.2 Modelul computaţional

Dacă am luat un program şi l-am "întins" pe o suprafaţă, întrebarea următoare este: "cît de repede merge"? Pentru a putea răspunde, trebuie să precizăm alte cîteva lucruri despre modelul nostru computaţional:

- Am transformat fiecare bloc de instrucţiuni aflate între două instrucţiuni de salt (jump/branch) consecutive într-un circuit; (nu am folosit deci metoda descrisă mai sus pentru a genera cod speculativ: un `if` va executa numai ramura corectă).
- Am presupus că în interiorul fiecărui bloc toate instrucţiunile independente se vor executa în paralel.
- Am presupus că timpul de transmisiune a unui semnal de-a lungul unei muchii este proporţional cu lungimea muchiei.
- Am presupus că un bloc este pornit în execuţie numai după ce blocul anterior a terminat complet de calculat, şi că mesajul prin care execuţia este pornită este trimis împreună cu datele de la intrarea blocului.
- Nu am făcut nici un fel de optimizări specifice generării de hardware (de exemplu, un circuit care adună cu valoarea constantă 1 poate fi făcut mult mai simplu şi mai rapid decît un circuit general de adunare).

3.3 Duplicarea codului

Pornind de la acest model am evaluat performanţa programelor. Rezultatele au fost amestecate. Am căutat apoi metode de a îmbunătăţi performanţa prin transformări simple.

Primul obiectiv asupra căruia ne-am îndreptat atenţia au fost "stelele"; acestea sunt detrimentale performanţei, pentru că, avînd mulţi vecini, unii dintre aceştia vor fi în mod necesar plasaţi la distanţă mare de centru, deci vor fi accesaţi foarte greu.

Ne-am întrebat: nu am putea cumva să "spargem" o stea în bucăţi, în aşa fel încît să facem două stele mai mici? Răspunsul este: da, şi chiar foarte simplu. Tot ce trebuie făcut pentru programul din figură este să duplicăm codul funcţiei `memcpy` şi să aranjăm ca din locuri diferite să fie chemate copii diferite. O tehnică foarte simplă pentru a realiza acest lucru se numeşte în compilatoare "inlining", în care funcţia chemată este inserată textual în codul chemătorului.

După ce am aplicat această transformare, am obţinut graful din figura 8.

3.4 Performanţă

Rezultatele au fost destul de interesante. Am comparat performanţa acestor programe cu execuţia lor pe un microprocesor Alpha 21164 la 500 Mhz. Am presupus că în implementarea spaţială vom putea executa o operaţie pe ciclu de ceas şi că vom putea folosi aceeaşi frecvenţă de ceas cu Alpha.

În medie programele noastre funcţionează ceva mai puţin de 2 ori mai încet decît pe Alpha, dar unele dintre ele merg mai rapid, după cum se vede în figura 9, în stînga. Graficul din dreapta arată cum îşi petrece timpul fiecare din programe.

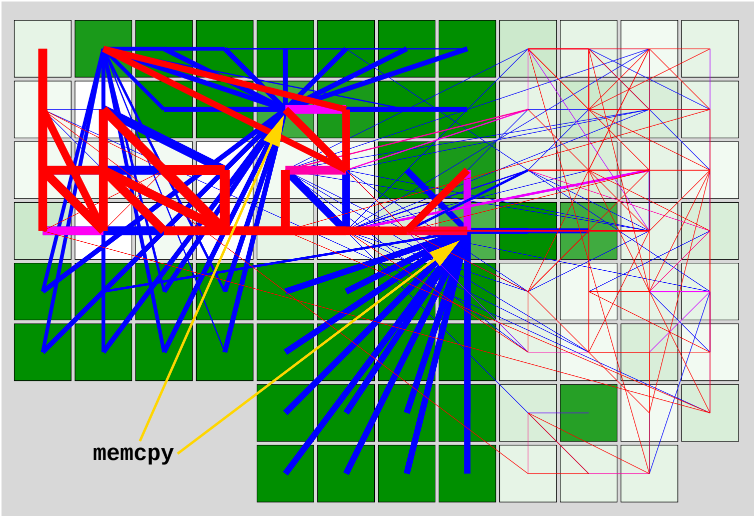

Figura 8: Programul din figura 7 după ce corpul funcţiei `memcpy` a fost duplicat prin *inlining*. Observaţi cum stelele sunt mult mai mici şi mai bine plasate.

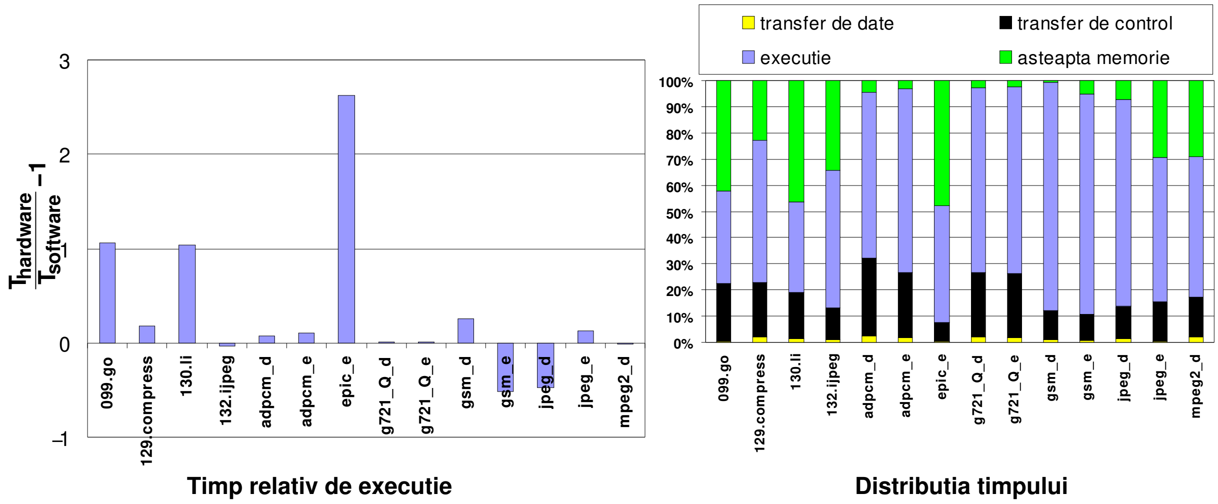

Figura 9: **În stînga** este raportul dintre timpul de execuţie al programului executat pe un microprocesor Alpha şi pe un model spaţial de calcul. Valoarea zero indică egalitatea timpului de execuţie. O valoare negativă indică execuţie mai rapidă în hardware, pe cînd o valoare pozitivă indică execuţie mai rapidă pe Alpha. **În dreapta** este ilustrat timpul de execuţie al fiecărui program. Culoarea verde indică timp petrecut aşteptînd ca citiri din memorii de la distanţă să returneze rezultatul. Albastru este pentru timp în care se execută calcule utile. Maro arată timpul petrecut transferînd control între blocuri consecutive, şi galben măsoară timpul de transfer al datelor între blocuri consecutive. Porţiunile verde şi maro pot fi probabil reduse prin optimizări mai agresive, cum ar fi folosirea de cache-uri şi respectiv execuţie speculativă.

3.5 Cercetări viitoare

Acest text ilustrează cercetările noastre preliminare în domeniul modelului spaţial de execuţie; foarte multe lucruri mai rămîn de făcut. Rezultatele noastre sunt însă încurajatoare: avem o metodologie care ne permite să compilăm programe destul de complicate iar performanţa nu este substanţial degradată faţă de modelele de calcul convenţionale.

Va trebui însă să dezvoltăm o grămadă de tehnici noi, plecînd de la construcţia hardware-ului reconfigurabil însuşi, trecînd prin metode de compilare şi scule de simulare mai sofisticate, care iau în considerare mai multe detalii despre arhitectura reală a unui astfel de sistem.

4 Alte surse de informaţie

- Pagina Alan Turing: `http://www.turing.org.uk/turing`.
- O scurtă introducere în VHDL: `http://www.eej.ulst.ac.uk/tutor/vhdnotes.html`.
- Un site despre Verilog: `http://www.verilog.com`.
- Articolul din conferinţa ISCA 2001 care prezintă rezultatele descrise în acest articol este disponibil la `http://www.cs.cmu.edu/~mihaib/research/isca01.ps`.

  Articole ale mele anterioare din PC Report:

  - august 1999, despre predicţia salturilor `http://www.cs.cmu.edu/ mihaib/articles/complex/articles.html#predictie`.
  - luna decembrie 1999, despre teoria complexităţii şi maşina Turing universală: `http://www.cs.cmu.edu/ mihaib/articles/complex/articles.html#complex`.
  - luna mai 2001, despre nanotehnologie: `http://www.cs.cmu.edu/~mihaib/articles/articles.html#nano`.
  - iulie 1998, despre hardware reconfigurabil: `http://www.cs.cmu.edu/~mihaib/articles/articles.html#reconf`.