UNIVERSITATEA “LUCIAN BLAGA” DIN SIBIU
FACULTATEA DE INGINERIE
DEPARTAMENTUL DE CALCULATOARE ŞI INGINERIE ELECTRICĂ

PROIECT DE DIPLOMĂ

Conducator ştiinţific : conf. dr. ing. Adrian Florea
Îndrumător: conf. dr. ing. Adrian Florea

Absolvent:
Klein Andrei Florin
Specializarea Calculatoare

- Sibiu, 2013 –

UNIVERSITATEA “LUCIAN BLAGA” DIN SIBIU
FACULTATEA DE INGINERIE
DEPARTAMENTUL DE CALCULATOARE ŞI INGINERIE ELECTRICĂ

Cadru distribuit de simulare pentru optimizarea microarhitecturilor de calcul prin metode euristice

Conducător ştiinţific: conf. dr. ing. Adrian Florea
Îndrumător: conf. dr. ing. Adrian Florea

Absolvent:
Klein Andrei Florin
Specializarea Calculatoare

Cuprins

1. Introducere

1.1. Problema optimizării unei arhitecturi de microprocesor. Complexitatea

Optimizarea unei arhitecturi de calcul pornește de la simularea caracteristicilor de performanță ale acestei arhitecturi folosind un instrument software numit simulator. Simulatoarele sunt elemente esențiale în cercetare, ele fiind folosite pentru a studia comportamentul unei microarhitecturi fără a fi necesară implementarea și construcția fizică a acesteia [6]. Acest avantaj principal al utilizării simulatoarelor permite o flexibilitate mult mai mare în proiectarea arhitecturilor noi cât și în îmbunătățirea arhitecturilor existente deoarece timpul necesar pentru dezvoltarea unui simulator software este mult mai scurt decât cel necesar pentru a construi un microprocesor real bazat pe microarhitectura studiată. Alte avantaje ale utilizării simulatoarelor sunt costurile mai mici aferente dezvoltării și ușurința depanării erorilor și problemelor apărute (*debugging*). Utilizarea simulatoarelor în cadrul procesului de proiectare și îmbunătățire al microarhitecturiilor moderne a devenit astfel un element esențial [1].

Simulatoarele permit astfel observarea mult mai rapidă a efectelor oricărei modificări aduse unei microarhitecturi, cum ar fi schimbări in performanța obținută, temperatura maximă atinsă de procesor sau puterea consumată. Cu toate acestea timpul de execuție al unei simulări nu este neglijabil. În funcție de metodologia de simulare, de parametrii microarhitecturali, de obiectivele urmărite (metrici), complexitatea simulatorului necesar variază. Astfel în funcție de această complexitate și de acuratețea simulatorului, cât și de numărul de instrucțiuni al benchmark-urilor folosite acest timp poate ajunge de la doar câteva secunde la ore sau în unele cazuri chiar la zile.

De exemplu, o microarhitectură de procesor superscalar modernă poate avea zeci de parametri, cum ar fi dimensiunile memoriilor cache de instrucțiuni și date de nivele 1, 2 și 3, dimensiunile buffer-elor de instrucțiuni și date, fetch rate, issue rate și numărul unităților de execuție (centralizat sau nu), structura predictoarelor de salturi implementate, numărul de intrări în buffer-ele de reordonare și redenumire, etc. Numărul de configurații posibile poate ajunge chiar și pentru arhitecturi simple la zeci sau sute de miliarde. Astfel, dacă dorim să simulăm toate configurațiile posibile ale unei arhitecturi foarte simple cu doar 10 parametri

(**Fetch Rate, Instruction Buffer Size, Data Buffer Size, Issue Rate Max, Instruction Cache Size, Data Cache Size, Memory Latency, Reorder Buffer Size, Rename Buffer Size, Compiler Optimizations**), fiecare cu 4 valori distincte, rezultă numărul total al configurațiilor egal cu:

$$4^{10} = 2^{20} = 1048576$$

Dacă timpul necesar simulării unei configurații este de 1 oră, atunci durata căutarii exhaustive ar fi aproximativ:

$$1048576 \div 24 \div 365 = 119{,}7\ ani$$

Pentru a identifica configurația cea mai bună (optimul global) dintr-un anumit punct de vedere, cum ar fi performanța, este necesară o căutare exhaustivă asupra întregului spațiu definit de parametrii arhitecturali. Având în vedere durata execuției unei simulări și numărul foarte mare de configurații posibile, chiar și în cele mai bune cazuri în care se folosește o microarhitectură reală ar fi necesari mii de ani pentru o astfel de căutare folosind un singur procesor. Chiar dacă s-ar folosi un sistem distribuit pentru a executa simulări, timpul necesar este tot prea mare, deci o căutare exhaustivă este practic imposibilă. Problema găsirii unei configurații optime se transformă astfel într-o problemă de optimizare cu scopul de a găsi o configurație cât mai bună într-un timp de așteptare rezonabil.

În acest scop se aplică atât simularea microarhitecturală cât și tehnici de optimizare. Unele metode de optimizare care au aplicabilitate în problemă căutării celei mai bune configurații (folosind unul sau mai multe criterii pentru a determina această configurație) sunt:

- Metode euristice care caută în spațiul definit de parametrii arhitecturali, parametrii compilatorului și de caracteristicile aplicațiilor folosind criterii multiple pentru a determina configurația cea mai bună (performanța procesorului, consumul de putere, căldura disipată, numărul de nuclee ale procesorului) [3, 4]
- Algoritmi de căutare locală, în vecinătatea configurației microarhitecturale curente: Hill Climbing (determinist), Simulated Annealing (stohastic) [7]
- Metode avansate Machine Learning, algoritmi genetici, tehnici bio-inspirate bazate pe metode Particle Swarm Optimization, algoritmi stigmergici bazați pe mișcările coloniilor de furnici

Această lucrare încearcă să ofere un răspuns modest provocării reprezentate de căutare automată a configurației optime prin dezvoltarea instrumentului ***FOCAP*** („*Framework for Optimizing Computer Architecture Performance"*), un cadru de simulare și optimizare a performanței pentru microarhitecturi de calcul. În acest scop vor fi aplicați algoritmi de optimizare bazați pe metode de căutare euristică care eșantionează în diferite moduri spațiul de proiectare și simulează un număr mult mai mic de configurații arhitecturale. Algoritmii folosiți în acest caz sunt:

- Hill Climbing: un algoritm determinist asemănător în funcționare cu tehnica de programare *greedy*
- Simulated Annealing: un algoritm probabilistic inspirat de procesul de călire al metalelor prin răcire treptată („*annealing"*).
- Algoritmi genetici: un tip de algoritmi asemănători evoluției naturale care folosesc conceptele de "selecție naturală" și "moștenire genetică", noțiunile de generație și procesele de selecție („*selection"*), mutație („*mutation"*), încrucișare („*crossover"*) sau recombinare („*recombination"*) precum și supraviețuirea urmașilor în generația următoare "*survival of the fittest*".

Acești algoritmi de optimizare sunt adaptați pentru un cadru de simulare distribuită a microarhitecturilor parametrizabile și implementați în FOCAP, realizând o căutare automată în spațiul de proiectare microarhitectural. Totodată, aplicația dezvoltată are un caracter flexibil și modular permițând implementarea și altor algoritmi de optimizare folosind funcțiile puse la dispoziție de celelalte module fără ca funcționarea acestora să fie afectată și fără să fie nevoie să le fie aduse schimbări.

1.2. Obiectivele lucrării

Obiectivul principal al acestei lucrări a fost implementarea unui cadru flexibil pentru optimizarea unei microarhitecturi care să ruleze eficient pe sisteme multiprocesor sau multicore într-un mediu distribuit de tip LAN ("local area network") sau HPC ("high performance computing").

Pentru a putea exploata eficient puterea de calcul a sistemelor moderne care dispun de procesoare cu mai multe nuclee și sunt capabile să execute în paralel multiple fire de execuție

este necesar ca aplicația să poată folosi oricâte fire de execuție pentru a rula simulările mari consumatoare de timp. Având în vedere că multe simulatoare existente care ar putea fi folosite sunt aplicații care folosesc doar un singur fir de execuție, aplicația este construită în așa fel încât execuția paralelă nu depinde de simulator. O soluție simplă și eficientă la această problemă este execuția în paralel a simulatorului folosind o instanță pentru fiecare fir de execuție oferit de procesorul sau procesoarele sistemului.

Exploatarea eficientă a procesoarelor multicore totuși nu este suficientă pentru a asigura rezultate într-un timp cât mai scurt. Pentru a crește performanța sistemului și a reduce timpul necesar procesului de optimizare este necesară implementarea și a unui sistem de distribuție care să permită ca simulările să ruleze pe clienți multipli aflați într-o rețea și conectați la un server. Un astfel de sistem poate folosi mai multe calculatoare, fiecare echipat cu unul sau mai multe procesoare multicore, pentru a rula în paralel un număr mare de simulări și pentru a reduce semnificativ timpul necesar obținerii unui rezultat.

Obiectivul cel mai important pentru funcționarea cadrului este implementarea algoritmilor de optimizare și adaptarea lor atât la simularea microarhitecturilor cât și la execuția distribuită și paralelă a acestora. Algoritmii aleși pentru a fi implementați sunt Hill Climbing, Simulated Annealing și un algoritm genetic.

Pentru a crește performanța cadrului de optimizare a fost implementat și un modul de bază de date pentru a stoca rezultatele simulărilor deja rulate. Baza de date este folosită asemănător unei memorii cache, rezultatele sunt direct disponibile, simularea configurațiilor corespondente nemafiind necesară. Pentru accesarea eficientă a bazei de date a fost implementată și o metodă de compresie a parametrilor configurației. Această compresie transformă o configurație într-un număr întreg unic reprezentat pe 64 biți. Un alt avantaj al compresiei este faptul că tabela nu trebuie editată în cazul în care setul de parametri se schimbă.

1.3. Structura lucrării

Pe scurt, această lucrare este structurată astfel:

În capitolul 2 este prezentată o cercetare bibliografică ce vizează stadiul actual în domeniul abordat, optimizarea euristică a arhitecturilor de calcul. În acest capitol este descris conceptul și procesul care stă la baza explorării automate a spațiului de proiectare (*„Automatic Design Space Exploration”*) și algoritmii euristici folosiți. Algoritmii descriși sunt atât de tip determinist, cum este Hill Climbing, cât și euristici și stohastici, cum sunt Simulated Annealing, algoritmii stigmergici, algoritmii bioinspirați (*particle swarm optimization*) și algoritmii genetici. De asemenea sunt descriși algoritmi genetici de optimizare mono-obiectiv, dar și moduri de abordare multi-obiectiv (agregare, sortare lexicografică, bazate pe fronturi Pareto). Tot în acest capitol este prezentată arhitectura client-server cât și modul de funcționare multifir al cadrului de optimizare.

Capitolul 3 descrie și explică detaliile și alegerile făcute pentru implementarea FOCAP. Pentru început este prezentată structura distribuită a cadrului de optimizare și se ilustrează modul de interacțiune dintre cele 4 componente majore: aplicația server, aplicația client, interfața grafică și baza de date. Sunt de asemenea prezentate protocoalele de comunicare pe rețea folosite: protocolul de comunicare server-client pentru distribuția simulărilor, respectiv protocolul de comunicare dintre interfața grafică și server. Urmează descrierea modului de funcționare al clientului și a arhitecturii software a acestuia cât și prezentarea parametrilor configurabili ai microarhitecturii utilizate. Partea server a cadrului de optimizare conține mecanismul de distribuție al simulărilor, modulul care implementează funcționalitatea de tip cache pentru simulări folosind o bază de date și algoritmii de optimizare implementați în așa fel încât să exploateze eficient resursele de calcul paralele oferite de clienți prin intermediul mecanismului de distribuție automată.

În capitolul 4 sunt prezentate rezultatele obținute în urma procesului de simulare și optimizare. Pentru fiecare dintre cei 3 algoritmi de optimizare implementati este prezentat un set de rezultate separat, astfel incat diferențele pot fi observate și studiate.

Capitolul 5 conține concluziile la care am ajuns în timpul dezvoltării lucrării de față, dar și planurile pentru dezvoltarea ulterioară și pentru îmbunătățirea cadrului de optimizare.

2. Stadiul actual în domeniul algoritmilor de optimizare

2.1. Soluții. Automatic Design Space Exploration

În primul capitol am prezentat problema optimizării microarhitecturilor și complexitatea acesteia care o face inabordabilă pentru o căutare exhaustivă a configurației optime din punct de vedere al performanței. Se impun astfel soluții bazate pe metode euristice (deterministe și probabilistice) care încearcă să obțină o configurație cât mai apropiată de cea optimă și care nu explorează decât o mică parte din întregul spațiu de proiectare [3].

Automatic Design Space Exploration (explorarea automată a spațiului de proiectare) este o metodă de optimizare care presupune eșantionarea a unei mici părți a spațiului de explorare în căutarea configurației optime sau a unei configurații cât mai apropiată de cea optimă din punctul de vedere al performanței. Procesul de eșantionare este ghidat de algoritmul de optimizare folosit și depinde în întregime de acesta. În consecință și calitatea rezultatelor obținute cât și timpul necesar execuției sunt dependente atât de algoritmul folosit cât și de parametrii de control ai acestui algoritm [3, 5, 10]. Rezultatele în cazul algoritmului Hill Climbing sunt influențate și de configurația inițială cu care este pornit algoritmul datorită modului determinist de funcționare al acestuia

Metoda Automatic Design Space Exploration este generală și poate fi aplicată oricărei probleme de optimizare în care căutarea exhausitvă nu este posibilă practic, cum ar fi Air Traffic Control, probleme de scheduling (problema comisului voiajor), Pattern Recognition, VLSI Circuit Layout, etc. În cazul optimizării microarhitecturilor superscalare pentru performanță cât mai bună se folosesc benchmark-uri executate folosind un simulator pentru arhitectura țintă [7, 9, 10]. Pentru optimizare a performanței se urmărește fie creșterea IPC, a numărului mediu de instrucțiuni executate într-un singur ciclu procesor, fie scăderea CPI, a numărului mediu de ciclii procesor necesari pentru execuția unei instrucțiuni, fie scăderea numărului total de ciclii procesor necesari pentru execuția programelor benchmark folosind simulatorul. Toți acești parametri sunt indicatori echivalenți ai performanței, respectiv ai

vitezei de procesare obținută cu o microarhitectură, cu condiția ca programul sau programele benchmark folosite să fie aceleași la fiecare rulare. De asemenea este important ca parametrii simulărilor, cum ar fi numărul maxim de instrucțiuni executate, să nu fie modificați.

2.2. Arhitectura client-server. Simulare multithreaded distribuită

Modelul arhitectural client-server reprezintă o structură distribuită formată din mai multe dispozitive hardware (sisteme de calcul) sau din mai multe programe (software) care împarte sarcini între sistemele care oferă accesul la resurse, numite *server*, și sisteme care accesează resurse prin intermediul unor cereri, numite *client*. Sistemele client adresează cereri sistemelor server, care așteaptă aceste cereri.

Aplicațiile client respectiv server pot rula pe același sistem de calcul, dar adesea se află pe mașini diferite, iar comunicarea are loc prin rețea. Rețelele client-server sunt centralizate, un singur server având capaciatatea de a deservi mai mulți clienți conectați. Topologia logică a unei rețele client-server este topologia stea. Datorită acestei structuri funcționarea și fiabilitatea server-ului sunt esențiale pentru întregul sistem, dacă apare o defecțiune sau o problemă la server atunci toți clienții dependenți de resursele care erau partajate nu vor mai putea funcționa. În cele mai multe cazuri totuși, chiar și în ceea ce privește simularea și optimizarea distribuită, dacă apare o defecțiune la un client restul sistemului continuă să funcționeze. Clienții de asemenea se pot conecta și deconecta, sau pot chiar să apară clienți cu totul noi, în timp ce rețeaua client-server execută deja o anumită sarcină.

Pentru a putea comunica sistemele client-server au nevoie de un protocol comun de comunicare care să fie întotdeauna respectat atât de server cât și de client. Protocoalele implementate pentru arhitecturi client-server implică transmiterea unor solicitări de la client, urmând ca server-ul să ofere apoi un mesaj răspuns. Aceste protocoale sunt implementate la nivelul aplicației în cadrul programelor server și client.

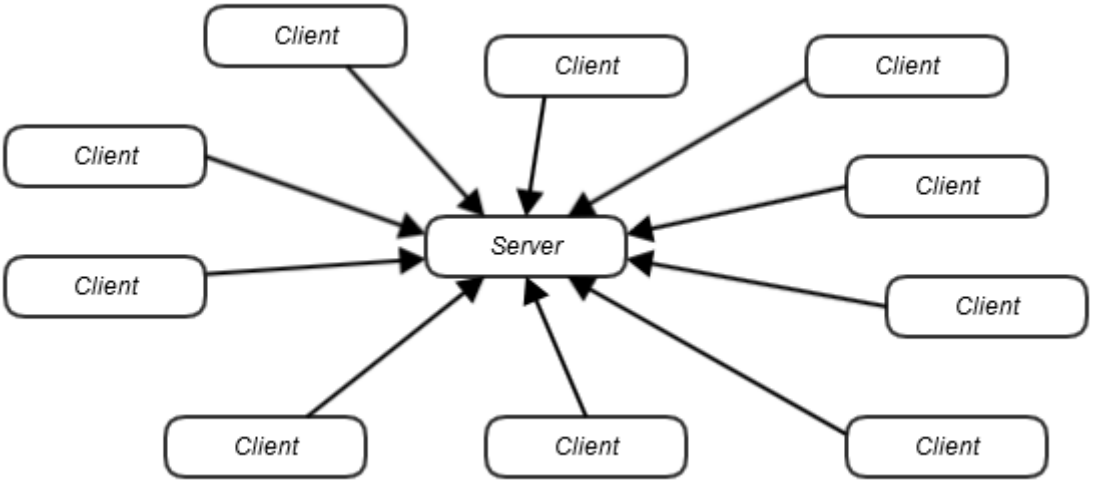

Figura 2.1 Arhitectura client-server

În ultimii ani procesoarele au urmat un alt curs evolutiv decât cel bine cunoscut care urmărea creșterea performanțelor prin îmbunătățirea vitezei de procesoare a unui singur fir de execuție. Au apărut procesoare cu mai multe nuclee (*„multicore”*), capabile să execute mai multe fire în paralel. Un program care folosește un singur fir de execuție nu va putea beneficia de puterea de procesare pusă la dispoziție de o microarhitectură modernă, deoarece va utiliza doar un singur nucleu. Pentru a exploata capacitatea de execuție paralelă a procesoarelor moderne este necesar un nou mod de abordare al programării, bazat pe dezvoltarea de aplicații care să fie capabile să folosească mai multe fire de execuție pentru a spori performanța. În cazul aplicațiilor vechi care nu sunt scrise pentru execuție paralelă există totuși posibilitatea lansării simultane în execuție a mai multor instanțe ale programului, fiecare urmând să folosească un singur nucleu al procesorului.

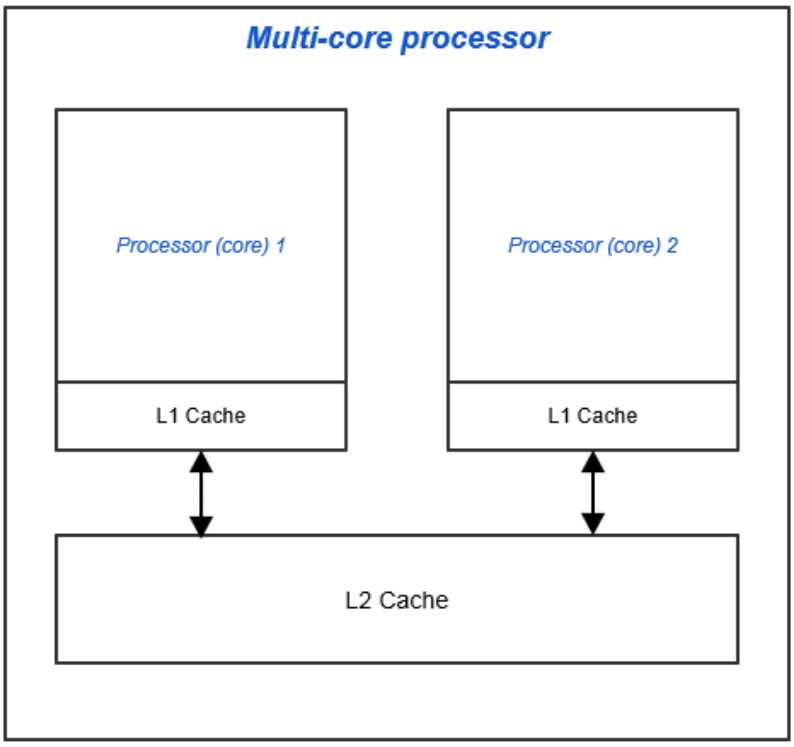

Figura 2.2 Diagrama simplificată a unui procesor multi-core

Aplicate împreună arhitectura client-server și programarea multifir permit implementarea de software scalabil care suportă atât utilizarea unui număr mare de sisteme de calcul cât și utilizarea procesoarelor multicore pentru a crește semnificativ performanța prin reducerea substanțială a timpului de execuție comparativ cu rularea sarcinii de lucru folosind un singur sistem și un singur nucleu.

2.3. Algoritmi de optimizare. Clasificare

Optimizarea este selecția celui mai bun element dintr-o mulțime de elemente alternative disponibile. În cazul cel mai simplu optimizarea reprezintă maximizarea sau minimizarea unei funcții reale alegând valori din domeniul de definiție al acesteia și calculând valoarea corespunzătoare.

Fie funcția de o singură variabilă:

$f : D \rightarrow \mathbb{R}$, definită pe mulțimea D cu valori în mulțimea numerelor reale

Se numește *problemă de optimizare* căutarea unui element x_0 astfel încât:

$$(\forall)x \in D, f(x) \leq f(x_0) \quad \text{(maximizare)}$$

sau

$$(\forall)x \in D, f(x) \geq f(x_0) \quad \text{(minimizare)}$$

Funcția f se numește *funcție obiectiv* sau *funcție cost*, elementul x_0 fiind valoarea optimă globală căutată.

În cazul optimizării arhitecturilor de calcul domeniul *D* este format din toate combinațiile posibile de valori ale parametrilor microarhitecturali. Se caută o configurație optimă fie din punctul de vedere al unui singur obiectiv, cum ar fi performanța, fie urmărind mai multe obiective (performanță, consum de putere, temperatură maximă, etc.). Având în vedere imposibilitatea căutării exhaustive se aplică algoritmi de căutare euristică din subdomeniul Machine Learning al inteligenței artificiale pentru a găsi o configurație cât mai apropiată de cea optimă. Algoritmii genetici fac parte din categoria algoritmilor de calcul evolutiv și sunt inspirați de teoria lui Darwin asupra evoluției, folosind conceptele de selecție naturală și moștenire genetică.

În funcție de modul de funcționare al acestor algoritmi, pot fi împărțiți în două categorii distincte [13]:

- Algoritmi determiniști: algoritmi care vor avea de fiecare dată același rezultat și vor fi executați identic, atâta timp cât condițiile inițiale (cum ar fi configurația microarhitecturală de început) sunt la fel. Acești algoritmi nu conțin elemente aleatoare.
    - Hill Climbing: algoritm de căutare locală asemănător cu tehnica de programare greedy
- Algoritmi probabilistici (stohastici): algoritmi care utilizează valori aleatoare în ideea de a obține performanță mai bună în medie. Acești algoritmi nu vor avea aceleași rezultate și nu vor urma aceeași cale de execuție de fiecare dată chiar dacă datele de intrare sunt identice.
    - Simulated Annealing: alt algoritm de căutare locală, inspirat din procesul de călire a metalelor prin răcire treptată
    - Algoritmi genetici: algoritmi inspirați de procesul evoluției biologice cu evaluare mono-obiectiv sau multi-obiectiv

- Algoritmi bioinspirați de tip Particle Swarm Optimization (stoluri de păsări, bancuri de pești, roiuri de albine)
- Algoritmi stigmergici: algoritmi bazați pe studiul mișcărilor coloniilor de furnici

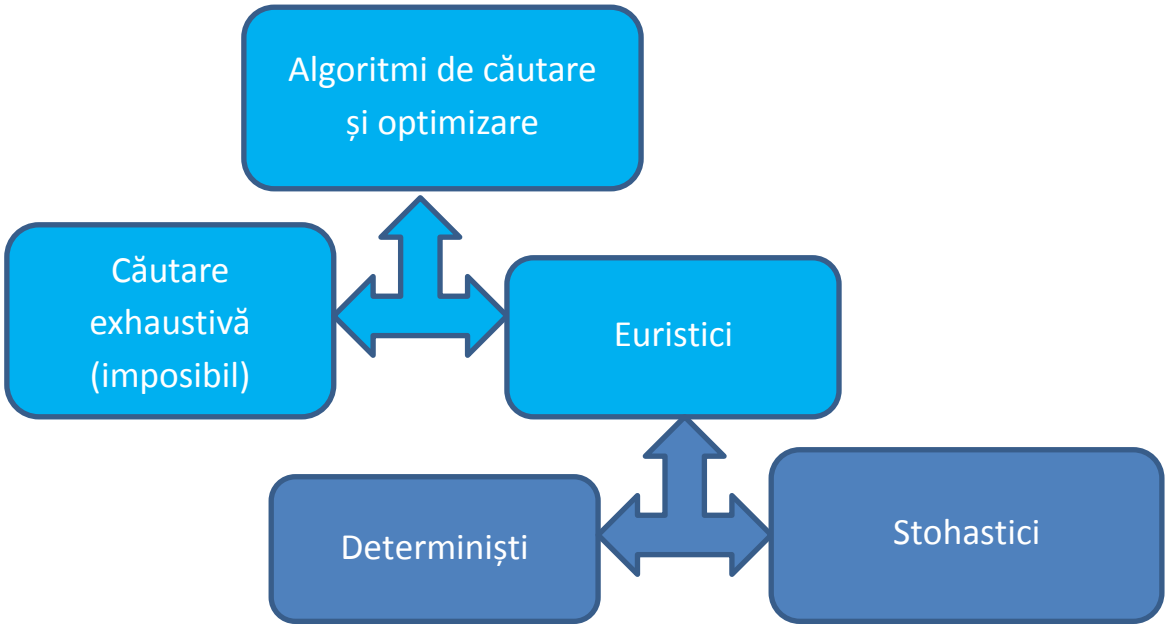

Figura 2.3 Clasificare a algoritmilor de optimizare

2.3.1. Hill Climbing. Pseudocod. Limitări

Hill Climbing este o metodă de optimizare care face parte din categoria algoritmilor de căutare locală. Începând de la o soluție oarecare a unei probleme Hill Climbing încearcă în fiecare iterație să modifice succesiv fiecare parametru al configurației curente urmând ca cea mai bună configurație curentă să fie aleasă pentru iterația următoare. Acest proces este repetat până când nu mai pot fi aduse îmbunătățiri configurației.

Algoritmul Hill Climbing încearcă să maximizeze sau să minimizeze o funcție țintă $f(X)$, unde X este un vector de valori continue sau discrete:

$$X = (x_1, x_2, x_3, \dots, x_n), \qquad \text{unde:}$$

$x_1, x_2, \dots, x_n$ sunt valori ale parametrilor *1...n* ai soluției

Pornind de la o soluție cu vectorul *X* inițializat aleator în fiecare iterație se modifică succesiv valorile $x_1, x_2, \dots, x_n$ și se evaluează soluțiile nou obținute. Cea mai bună dintre aceste soluții este memorată și utilizată în iterația următoare a algoritmului. Procesul se încheie în momentul în care nu se poate obține o soluție mai bună decât cea curentă, aceasta fiind un optim local al funcției. În cazul în care algoritmul este întrerupt înainte să fie îndeplinită condiția de terminare soluția curentă este una atât corectă cât și mai bună decât cea inițială, dar nu un optim local.

Hill Climbing este unul dintre cei mai simpli algoritmi de optimizare și prezintă un dezavantaj semnificativ. Datorită modului său de funcționare care se aseamănă cu tehnica de programare greedy algoritmul nu explorează decât o vecinătate în jurul configurației de start, iar soluția găsită când execuția se sfârșește nu va fi decât un optim local. Algoritmul nu poate trece de la un optim local la o soluție mai slabă, chiar dacă această tranziție ar putea conduce la găsirea unei soluții mai bune decât acest optim. Din același motiv în cazul în care algoritmul întâlnește un platou al funcției sau o zonă în care panta este prea mică pentru a distinge îmbunătățiri în soluție algoritmul se va opri fără să parcurgă acest platou.

Pentru a obține rezultate mai bune algoritmul Hill Climbing poate fi rulat de mai multe ori cu vectori soluție inițiali aleatori diferiți. Acest lucru permite explorarea spațiului de proiectare eliminând problema principală, anume explorarea limitată doar la vecinătatea soluției inițiale în direcția unui optim local.

Unele avantaje ale algoritmului Hill Climbing sunt simplitatea implementării și timpul mai scurt necesar execuției comparativ cu unele metode mai avansate cum sunt algoritmii genetici.

Algoritmul Hill Climbing în varianta care maximizează rezultatul este descris în pseudocod în modul următor:

```
HillClimbing
      solution = initial_solution;
      repeat
            V = neighbours(solution);
            max_val = -inf;
```

```
next_solution = NULL;
for each x in V do
      if evaluate(x) > max_val
            next_solution = x;
            max_val = evaluate(x);
if max_val <= evaluate(solution)
      return solution;
```

Figura 2.4 Algoritmul Hill Climbing în pseudocod

2.3.2. Simulated Annealing

Simulated Annealing este o metodă stohastică folosită pentru a căuta minimul sau maximul global al unei funcții într-un spațiu mare care conține mai multe minime sau maxime locale. Numele metodei („*annealing”*: călire) provine de la tehnica folosită în metalurgie care implică încălzirea unui metal până la o temperatură mare urmând ca acesta să fie răcit treptat și controlat (în funcție de metal). Acest proces întărește metalul și elimină defectele. În cazul optimizării algoritmul este asemănător cu procesul fizic, folosind un parametru numit *temperatură* care pornește de la o valoare inițială mare și scade treptat în fiecare iterație. Această valoarea controlează probabilitatea cu care soluțiile mai puțin bune decât cea curentă sunt acceptate.

Un exemplu clasic de problemă de optimizare pentru rezolvarea căreia se poate aplica Simulated Annealing este *problema comisului voiajor*. Acestă problemă presupune căutarea unui traseu optim, adică fie de lungime, fie de cost minim dacă acesta nu este direct proporțional cu distanța parcursă, pentru un comis voiajor care trebuie să viziteze *N* orașe o singură dată (*ciclu hamiltonian*) și apoi să se întoarcă de unde a plecat. O soluție pentru această problemă este o permutare a orașelor, numărul total de soluții posibile fiind deci *N!*. În cazul în care numărul de orașe este mic este posibilă căutarea exhaustivă pentru a rezolva problema. Dacă numărul orașelor este mare, o căutare exhaustivă ar dura prea mult datorită numărului de soluții care crește exponențial. Simulated Annealing poate fi aplicat pentru a optimiza o soluție, acesta aducând modificări mici soluției curente pentru a explora spațiul (spre exemplu, interschimbarea poziției a două orașe în soluție).

Scopul algoritmului Simulated Annealing este de a găsi o soluție cu *energie* minimă pornind de la o soluție inițială oarecare, în unele cazuri chiar aleatoare, evaluată la o anumită energie inițială. La fiecare pas Simulated Annealing evaluează una sau mai multe soluții vecine ale soluției curente alese după o anumită regulă sau aleator. În cazul în care o soluție vecină este mai bună decât cea curentă, algoritmul trece la această soluție. Dacă soluția nouă nu este mai bună decât cea curentă aceasta aceasta poate totuși să fie acceptată cu o anumită probabilitate. Această probabilitate depinde atât de temperatura curentă (cu cât temperatura este mai mare cu atât probabilitatea de acceptare este mai mare) cât și de diferența dintre energia soluției noi și cea a soluției curente (cu cât această diferență este mai mare cu atât probabilitatea de acceptare este mai mică). Simulated Annealing continuă până când una dintre condițiile de terminare este atinsă:

- O soluție care este considerată îndeajuns de bună a fost deja găsită, deci continuarea execuției algoritmului nu este necesară
- După un anumit număr predefinit de iterații nu a fost găsită o soluție mai bună decât cea curentă
- Un număr maxim de iterații a fost atins
- Timpul alocat pentru execuția algoritmului a expirat, soluția curentă sau cea mai bună soluție întâlnită este returnată

2.3.2.1. Soluții vecine în reprezentarea parametrilor microarhitecturali

O soluție vecină este o soluție obținută în urma modificării soluției curente într-un anume mod bine definit. Modul în care soluțiile vecine sunt obținute depinde în cele mai multe cazuri de modul în care o soluție este reprezentată și de problema căreia îi este aplicată optimizarea prin Simulated Annealing. Spre exemplu în cazul problemei clasice a comisului voiajor o soluție poate fi o permutație a orașelor, reprezentând ordinea în care sunt parcruse. Vecinii astfel pot fi generați prin interschimbări a oricăror două elemente respectiv orașe. Stările vecine sunt generate prin schimbări minore aduse stării curente.

În cazul în care Simulated Annealing este aplicat optimizării microarhitecturale spațiul de căutare respectiv proiectare este unul cu valori discrete și nu continue. Fiecare parametru

poate lua valori dintr-o mulțime finită specifică microarhitecturii țintă. Soluțiile vecine pot fi generate prin schimbări aduse unui singur parametru microarhitectural, pentru a evalua efectul creșterii sau scăderii acestuia asupra configurației curente, trecând la treapta inferioară sau cea superioară față de valoarea curentă. Se realizează astfel o căutare locală în spațiul de proiectare.

2.3.2.2. Probabilitatea de acceptare a soluțiilor inferioare. Funcția de acceptare

Simulated Annealing acceptă orice soluție care este mai bună, adică evaluarea indică energie mai mică decât soluția curentă. Spre deosebire de Hill Climbing care nu face decât să respingă orice soluție inferioară Simulated Annealing poate accepta aceste soluții cu o anumită probabilitate dependentă de temperatură și de diferența de energie dintre soluția curentă și cea nouă [11].

Probabilitatea de acceptare este o funcție de energia soluției curente, energia soluției inferioare propuse și de temperatura curentă.

Fie *s* soluția curentă și *s'* o soluție vecină evaluată.

$P(e, e', T)$, este funcția probabilitate de acceptare

$e = eval(s)$, energia soluției curente

$e' = eval(s')$, energia soluției candidat

T este temperatura curentă, care pornește de la o valoare pozitivă mare și scade treptat și controlat spre 0.

De asemenea în acest caz soluția candidată *s'* este inferioară soluției curente *s* (altfel *s'* ar fi fost deja acceptată), deci *e' > e*, având în vedere că o soluție cu energie mai mică este superioară uneia cu energie mai mare. Pentru ca o funcție să fie o alegere validă ca funcție de acceptare pentru Simulated Annealing trebuie să îndeplinească anumite condiții:

- Când temperatura (*T*) tinde spre zero, funcția *P* trebuie să tindă spre 0, astfel când temperatura este foarte mică algoritmul este „înghețat" și nici o soluție inferioară nu mai este acceptată

- Funcția *P* trebuie să fie pozitivă
- Cu cât diferența $e' - e$ este mai mare, cu atât probabilitatea de acceptare trebuie să fie mai mică. Dacă diferența de energie este mare înseamnă că soluția propusă *s'* este semnificativ mai slabă decât cea curentă *s*, dacă diferența este mică înseamnă că *s'* este doar puțin mai slabă decât *s*

Funcția de acceptare folosită este [11]:

$$P(en, en', T) = e^{\frac{-(en'-en)}{T}}$$

en energia soluției actuale

en' energia soluției candidat

T temperatura la iterația actuală

Această funcție îndeplinește condițiile amintite. Proprietățile menționate sunt vizibile în graficul funcției din figura 2.5. Se consideră evaluarea probabilității de acceptare față de o anumită soluție la diferite temperaturi, deci energia curentă (*en*) este constantă. Astfel valoarea funcției variază doar la schimbări de temperatură și la schimbări ale valorii energiei soluției candidat (altfel spus la schimbări ale diferenței $en' - en$).

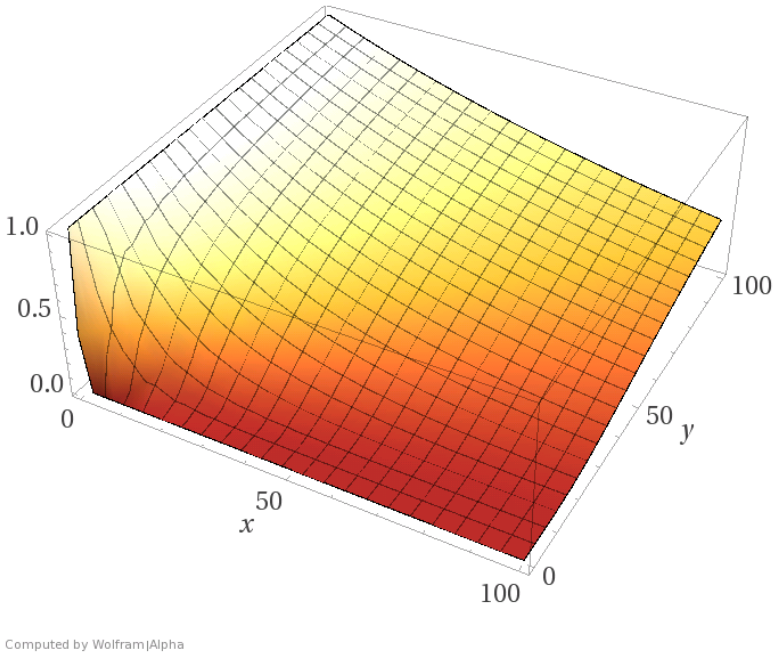

Figura 2.5 Graficul funcției P (probabilitatea de acceptare)

x reprezintă diferența dintre energia soluției candidate și energia soluției actuale ($en' - en$) iar *y* reprezintă temperatura. După cum se poate observa probabilitatea de acceptare crește odată cu creșterea temperaturii și scade odată cu creșterea diferenței dintre energia curentă si cea a soluției candidat. Valorile pentru temperatură și diferență sunt alese doar pentru exemplificare deoarece acestea depind în practică de problema la care este aplicat Simulated Annealing.

Pentru a pune în mai bună evidență aceste proprietăți sunt prezentate și grafice ale funcției la temperatură respectiv diferență constantă.

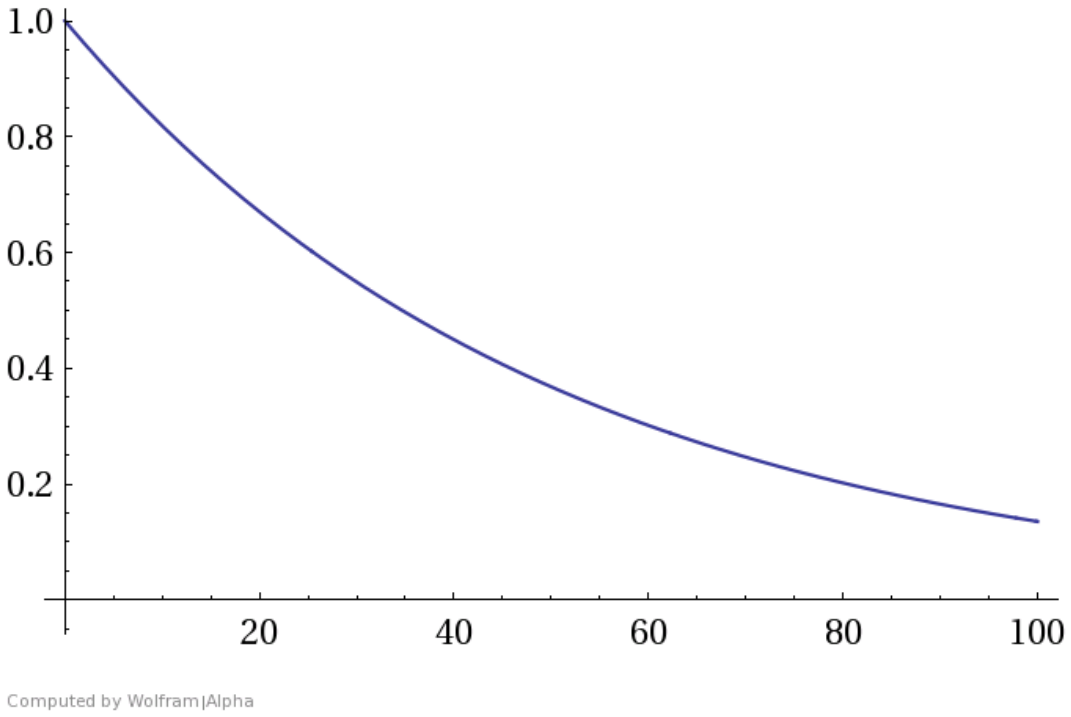

Figura 2.6 Probabilitatea de acceptare la temperatură constantă

Figura 2.6 reprezintă dependența probabilității de acceptare față de diferența dintre energia actuală și energia soluției candidat. Pe axa *Ox* este diferența $en' - en$, iar pe *Oy* este probabilitatea de acceptare. Pentru o bună vizibilitate a comportamentului funcției temperatura a avut valoarea 50. Funcția în acest caz este:

$$P(en', en) = e^{\frac{-(en'-en)}{50}}$$

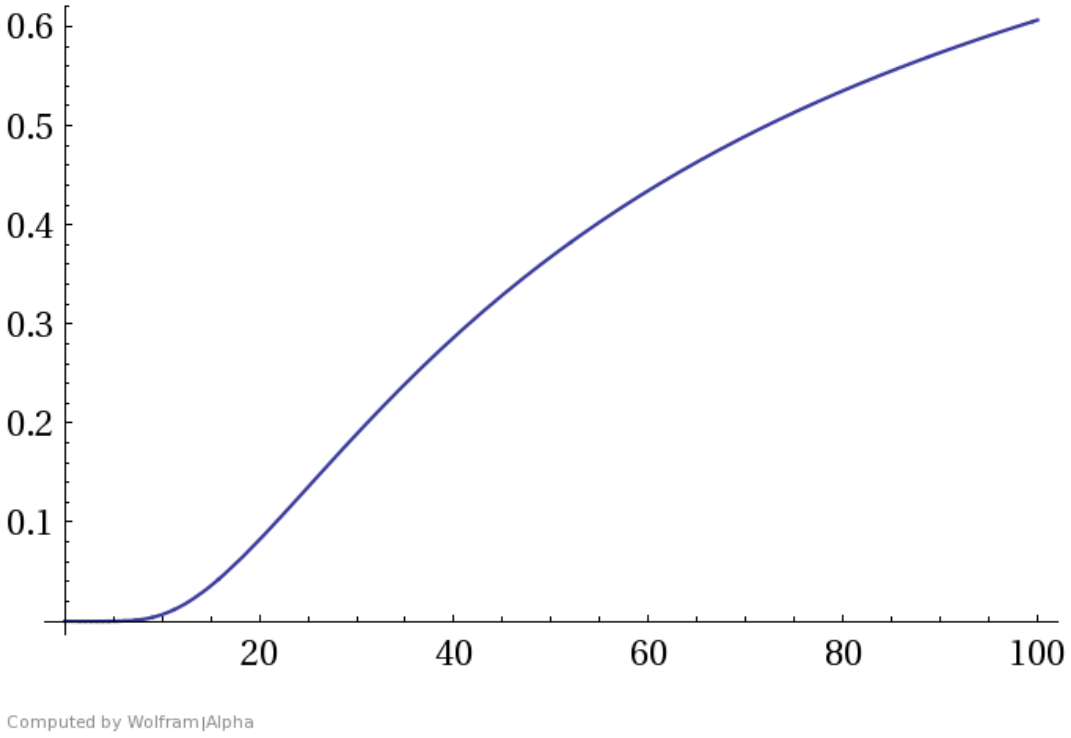

Figura 2.7 Probabilitatea de acceptare la diferență constantă

Figura 2.7 reprezintă dependența probabilității de acceptare față de temperatură. Pe axa *Ox* este temperatura, iar pe axa *Oy* este probabilitatea de acceptare. Diferența $en' - en$ dintre energiile soluției actuale și soluției candidat a avut valoarea 50. Funcția probabilității este astfel:

$$P(T) = e^{\frac{-50}{T}}$$

2.3.2.3. Programul de răcire

După cum am menționat anterior, Simulated Annealing folosește un parametru numit *temperatură* pentru a controla probabilitatea de acceptare a soluțiilor inferioare întâlnite. Acest parametru pornește de la o valoare mare inițială și scade în fiecare iterație a algoritmului conform unui *program de răcire („cooling schedule”)*. Valoarea inițială a temperaturii cât și programul de răcire ales pot influența rezultatele optimizării deoarece numărul de iterații necesare pentru a reduce temperatura la 0 și timpul petrecut la temperatură mare respectiv mică depind de această alegere. De asemenea în cazul în care algoritmul nu

este întrerupt înainte să ajungă temperatura la 0 aceste alegeri influențează și timpul de execuție, astfel algoritmul are nevoie de mai mult timp cu cât este mai mare temperatura inițială și cu cât este răcirea mai lentă.

Un program de răcire se poate descrie și implementa printr-o funcție temperatură de timp sau de iterație a algoritmului. Unele funcții de temperatură sunt:

a) Răcire liniară [15]

$$T(i) = T_0 - i \cdot c, \qquad \text{unde:}$$

i iterația algoritmului

T_0 temperatura inițială

c o valoare constantă pozitivă care controlează viteza răcirii

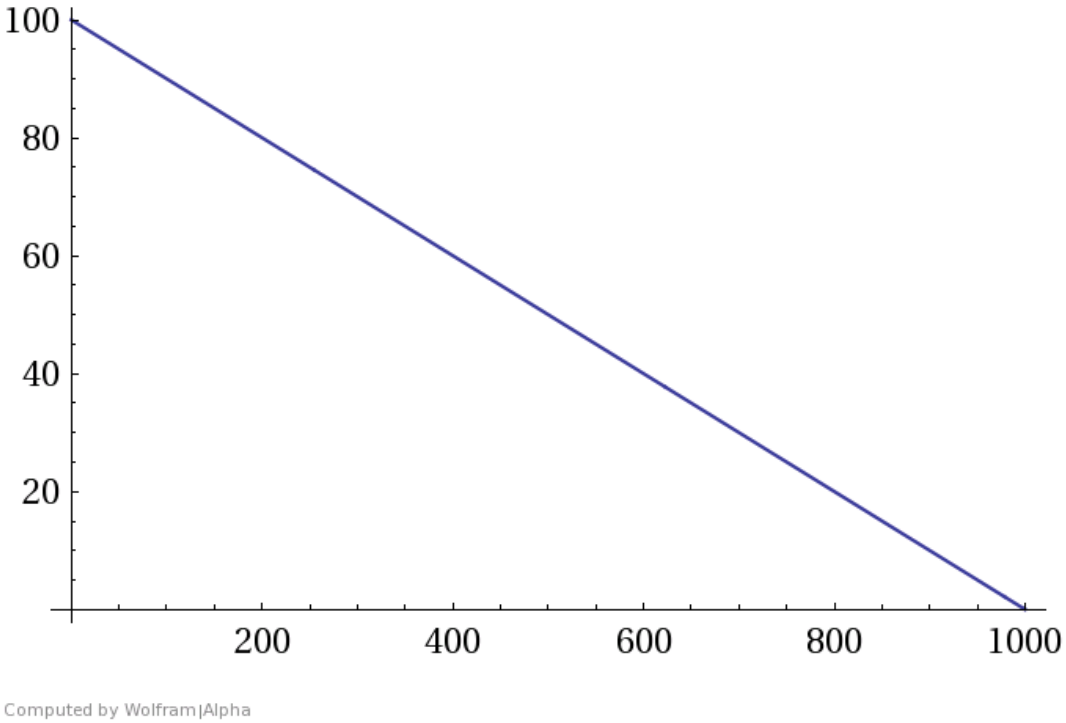

Figura 2.8 Răcire liniară

b) Răcire exponențială [15]

$$T(i) = T_0 \cdot c^i, \qquad \text{unde:}$$

i iterația algoritmului

T_0 temperatura inițială

c o valoare constantă pozitivă și subunitară care controlează viteza răcirii

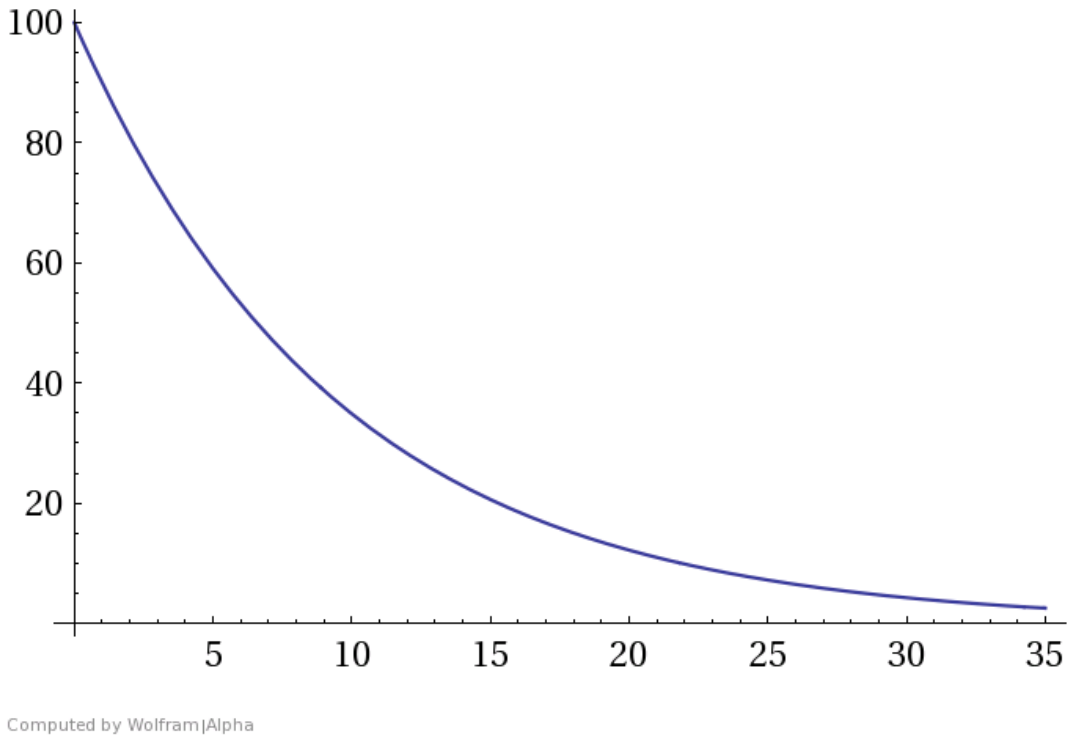

Figura 2.9 Răcire exponențială

c) Răcire Cauchy rapidă (*„Fast Cauchy”*) [16]

$$T(i) = T_0 \cdot \frac{1}{i}, \qquad \text{unde:}$$

i iterația algoritmului

T_0 temperatura inițială

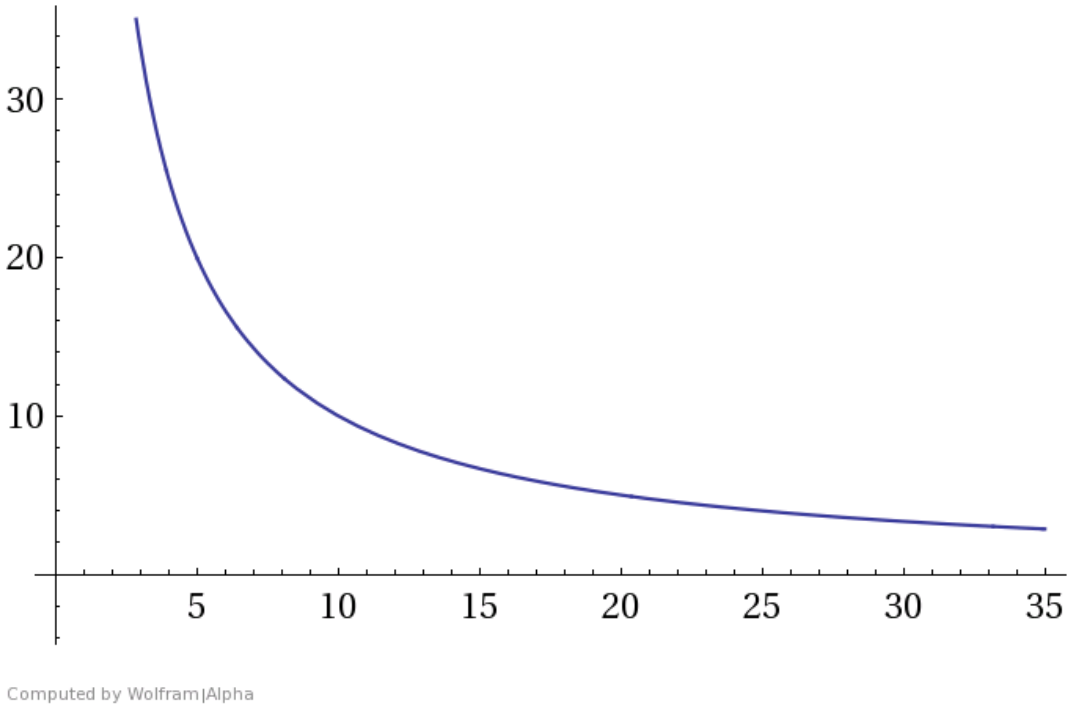

Figura 2.10 Răcire Cauchy

Spre deosebire de răcirea liniară și răcirea exponențială această funcție de răcire nu depinde de o constantă suplimentară pentru a controla viteza răcirii.

d) Răcire Boltzmann [16]

$$T(i) = T_0 \cdot \frac{1}{\ln i}, \qquad \text{unde:}$$

i iterația algoritmului

T_0 temperatura inițială

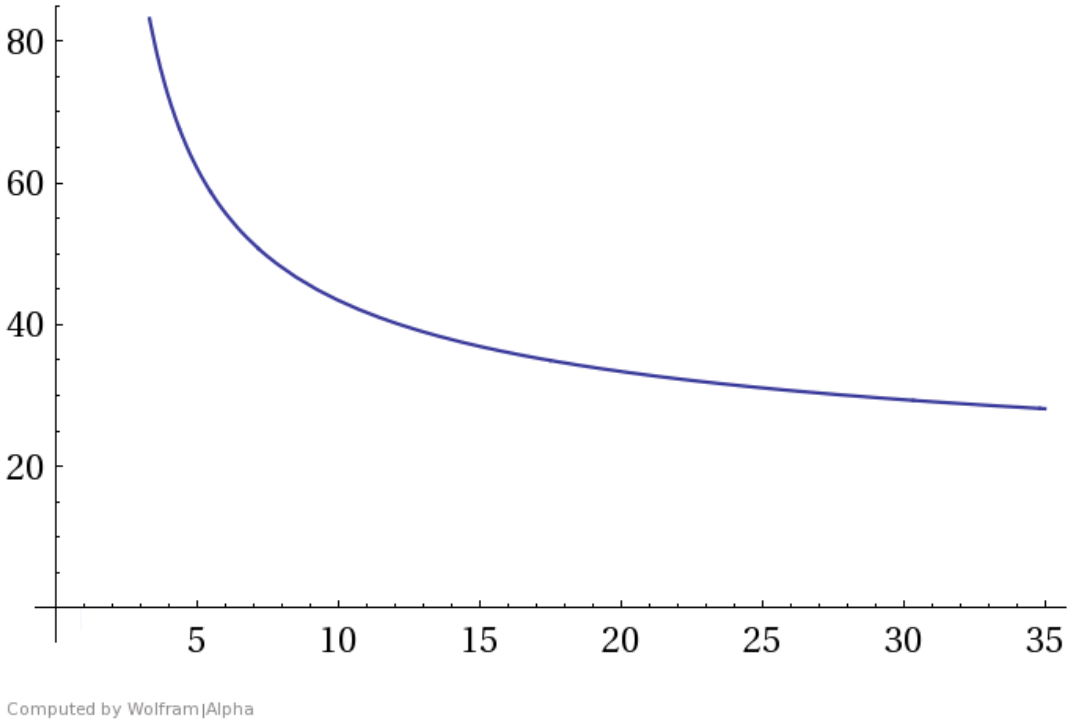

Figura 2.11 Răcire Boltzmann

La fel ca răcirea Cauchy, răcirea Boltzmann nu depinde de nici o constantă care controlează viteza de răcire.

2.3.2.4. Pseudocod

Algoritmul Simulated Annealing poate fi implementat cu multe variații. În continuare este prezentată o variantă generală [12].

```
SimulatedAnnealing
      solution = initial_solution;
      e = eval(solution);
      T = inital_temp;
      i = 0;
      repeat
            for k = 1 to some_constant
```

```
                next_solution = get_random_neighbour();
                if e > eval(next_solution)
                      e = eval(next_solution);
                      solution = next_solution;
                else if accept(solution, next_solution, T)
                      e = eval(next_solution);
                      solution = next_solution;
          end for
          i = i + 1;
          T = update_temp(i);
    until some stop condition is met;
    return solution;
```

Figura 2.12 Algoritmul Simulated Annealing în pseudocod

2.3.3. Algoritmi genetici

În domeniul inteligenței artificiale un algoritm genetic este o căutare euristică inspirată de procesul natural al evoluției utilizând operațiunea de selecție (*„selection”*), operatorii de mutație (*„mutation”*), de încrucișare (*„crossover”*) sau de recombinare (*„recombination”*) și conceptele de populație, individ, moștenire genetică și supraviețuirea indivizilor puternici (*„survival of the fittest”*). Algoritmii genetici sunt aplicați pentru a rezolva probleme de optimizare și de căutare și fac parte din categoria algoritmilor evolutivi (*„evolutionary algorithms”*).

Un algoritm genetic folosește o populație de soluții candidate (o soluție este numită *„individ”* sau *„cromozom”*) ale unei probleme de optimizare care sunt modificate și supuse unui proces de evoluție pentru a obține o populație de soluții mai bune. O soluție are un set de proprietăți (*„gene”*) care pot fi modificate prin procese de încrucișare și mutație. O reprezentare uzuală pentru o soluție este un șir de biți. Acest tip de reprezentare simplifică operațiile de încrucișare și mutație, dar în funcție de problemă pot fi folosite și alte reprezentări (permutări, șir de valori reale, etc.) [14].

Procesul de evoluție pornește de la o populație de indivizi generați aleator. Algoritmul genetic este un proces iterativ, fiecare iterație fiind numită *generație*. În fiecare generație este creată o populație nouă prin încrucișare sau recombinare și mutație folosind indivizi din populația curentă. Indivizii folosiți pentru a crea noua generație sunt selectați probabilistic, astfel indivizii mai buni respectiv soluțiile mai bune au șansă mai mare să ia parte la procesul de încrucișare, dar indivizii slabi nu sunt complet excluși, astfel se evită generarea de populații cu diversitate foarte mică în care toți indivizii sunt asemănător. Procesul aleator de mutație de asemenea ajută la explorarea spațiului de proiectare și la creșterea diversității populației noi[14].

Într-un algoritm genetic fiecare soluție respectiv fiecare individ are asociată o valoare numită *fitness* care reprezintă calitatea soluției. În cazul general se urmărește găsirea unei soluții cu fitness cât mai mare, iar cu cât un individ are o valoare de fitness mai mare cu atât este mai probabil ca acesta să fie ales în cadrul procesului de selecție pentru generarea urmașilor.

Pentru a adapta un algoritm genetic pentru o anumită problemă trebuie în primul rând găsită o reprezentare a soluțiilor. O reprezentare uzuală este un șir de biți, dacă această reprezentare poate fi folosită încrucișarea și mutația sunt ușor de implementat și folosit [14]. Metodele de încrucișare și mutație trebuie de asemenea adaptate pentru metoda de reprezentare a soluțiilor aleasă. De asemenea este nevoie de o funcție de evaluare a fitness-ului indivizilor. Această funcție este folosită pentru a calcula fitness-ul fiecărui individ din populație și din nou depinde de metoda de reprezentare utilizată cât și de problema la care este aplicat algoritmul. Spre exemplu, în cazul optimizării unei microarhitecturi pentru performanță se pot urmări valorile IPC (instructions per cycle), CPI (cycles per instruction) sau numărul total de ciclii mașină necesari pentru execuția benchmark-urilor. În unele cazuri, chiar în evaluarea performanței unei microarhitecturi, un individ nu poate fi evaluat direct folosind o funcție fitness. În aceste cazuri pot fi necesare simulări pentru a determina calitatea individului evaluat.

Un algoritm genetic poate avea mai multe condiții de oprire. În funcție de problemă, acestea pot fi:

- Un individ îndeajuns de bun a fost găsit
- Un număr maxim predefinit de generații a fost atins
- După un număr predefinit de generații nu a mai fost găsit un individ mai bun

În figura 2.13 este prezentat un algoritm genetic general [13].

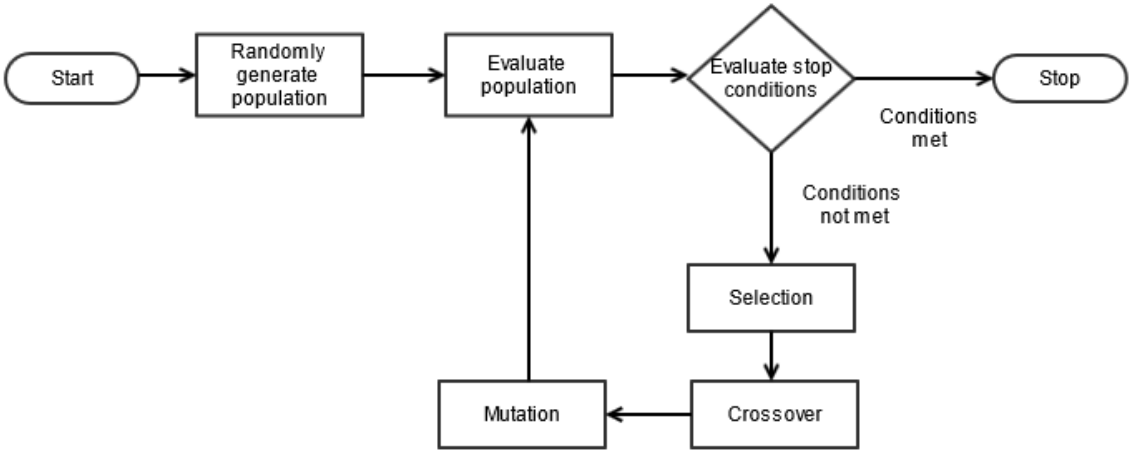

Figura 2.13 Algoritm genetic

2.3.3.1. Procesul de selecție

Operatorul de selecție are rolul de a alege un număr de indivizi din populația curentă care vor deveni părinți pentru indivizii din populația următoare. Cu cât un individ este mai bun, adică are fitness mai mare, cu atât este mai probabil ca operatorul de selecție să îl aleagă. Pentru a menține diversitatea unei populații indivizii mai slabi nu sunt complet excluși de majoritatatea metodelor de selecție. O populație în care nu există diversitatea se dorește a fi evitată deoarece aceasta limitează explorarea spațiului de căutare iar încrucișarea va produce doar indivizi asemănători.

În contextul algoritmilor genetici noțiunea de *elitism* descrie un algoritm genetic respectiv un proces de selecție care asigură supraviețuirea și existența în populația generației următoare a individului cu fitness maxim. Unele procese de selecție sunt elitiste implicit iar altele nu garantează alegerea celui mai bun individ, chiar dacă acesta are șansa cea mai mare de a fi selectat.

Există mai multe metode de a implementa operația de selecție. Printre cele mai cunoscute sunt *roulette wheel, stochastic universal sampling, rank selection* și *tournament*. În continuare vor fi prezentate aceste metode [13, 14].

a) Selecție *roulette wheel*

Metoda de selecție *roulette wheel* este una dintre cele mai simple opțiuni pentru a implementa această operație. Această metodă acordă fiecărui individ o șansă de a fi ales direct proporțională cu fitness-ul acestuia. Din acest motiv indivizii cu fitness foarte mic au o șansă foarte mică să fie aleși, deci diversitatea poate fi negativ afectată dacă doar indivizi buni sau foarte buni sunt aleși. Pentru a selecta un individ se calculează suma valorilor fitness din toată populația, apoi se generează un număr aleator mai mare decât 0 și mai mic decât această sumă. Fiecărui individ îi corespunde un interval între 0 și sumă de lungime egală cu fitness-ul acestui individ. Probabilitatea ca un individ să fie selectat este egală cu proporția fitness-ului individului din fitness-ul total. Selecția *roulette wheel* poate alege același individ de mai multe ori sau niciodată, cu cât fitness-ul este mai mare cu atât este mai probabil ca individul să fie ales de mai multe ori și să apară de mai multe ori în populația generației următoare.

Metoda de selecție *roulette wheel* nu este o metodă care garantează elitism, indiferent cât de mare este fitness-ul oricărui individ în comparație cu ceilalți.

În figura 2.14 sunt prezentate intervalele de selecție pentru o populație cu cinci indivizi respectiv soluții.

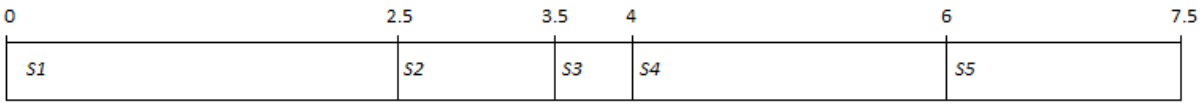

Figura 2.14 Intervale pentru selecție roulette wheel

În acest exemplu soluțiile din populație sunt notate *S1*...*S5* și au următoarele valorile fitness: *S1* – 2.5, *S2* – 1, *S3* – 0.5, *S4* – 2 și *S5*- 1.5. Suma valorilor fitness este 7.5 deci se generează

un număr aleator între 0 și 7.5. În funcție de intervalul în care se află numărul generat este selectat un individ:

S1 – între 0 și 2.5

S2 – între 2.5 și 3.5

S3 – între 3.5 și 4

S4 – între 4 și 6

S5 – între 6 și 7.5

b) Selecție *stochastic universal sampling*

Selecția folosind *stochastic universal sampling* este asemănătoare cu metoda *roulette wheel*, dar spre deosebire de aceasta este nevoie de o singură valoare aleatoare pentru a selecta întregul număr dorit de indivizi. De asemenea toți indivizii care au fitness-ul peste o anumită valoare vor fi selectați de una sau mai multe ori. *Stochastic universal sampling* folosește aceleași intervale ca metoda *roulette wheel* dar selectează indivizi folosind puncte echidistante între 0 și suma valorilor fitness. Numărul de puncte este egal cu numărul de indivizi care trebuie selectați iar distanța dintre puncte este suma calculată împărțită la numărul de puncte. În cazul în care fitness-ul unui individ este mai mare decât distanța dintre punctele de selecție, acesta va fi selectat cel puțin odată. Toate punctele de selecție sunt deplasate cu o valoare aleatoare mai mare decât 0 și mai mică decât distanța dintre puncte. Această metodă este utilizată deoarece încearcă să evite selectarea de foarte multe ori a unui singur individ dacă acesta are o valoare fitness foarte mare comparativ cu ceilalți indivizi din populație.

Stochastic universal sampling este o metodă elitistă în unele cazuri. Dacă dintr-o populație de *N* indivizi vor fi selectați *N* indivizi pentru a deveni părinți atunci toți indivizii cu fitness mai mare decât fitness-ul mediu vor fi selectați. Acest lucru se întâmplă deoarece distanța dintre puncte este egală cu media aritmetică a fitness-ului [14]. Având în vedere ca fitness-ul individului cel mai bun este evident mai mare decât fitness-ul mediu selecția acestuia este garantată.

În figura 2.15 este exemplificată metoda *stochastic universal sampling* folosind aceiași indivizi ca în cazul metodei *roulette wheel*.

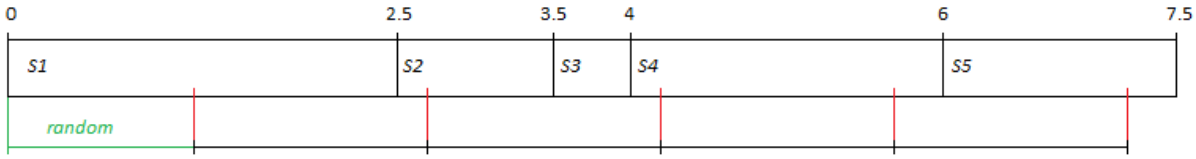

Figure 2.15 Selecție stochastic universal sampling

În acest exemplu valoarea aleatoare folosită este marcată cu verde, iar cu roșu sunt marcate punctele de selecție. Au fost aleși cinci indivizi, deci distanța dintre punctele de selecție este egală cu 1.5. Dacă valoarea aleatoare este egală cu cea din figură, cei cinci indivizi selectați sunt *S1*, *S2*, *S4*, *S4* și *S5*. După cum se poate observa *S4* a fost ales de două ori.

c) Selecție bazată pe rang (*rank selection*)

Metoda de selecție a indivizilor bazată pe rang este folosită pentru a reduce fitness-ul indivizilor foarte buni și pentru a crește fitness-ul indivizilor foarte slabi astfel încât să asigure o diversitate mai mare a populației următoare (înlătură dezavantajul selecției bazate pe ruletă prin considerarea unei ordini bazate pe rang, un fitness relativ, mai degrabă decât de fitness absolut). Totuși această metodă poate avea ca și consecință o convergență mai lentă spre o soluție foarte bună, tocmai din cauză că reduce fitness-ul celor mai buni indivizi. Selecția bazată pe rang întâi sortează populația în funcție de fitness, apoi valoarea fitness a fiecărui individ este înlocuită de o valoare între 1 și *N*, *N* fiind dimensiunea populației. Individul cu fitness cel mai mic primește valoarea 1, următorul valoarea 2 și așa mai departe, cel mai bun primind valoarea *N*. După ce a fost modificat fitness-ul fiecărui individ, selecția se poate face prin alte metode, cum ar fi cele prezentate anterior.

d) Selecția turneu (*tournament selection*)

Selecția de tip turneu este o metodă folosită pentru a menține o populație diversă. Această metodă alege aleator *k* indivizi din populație, după care individul cel mai bun dintre cei *k* este selectat. Această metodă acordă și indivizilor mai slabi o șansă mai mare de a fi selectați deoarece fiecare individ concurează doar cu alți *k – 1* indivizi, nu cu întreaga populație. Cu cât numărul de indivizi din turneu este mai mare cu atât probabilitatea ca un individ slab să fie

selectat scade. Dacă acest număr este mic indivizii mai slabi au șanse mai bune de a fi selectați. În cazul în care doar un individ ia parte la turneu, această metodă de selecție este complet aleatoare și nu ține cont de fitness.

2.3.3.2. Operatorul de încrucișare (*crossover*)

Operatorul de încrucișare (*crossover*) în cadrul unui algoritm genetic este echivalentul procesului biologic de încrucișare cromozomică prin care are loc schimbul de informație genetică între doi cromozomi. Procesul de încrucișare produce indivizi noi pornind de la doi sau mai mulți indivizi inițiali care sunt numiți *părinți*. Acest operator joacă rolul principal în crearea populației pentru generația următoare folosind indivizii aleși în timpul procesului de selecție. În mod uzual se folosesc doi părinți pentru a forma alți doi indivizi, fiecare fiind format din informație care provine de la ambii părinți. Proporția informației care provine de la fiecare părinte nu trebuie să fie egală și în funcție de metoda de crossover folosită poate să fie aleatoare.

Indiferent de tipul de încrucișare implementat, metoda trebuie să fie adaptată la reprezentarea folosită pentru indivizi. O reprezentare tipică pentru indivizi este un șir binar [14], operația de încrucișare având astfel rolul de a interschimba biți de la un părinte cu biți de la altul pentru a produce indivizi noi. În continuare vor fi prezentate unele metode de încrucișare: într-un singur punct (*one-point crossover*), în două puncte (*two-point crossover*) și încrucișare uniformă (*uniform crossover*) [14].

a) Încrucișare într-un singur punct (*one-point crossover*)

Încrucișarea într-un singur punct presupune alegerea unui punct de încrucișare și interschimbarea informației între părinți astfel încât să rezulte doi indivizi noi. Punctul de încrucișare este ales aleator și este același pentru ambii părinți. În cazul reprezentării indivizilor ca un șir de biți, se interschimbă biții pornind de la un anumit bit din șir, ales aleator. Cantitatea de informație provenită de la fiecare părinte este aleatoare, deoarece depinde în întregime de alegerea punctului de crossover.

În figura 2.16 este exemplificată încrucișarea într-un singur punct.

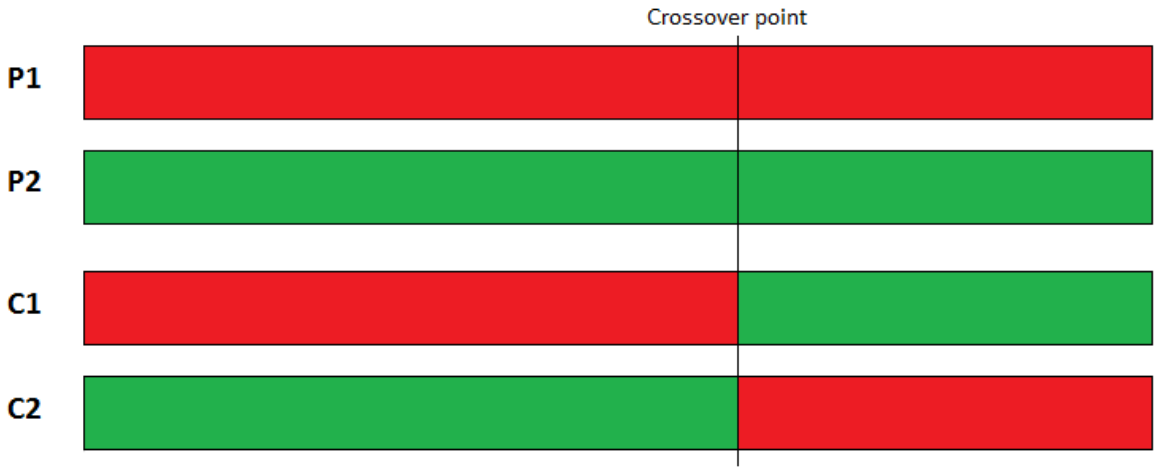

Figura 2.16 Încrucișare într-un singur punct

Indivizii *P1* (informație marcată cu roșu) și *P2* (informație marcată cu verde) sunt indivizii părinte care iau parte la încrucișare. După ce este ales aleator punctul de încrucișare, rezultă indivizii noi *C1* și *C2*, aceștia fiind construiți din informație provenind de la ambii părinți.

b) Încrucișarea în două puncte (*two-point crossover*)

Metoda încrucișării în două puncte este foarte asemănătoare cu încrucișarea într-un singur punct prezentată anterior. În acest caz se aleg două puncte de încrucișare, aceleași pentru ambii părinți și de această dată. Informația este interschimbată între punctele de încrucișare alese. În cazul reprezentării individului ca un șir binar se aleg doi biți iar șirurile dintre cele două poziții alese se schimbă între ele. La fel ca în cazul încrucișării într-un singur punct cantitatea de informație primită de la fiecare părinte este aleatoare și depinde doar de alegerea punctelor de încrucișare.

În figura 2.17 este prezentată metoda încrucișării în două puncte.

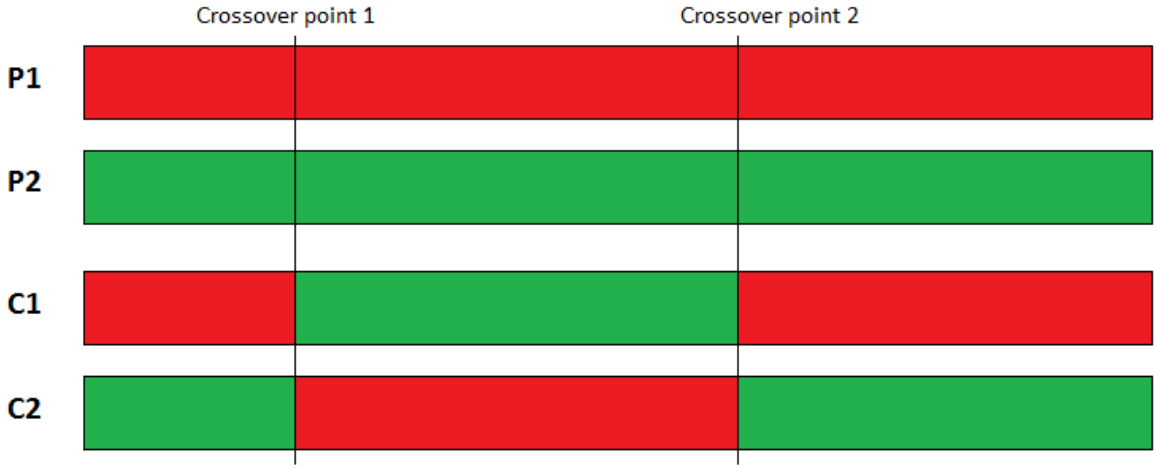

Figura 2.17 Încrucișare în două puncte

La fel ca în cazul încrucișării într-un singur punct, *P1* și *P2* sunt indivizii părinte iar *C1* și *C2* sunt indivizii noi formați din interschimbarea informației dintre punctele de încrucișare alese aleator.

c) Încrucișare uniformă (*uniform crossover*)

Încrucișarea uniformă este o metodă care interschimbă probabilistic informația la nivel de genă (unitate minim divizibilă de informație din reprezentarea aleasă). În loc să se interschimbe segmente întregi de informație, se interschimbă între ele doar gene cu valori diferite cu o anumită probabilitate care controlează proporția de informație care provine de la fiecare părinte. În cazul reprezentării indivizilor ca șiruri binare este evaluat fiecare bit în parte. În cazul în care biții de pe o anumită poziție au valori diferite în cadrul indivizilor părinte aceștia se schimbă între ei cu o anumită probabilitate. Această probabilitate controlează numărul de biți provenit de la fiecare părinte, iar în cazul în care este 0.5 fiecare părinte va contribui în medie același număr de biți.

În figura 2.18 este prezentat schematic un exemplu de încrucișarea uniformă.

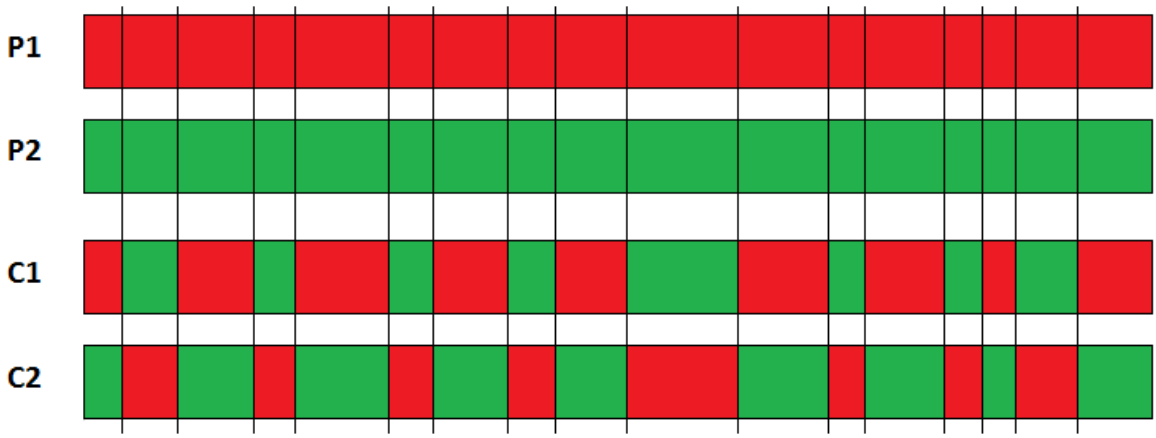

Figura 2.18 Încrucișare uniformă

Indivizii *P1* și *P2* sunt indivizii părinte, iar *C1* și *C2* sunt indivizii nou formați prin încrucișare uniformă. Spre deosebire de încrucișarea în unul sau două puncte în acest caz nu există puncte de încrucișare generate anterior.

2.3.3.3. Operatorul de mutație (*mutation*)

Operatorul de mutație în cadrul unui algoritm genetic are rolul de a schimba aleatori unii indivizi din populație. Mutația astfel ajută la păstrarea diversității populației și la explorarea spațiului de căutare al problemei. Acest operator modifică una sau mai multe gene ale unui individ, soluția deci poate fi complet diferită față de individul inițial și astfel pot fi găsite soluții mai bune cu gene care nu se găsesc în nici un individ din populația curentă, deci nu pot fi obținute prin încrucișare.

Mutația are loc cu o probabilitate care trebuie să fie foarte mică. Dacă această probabilitate ar fi foarte mare indivizii s-ar schimba foarte mult de la o generație la alta, deci îmbunătățirea iterativă și evoluția pe care se bazează algoritmii genetici nu ar putea avea loc. În acest caz algoritmul genetic s-ar transforma într-o simplă căutare aleatoare. În cazul în care probabilitatea de mutație este îndeajuns de mică schimbările introduse ajută la păstrarea diversității populației și astfel algoritmul genetic poate evita cazul în care toți indivizii sunt

foarte asemănători și converg spre un punct de extrem local și nu spre fitness-ul maxim global.

În cazul reprezentării indivizilor ca șiruri de biți operatorul de mutație poate funcționa în următorul mod. Pentru fiecare bit din reprezentare se generează un număr aleator, apoi cu o anumită probabilitate predefinită bitul este inversat. Această probabilitate este foarte mică, sub 1% în cele mai multe cazuri, iar cu cât reprezentarea individului folosește un număr mai mare de biți cu atât este mai probabil ca acesta să fie modificat. În alegerea probabilității de mutație trebuie deci ținut cont și de lungimea reprezentării [14].

O altă metodă de implementare a operatorului de mutație presupune generarea aleatoare a unui bit în loc de inversarea bitului [14]. Această metodă este echivalentă cu prima metodă prezentată, doar că mutația va modifica un bit doar în 50% din cazuri, anume doar în cazurile în care bitul generat aleator nu este egal cu cel aflat în reprezentarea individului.

Aceste metode de mutație nu pot fi aplicate în cazul în care soluțiile sunt reprezentate în așa fel încât valorile genelor sunt restricționate de ordinea apariției. Un astfel de caz este problema comisului voiajor în care o soluție este o permutare a numerelor de ordine ale orașelor. O schimbare aleatoare a unui număr din soluție nu este posibilă deoarece fiecare oraș apare o singură dată în reprezentare, trebuie deci o altă abordare a operatorului de mutație. Pentru această problemă mutația poate fi implementată prin interschimbări a numerelor orașelor de pe două poziții alese aleator, realizând astfel o schimbare aleatoare a traseului reprezentat.

2.3.3.4. Abordări multiobiectiv

Algoritmii genetici standard urmăresc optimizarea unui singur obiectiv, cum ar fi performanța în cazul optimizării unei microarhitecturi. În aplicații practice optimizarea unui singur obiectiv nu este întodeauna suficientă pentru o implementare reală a soluției găsite. Spre exemplu performanța unei configurații microarhitecturale este irelevantă dacă aceasta ar consuma atât de multă putere încât un procesor baza pe această microarhitectură nu ar putea fi răcit în realitate. Din acest motiv este adesea nevoie de o abordare a optimizării care să

urmărească mai multe obiective, cum ar fi performanța, consumul de putere și temperatura maximă a unui procesor care ar implementa o anumită microarhitectură. Totodată aceste obiective pot să fie conflictuale (cu cât o configurație microarhitecturală este mai performantă cu atât ar putea consuma mai multă putere), deci nu există o soluție care să fie optimă din punctul de vedere al tuturor obiectivelor urmărite, evaluarea soluțiilor devine astfel dificilă și nu există un mod simplu de a răspunde la întrebarea care este cel mai bun individ dintr-o populație [13].

În multe cazuri nu există o singură soluție pentru o problemă de optimizare multiobiectiv, deci algoritmii aplicați caută să optimizeze fiecare obiectiv până când alte obiective sunt afectate negativ. Algoritmii genetici de optimizare multiobiectiv sunt bazați pe algoritmul standard care optimizează un singur obiectiv [13].

Biblioteca *JMetal* pentru limbajul de programare Java oferă implementări a multor algoritmi de optimizare multiobiectiv și poate fi direct integrată în aplicații. În continuare vor fi prezentate schematic unele abordări ale optimizării multiobiectiv [13].

a) Abordare cu agregare a obiectivelor

Agregarea obiectivelor presupune utilizarea unei funcții de agregare care combină toate obiectivele într-o singură valoare care este apoi optimizată folosind un algoritm genetic mono-obiectiv. Funcția de agregare utilizează un coeficient pentru fiecare obiectiv, acestea fiind apoi însumate. În funcție de coeficienții aleși pentru fiecare obiectiv se poate stabili o prioritate a optimizării, obiectivele cu un coeficient mai mare fiind mai importante deoarece au un efect mai mare asupra valorii funcției.

b) Sortare lexicografică

Metoda sortării lexicografice presupune prioritizarea obiectivelor de către utilizator. Algoritmul optimizează pe rând obiectivele definite utilizând soluțiile obținute ca soluții inițiale pentru optimizarea următorului obiectiv. Această metodă produce rezultate diferite în funcție de ordinea în care obiectivele sunt definite.

c) Abordare bazată pe subpopulații

Această metodă împarte populație într-un număr de subpopulații egal cu numărul de obiective ale optimizării. Aceste subpopulații sunt apoi optimizate pentru câte un singur obiectiv urmând ca apoi să se realizeze un schimb de informație între ele pentru a produce soluții de compromis care să conțină optimizări pentru fiecare obiectiv.

d) Abordări bazate pe fronturi Pareto

Eficiența sau optimalitatea Pareto sunt concepte importante în economie, dar au aplicații și în teoria jocurilor cât și în inginerie. Abordarea optimizării multiobiectiv se bazează pe noțiunea de individ optim Pareto și pe fronturi Pareto formate din astfel de indivizi.

O situație este *eficientă sau optimă Pareto* când orice schimbare adițională benefică pentru un anume individ adusă acelei situații influențează negativ un alt individ. În cazul optimizării multiobiectiv o soluție este numită *eficientă sau optimă Pareto* când o schimbare adusă pentru a îmbunătăți un obiectiv va produce rezultate mai slabe din punctul de vedere al altui obiectiv.

O *îmbunătățire Pareto* este o schimbare care afectează pozitiv un individ fără să influențeze negativ alți indivizi. O schimbare adusă unei soluții care afectează pozitiv un obiectiv fără să producă rezultate mai slabe din punctul de vedere al altui obiectiv este o *îmbunătățire Pareto* în cazul optimizării multiobiectiv.

Un individ este numit *optim Pareto puternic (strong Pareto optimum – SPO)* când aceste este eficient Pareto, deci orice schimbare adusă soluției va avea efecte negative asupra a cel puțin un obiectiv. Acest termen este folosit pentru a diferenția față *optime Pareto slabe (weak Pareto optimum – WPO).*

Un *optim Pareto slab* este o soluție pentru care nu există schimbări care îmbunătățesc toate obiectivele.

Un *front Pareto* este o mulțime de indivizi optimi Pareto. Se spune că un individ *domină Pareto* alt individ dacă este mai bun din punctul de vedere al tuturor obiectivelor. Indivizii aflați astfel pe frontul Pareto sunt indivizi nedominați.

Algoritmii genetici bazați pe abordări Pareto funcționează fie eliminând indivizii dominați și generând indivizi noi prin mutație (*Simple Evolutionary Multiobjective Optimizer – SEMO, Fair Evolutionary Multiobjective Optimizer – FEMO)* fie prin sortări ale indivizilor pe fronturi Pareto succesive în etapa de selecție (*Nondominated Sorting Genetic Algorithm – NSGA, Nondominated Sorting Genetic Algorithm II – NSGA II*).

2.3.3.5. Pseudocod

În figura 2.19 este prezentată forma generală a unui algoritm genetic de optimizare bazat pe etapele de selecție, încrucișare și mutație.

```
GeneticAlgorithm
      generation = 0;
      P = generate_random_population();
      do
            evaluate(P);
            Parents = selection(P);
            Offspring = crossover(Parents);
            P = mutation(Offspring);
            generation = generation + 1;
      while stop conditions not met;
```

Figura 2.19 Un algoritm genetic în pseudocod

2.3.4. Algoritmi Particle Swarm Optimization

Particle Swarm Optimization este o tehnică de optimizare stohastică bazată pe o populație de indivizi și inspirată de comportamentul păsărilor. În cazul *Particle Swarm Optimization* indivizii (care se numesc *particule*) se deplasează prin spațiul de proiectare urmărind cel mai bun individ. „Roiul” de particule este alcătuit din indivizi care au asociată o poziție în spațiul

de explorare și o viteză a deplasării. Acești indivizi memorează poziția unde au obținut cele mai bune rezultate. De asemenea fiecare individ primește informații referitoare la cel mai bun individ din roi. În funcție de aceste informații se modifică viteza și poziția fiecărui individ, astfel realizându-se explorarea spațiului de căutare. La fel ca în cazul algoritmilor genetici populația inițială este generată aleator urmând ca această populație să fie actualizată o dată cu fiecare iterație executată [13].

2.3.5. Algoritmi stigmergici

Algoritmii stigmergici sunt inspirați de colonii de furnici și modul lor de funcționare este asemănător cu comportamentul acestora. Acești algoritmi se bazează pe o populație de indivizi echivalenți cu furnicile dintr-o colonie. În realitate furnicile caută aleator surse de hrană iar apoi se întorc la colonie lăsând o urmă de feromoni. Alte furnici sunt atrase de această urmă de feromoni și astfel vor urma același traseu pentru a găsi sursa de hrană. Feromonul lăsat în urmă se evaporă în timp. Această proprietate are ca rezultat căutarea unor drumuri mai scurte. Dacă un drum este mai lung durează mai mult până când o furnică îl poate parcurge, deci o parte mai mare din feromonul lăsat în urmă se evaporă. Dacă un drum este scurt acesta este parcurs mai des, deci sunt lăsați feromoni pe traseu mai des, astfel crescând intensitatea și atractivitatea pentru alte furnici. Aceste trăsături conduc la convergența spre drumul de lungime minimă deoarace acesta va avea intensitatea maximă a feromonului.

Un individ din cadrul populației reprezintă astfel o furnică și construiește iterativ o soluție bazându-se pe intensitatea feromonului pentru fiecare tranziție posibilă de la soluția intermediară curentă spre o soluție intermediară mai completă. Intensitatea feromonului artificial de asemenea scade în fiecare iterație a algoritmului pentru a simula evaporarea prezentă în realitate.

3. Implementare FOCAP Tool

FOCAP (*Framework for Optimising Computer Architecture Performance*) este un cadru flexibil pentru simularea distribuită multifir și optimizarea arhitecturilor de calcul folosind algoritmi evolutivi. Implementarea cadrului de optimizare permite adaptarea și integrarea ușoară a simulatoarelor existente, fără să fie necesare schimbări în aplicații. De asemenea cadrul de optimizare folosește o arhitectură modulară, atât în cadrul aplicațiilor implementate cât și în ceea ce privește împărțirea cadrului în aplicații cu funcții bine definite care comunică prin rețea folosind protocoale care vor fi descrise.

Cadrul de optimizare este compus din trei aplicații principale: server-ul FOCAP, clientul FOCAP care include simulatorul și interfața grafică pentru controlul server-ului. De asemenea este folosită și o bază de date MySQL care funcționează asemănător unei memorii cache pentru configurațiile microarhitecturale simulate de server. În cadrul acestei lucrări este tratată și explicată dezvoltarea și implementarea aplicațiilor server și client cât și integrarea clientului cu simulatorul extern pentru arhitectura SimpleScalar *Sim-Outorder*.

Aplicațiile server și client sunt implementate în limbajul de programare C folosind Microsoft Visual Studio 2012. Comunicarea prin rețea este implementată folosind biblioteca *WinSock2*, iar comunicarea cu server-ul MySQL utilizează biblioteca *libmysqlclient*.

3.1. Arhitectura generală

Cadrul de optimizare FOCAP este compus din trei aplicații care comunică prin rețea folosind diferite protocoale. Aceste aplicații sunt server-ul FOCAP, clientul FOCAP și interfața grafică pentru controlul și monitorizarea server-ului. Server-ul are un rol central în funcționarea aplicației. Acesta execută algoritmii de optimizare implementați, comunică cu baza de date, distribuie configurații microarhitecturale tuturor clienților, monitorizează funcționarea clienților pentru a detecta probleme și comunică cu instanțele de interfață grafică conectate. Server-ul de asemenea este implementat în așa fel încât să poate deservi un număr oricât de mare de clienți, fiecare client fiind capabil să utilizeze deplin resursele de execuție paralelă procesoarelor de tip multicore cu oricâte nuclee. Aplicația client are rolul de a executa

simulările primite de la server și de a trimite înapoi rezultatele acestora. Interfața grafică este folosită pentru a iniția procese de optimizare și pentru a seta parametrii acestora cum ar fi algoritmul de optimizare folosit, parametrii caracteristici algoritmului (cum ar fi probabilitatea de încrucișare și de mutație pentru un algoritm genetic) și configurația microarhitecturală inițială.

În figura 3.1 este prezentată arhitectura cadrului de optimizare, modulele principale din care sunt construite aplicațiile și interacțiunile acestora.

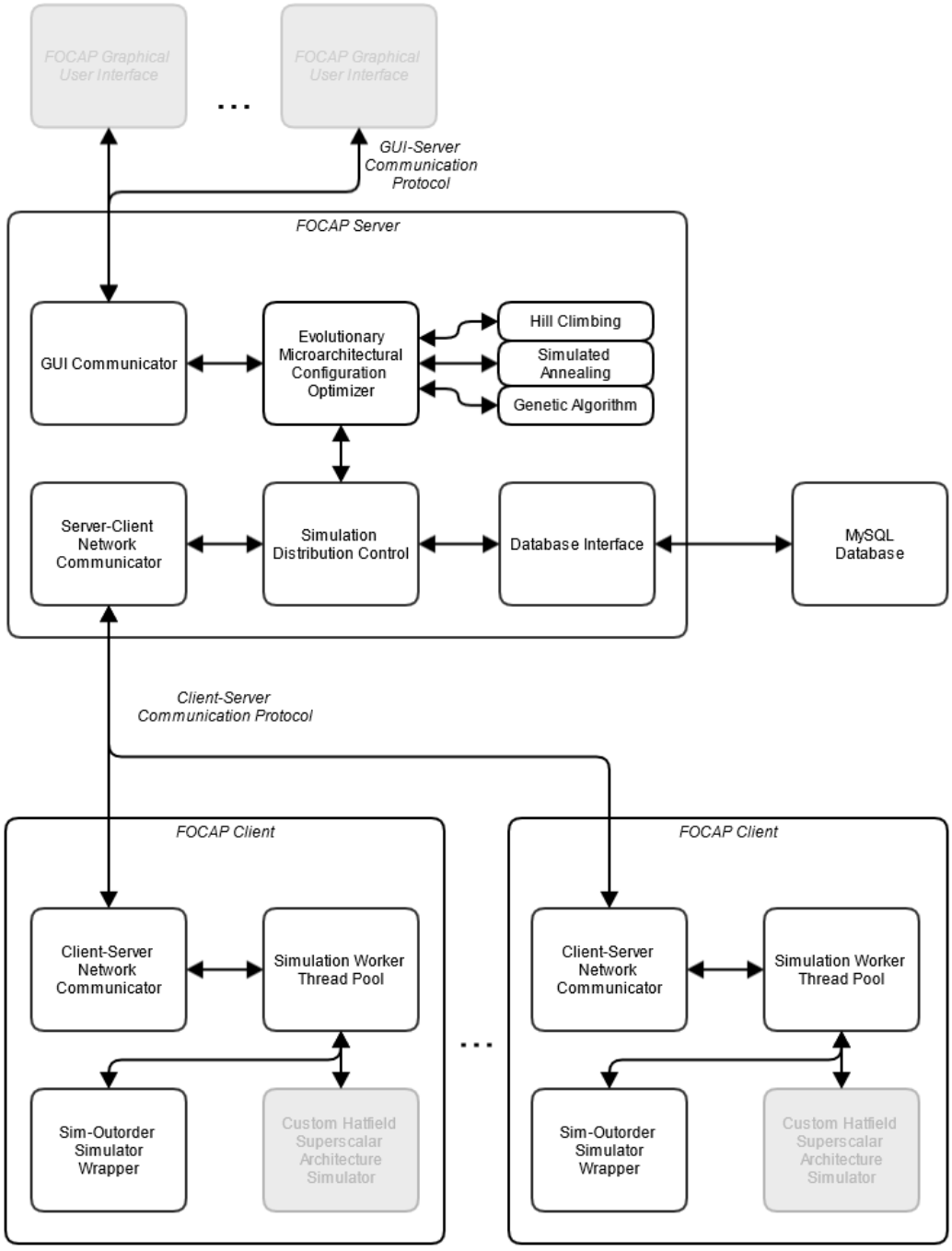

Figura 3.1 Arhitectura generală a cadrului de optimizare FOCAP

Modulele de culoare neagră din această diagramă implementează funcționalitatea de optimizare și distribuție a FOCAP. Fiecare modul îndeplinește o funcție specifică iar aceste module sunt realizate în așa fel încât influența lor asupra altor module să fie minimă. În cele mai multe cazuri fiecare modul folosește cel puțin un fir de execuție propriu iar comunicația între module diferite este realizată prin cozi FIFO (*First In First Out*) sincronizate accesate doar prin funcții specifice.

Aplicația server este formată din următoarele componente:

- Modul de comunicare cu interfața grafică (*GUI Communicator*): acest modul implementează protocolul de comunicare pentru interfața grafică și transmite cerințele de optimizare primite la modulul specific. Oricâte instanțe ale interfeței grafice pot fi conectate în același timp.
- Modul de optimizare microarhitecturală folosind algoritmi evolutivi (*Evolutionary Microarchitectural Configuration Optimizer*): acest modul primește cereri de optimizare și lansează în execuție algoritmul de optimizare corespunzător cererii curente. Acest modul de asemenea conține o coadă FIFO pentru cereri de optimizare multiple.
- Implementare a algoritmului Hill Climbing
- Implementare a algoritmului Simulated Annealing
- Implementare a unui algoritm genetic
- Modul de control al distribuției simulărilor (*Simulation Distribution Control*): funcția îndeplinită de acest modul este distribuția simulărilor, căutarea acestora în baza de date, inserarea în baza de date a rezultatelor noi și realocarea simulărilor care au fost trimise unui client dar rezultatele nu au fost returnate la timp
- Modul de comunicare cu server-ul MySQL (*Database Interface*): modul care îndeplinește funcția de comunicare cu server-ul MySQL, rolul de căutare a rezultatelor în baza de date și realizează adăugarea asincronă a rezultatelor noi la baza de date
- Modul de comunicare server-client prin rețea (*Server-Client Network Communicator*): acest modul implementează protocolul de comunicare asincronă folosit pentru transmiterea cererilor de simulare spre clienți și pentru recepția rezultatelor

Aplicația client conține următoarele module:

- Modul de comunicare client-server prin rețea (*Client-Server Network Communicator*): la fel ca modulul echivalent din cadrul aplicației server acest modul implementează protocolul de comunicare asincronă utilizat pentru distribuția simulărilor și pentru returnarea rezultatelor
- Modul de execuție multifir a simulărilor (*Simulation Worker Thread Pool*): acesta este modulul care implementează execuția paralelă multifir a mai multor simulări. Pentru fiecare fir de execuție pus la dispoziție de procesorul sistemului este creat un fir de execuție pentru rularea simulărilor
- Modul de adaptare pentru utilizarea simulatorului Sim-Outorder (*Sim-Outorder Simulator Wrapper*): acesta este un modul simplu care rolul de adaptor pentru simulatorul extern Sim-Outorder. Funcțiile îndeplinite sunt lansarea în execuție a simulatorului și interpretarea fișierului de ieșire pentru a extrage rezultatele
- Simulatorul propriu pentru arhitectura HSA (*Hatfield Superscalar Architecture*): un simulator dezvoltat pentru acest cadru de optimizare

Protocolul de comunicarea interfață-server este implementat folosind TCP și se bazează pe cereri trimise de interfață spre server care doar răspunde apoi acestor cereri. Server-ul nu inițiază comunicarea cu interfața în nici o situație.

Spre deosebire de protocolul de comunicare cu interfața grafică în cazul comunicării client-server se folosește UDP (*User Datagram Protocol*) și un singur soclu (*socket*) în cadrul aplicației server pentru toți clienții. Protocolul este implementat în acest mod pentru a evita necesitatea unui soclu de rețea pentru fiecare client.

3.2. Protocolul de comunicare client-server

Implementarea unui protocol de comunicare client-server a fost necesară datorită naturii distribuite a cadrului de optimizare. Având în vedere durata mare de timp necesară pentru a simula toate configurațiile microarhitecturale întâlnite în execuția unui algoritm de optimizare este necesară utilizarea a mai multor sisteme de calcul, pentru a beneficia de resursele de procesare paralelă disponibile, limitate doar de numărul sistemelor utilizate. Se impune deci

necesitatea dezvoltării unui cadru de simulare care să fie capabil să ruleze simulări în paralel pe un număr oricât de mare de sisteme. Din acest motiv a fost necesară implementarea și a unui protocol de comunicare pentru a stabili modul în care o configurație microarhitecturală este transmisă unui client și modul în care rezultatele simulării sunt returnate server-ului.

Pentru implementarea protocolului de comunicare a fost folosit UDP. Această alegere permite utilizarea unui singur soclu de rețea în cadrul server-ului pentru a gestiona comunicarea cu toți clienții conectați, spre deosebire de TCP în cazul căruia ar trebui folosit un soclu dedicat fiecărei conexiuni cu un client. Având în vedere faptul că viteza de simulare este puternic limitată de puterea de procesare pusă la dispoziție de clienți transferul de date pe rețea are nevoie de lățime de bandă mică, iar mesajele sunt trimise rar (la intervale de peste un minut). Riscul de a primi mesaje în ordine greșită este astfel foarte mic sau chiar inexistent în cadrul unei rețele LAN. De asemenea, având în vedere fiabilitatea în general foarte bună a rețelelor locale, dimensiunea mică a mesajelor și cât de rar sunt transmise aceste mesaje problemele cauzate de pierderea packet-elor din cauza rețelei sunt extrem de rare. Din aceste motive avantajele TCP (garantarea recepției informației și garantarea recepției în ordine) nu sunt necesare având în vedere overhead-ul asociat unui număr foarte mare de socluri utilizate (în cazul în care se folosesc mulți clienți) și simplitatea implementării UDP.

Pentru a distinge clienții și a simulările server-ul folosește valori de identificare care sunt transmise clienților împreună cu fiecare configurație microarhitecturală care trebuie simulată. Mesajele răspuns prin care sunt transmise server-ului rezultatele simulărilor conțin și aceste valori de identificare.

3.2.1. Descrierea mesajelor

Mesajele transmise sunt diferențiate în funcție de valoarea primului octet. Protocolul de comunicare este format din patru tipuri de mesaj:

a) Mesaj de conectare a clientului

   Acest tip de mesaj este transmis de fiecare client pentru a se conecta la server și conține numărul firelor de execuție ale clientului și timpul necesar clientului pentru a rula un benchmark standard. Primul octet are valoarea `SIM_MSG_HELLO (0x01)` iar

cele două valori sunt reprezentate ca numere întregi fără semn pe 32 biți. Dimensiunea totală este de 9B.

Nume câmp	**Valoare/descriere câmp**	**Dimensiune (B)**
SIM_MSG_HELLO	0x01	1
Threads	Numărul de fire de execuție ale procesorului client	4
Benchmark	Timpul necesar execuției benchmark-ului	4
Total	---	*9*

b) Mesaj de confirmare a conectării

Mesajul de confirmare a conectării este răspunsul server-ului la un mesaj de conectare. Acest tip de mesaj este doar un octet `SIM_MSG_ACK (0xFF)`

c) Mesaj de transmitere a unei simulări

Acest mesaj conține informații de identificare a simulării transmise, numele benchmark-ului care trebuie executat și configurația microarhitecturală care trebuie simulată. Primul octet este `SIM_MSG_WORK (0x07)` care face parte din header-ul de identificare. Acest header mai conține un identificator pe 32 biți pentru client, un identificator pe 32 biți pentru unitatea de lucru și un identificator pe 32 biți pentru simularea transmisă. Numele benchmark-ului este reprezentat folosind 21 de caractere iar configurația microarhitecturală este reprezentată folosind un vector de 21 de valori întregi pe 32 biți.

Nume câmp	**Valoare/descriere câmp**	**Dimensiune (B)**
SIM_MSG_WORK	0xFF	1
Client ID	Identificatorul clientului	4
Work Unit ID	Identificatorul unității de lucru (setul de simulări)	4
Work ID	Numărul simulării în cadrul setului	4
Benchmark	Numele benchmark-ului	21

Parameters	Parametrii microarhitecturali	21*4
Total	---	*118*

d) Mesaj de transmitere a rezultatului unei simulări

Aceste mesaje sunt transmise de clienți și conțin rezultatul unei simulări alături de informațiile aferente de identificare. De asemenea aceste mesaje apar doar ca răspuns la o cerere de simulare (un mesaj SIM_MSG_WORK). Valorile din header păstrează același format ca și în cazul cererii dar în loc de o reprezentare a unei simulări se trimite rezultatul simulării reprezentat prin 10 valori întregi pe 32 biți. Primul octet este `SIM_MSG_WORK_REPLY (0x00)`.

Nume câmp	**Valoare/descriere câmp**	**Dimensiune (B)**
SIM_MSG_WORK_REPLY	0x00	1
Client ID	Identificatorul clientului	4
Work Unit ID	Identificatorul unității de lucru (setul de simulări)	4
Work ID	Numărul simulării în cadrul setului	4
Benchmark	Numele benchmark-ului	21
Results	Rezultatele simulării	10*4
Total	---	*74*

3.3. Clientul FOCAP

Clientul FOCAP este aplicația folosită pentru execuția efectivă a simulărilor necesare. Această aplicație rulează pe sisteme de calcul client și este capabilă să exploateze eficient resursele de execuție paralelă a procesoarelor moderne. Mai mulți clienți se pot conecta la același server, iar performanța întregului sistem crește cu cât sunt conectați mai mulți clienți rulând pe mașini diferite.

3.3.1. Funcționalitate

Clientul FOCAP este o aplicație al cărei rol este execuția paralelă a simulărilor primite de la un server FOCAP. Aplicația este împărțită în mai multe module, fiecare modul îndeplinind o funcție bine definită care nu depinde de modul de implementare internă a altor module. Clientul este format din patru module ale căror funcționare va fi descrisă în continuare.

3.3.1.1. Modulul de comunicare prin rețea

Acest modul are rolul de a implementa comunicarea prin rețea a clientului. Tot acest modul conține mecanismul de confirmare a conexiunii cu un server, un interpretor pentru mesajele primite, o coadă FIFO sincronizată pentru mesajele care așteaptă să fie transmise și un fir de execuție propriu.

În momentul în care aplicația este lansată în execuție este inițializat modulul de rețea. Această inițializare presupune citirea dintr-un fișier de configurație a portului și a adresei IP a server-ului la care trebuie să se conecteze clientul. În caz de eroare clientul va căuta un server pe mașina pe care acesta rulează (*localhost* – 127.0.0.1). După determinarea adresei server-ului clientul trimite mesajul de conectare. În cazul în care clientul este la prima execuție, rulează un benchmark cronometrat al cărui rezultat este salvat apoi într-un fișier. Această valoare este trimisă server-ului. Dacă server-ul nu răspunde cu un mesaj de confirmare în 5 secunde, clientul încearcă de maxim 5 ori să se conecteze. În caz în care eșuează inițializarea nu continuă și aplicația devine inactivă. Tot în cadrul inițializării sunt create evenimente pentru recepția de date pe rețea (folosind funcția Windows API `WSAEventSelect`) și pentru transmisia de date (folosind funcția Windows API `CreateEvent`). Aceste evenimente sunt salvate într-un vector. În cazul în care nu au fost întâmpinate probleme este pornit firul de execuție principal al modulului.

Firul principal de execuție al acestui modul conține interpretorul de mesaje primite și codul de transmisie a mesajelor din coada FIFO menționată. Acest fir de execuție este folosit astfel încât comunicarea prin rețea să poată fi realizată într-un mod asincron din celelalte module.

Evenimentele create în etapa de inițializare a modulului sunt folosite în cadrul acestui fir de execuție.

Modul de funcționare este următorul:

1. Se așteaptă un eveniment folosind funcția API `WaitForMultipleObjects()` și vectorul de evenimente creat
2. Dacă evenimentul declanșat este cel pentru citire de pe soclu, se apelează `recvfrom()`
   2.1. Dacă a fost recepționată o cerere de simulare aceasta se copiază în coada FIFO a modulului de execuție multifir
3. Dacă evenimentul declanșat este cel pentru transmisie de mesaje, se transmit folosind `sendto()` toate mesajele din coada FIFO.
4. Se trece la pasul 1 pentru a aștepta noi evenimente

Acest mod de funcționare permite trimiterea de mesaje din alte module și fire de execuție a programului fără ca acestea să aștepte ca soclul de rețea să fie disponibil sau ca operațiile de citire să își încheie execuția.

3.3.1.2. Modulul de execuție multifir a simulărilor

Procesoarele moderne sunt procesoare multicore, având astfel capacitatea de a rula în paralel mai multe fire de execuție. Pentru a maximiza performanța clientului FOCAP pe sisteme care folosesc astfel de procesoare a fost implementat acest modul.

Pentru a utiliza deplin un procesor capabil să execute în paralel *N* sarcini, fie pentru că este un procesor cu *N* nuclee, fie pentru că este un procesor capabil de SMT (*Simultaneous Multithreading,* spre exemplu *HyperThreading* în cazul unor procesoare Intel moderne), sunt necesare cel puțin *N* fire de execuție. Din acest motiv modulul de execuție multifir determină numărul de fire de execuție care pot fi folosite prin intermediul funcției Windows API `GetSystemInfo()` și apoi creează *N* fire de execuție care așteaptă cereri de simulare prin intermediul cozii FIFO sincronizate a modulului.

În cadrul inițializării este creat un eveniment Windows care este declanşat în momentul în care sunt introduse cereri în coada FIFO a modulului. De asemenea tot în timpul inițializării sunt create firele de execuție necesare. Când nu sunt folosite pentru simulări aceste fire aşteaptă cereri, deci execuția unui fir nu se încheie în momentul în care o simulare a luat sfârşit.

Simulatorul este lansat în execuție din acest modul. Pentru a permite o mai mare flexibilitate în alegerea simulatorului nu sunt impuse condiții decât în ceea ce priveşte formatul datelor de intrare şi ieşire ale simulatorului. Datele de intrare, adică benchmark-ul care trebuie rulat şi configurația microarhitecturală, sunt reprezentate prin structurile `SimulationParameters` şi `SimulatorResult`. Atâta timp cât aceste convenții sunt respectate nu se impune nici o altă condiție asupra modului de execuție al simulatorului. Astfel pentru a integra alt simulator în clientul FOCAP nu este necesară decât implementarea unei scurte funcții care să realizeze traducerea datelor de intrare ale simulatorului şi a rezultatelor acestora. Acest lucru a fost implementat pentru aplicația externă Sim-Outorder care simulează un procesor cu arhitectură SimpleScalar. Modul în care această implementare a fost realizată va fi descris într-un capitol ulterior.

Firele de execuție create funcționează în următorul mod:

1. Se aşteaptă declanşarea evenimentului folosind funcția Windows API `WaitForSingleObject()`
2. Se extrage din coada FIFO o cerere de simulare
3. Din cererea de simulare se extrag datele de intrare pentru simulator, o structură `SimulatorParameters`
4. Este lansat în execuție simulatorul
5. După ce simularea s-a încheiat se construieşte mesajul răspuns care va fi trimis server-ului folosind structura `SimulatorResult` rezultată
6. Se apelează funcția de transmitere pusă la dispoziție de modulul de rețea
7. Se trece la pasul 1 pentru a aştepta alte cereri

Acești paşi sunt executați în paralel în cadrul mai multor fire echivalente şi nu există garanția că o simulare va fi executată de un anumit fir. Primul fir care răspunde evenimentului va fi cel

care execută o anume simulare, indiferent de câte fire de execuție sunt în stadiul de așteptare. În mod uzual în timp ce clientul FOCAP rulează simulări constant apare totuși doar rar cazul în care mai multe fire de execuție sunt libere în același timp.

3.3.1.3. Modulul de adaptare pentru utilizarea simulatorului Sim-Outorder

Sim-Outorder este un simulator de tip *execution-driven* pentru procesoare superscalare cu execuție speculativă bazate pe arhitectura SimpleScalar.

Acest simulator este o aplicație externă în linie de comandă (cu executabilul *sim-outorder.exe*) care poate fi controlată prin argumentele primite. Prin acest mod îi sunt transmise simulatorului informațiile referitoare la configurația microarhitecturală simulată, numele fișierelor de ieșire, numele fișierelor care conțin benchmark-ul cât și alți eventuali parametri pentru programul benchmark cum ar fi unele fișiere de intrare.

Pentru a integra acest simulator în clientul FOCAP este necesară transformarea structurii `SimulatorParameters` în argumente în linie de comandă pentru Sim-Outorder și apoi interpretarea fișierului de ieșire rezultat pentru a construi o structură de tip `SimulatorResult`.

Acest lucru este realizat printr-o funcție care ia locul simulatorului integrat în clientul FOCAP. O problemă care apare în cazul în care sunt folosite simulatoare externe care salvează rezultatele în fișiere de ieșire este numele acestor fișiere. Fiecare fir de execuție trebuie să apeleze simulatorul cu un alt nume pentru fișierul de ieșire, pentru a evita conflicte. Această problemă este rezolvată simplu folosind identificatorul fiecărui fir de execuție obținut cu funcția Windows API `GetCurrentThreadId()` și atașând această valoare numelui fișierului de ieșire.

Funcția construiește un șir de caractere care conține parametrii microarhitecturali și care este folosit pentru a apela sim-outorder.exe. Se așteaptă apoi ca execuția Sim-Outorder să se

încheie și se extrag rezultatele din fișierul de ieșire pentru a construi structura `SimulatorResult`.

Ultimul pas al execuției funcției de adaptare este ștergerea tuturor fișierelor generate de instanța simulatorului care tocmai și-a încheiat execuția.

3.3.2. Simulatorul pentru arhitectura HSA

Primul simulator care a fost folosit pentru FOCAP este o implementare proprie care a fost dezvoltată pentru simularea rapidă a benchmark-urilor Stanford (acestea fiind *fqueens, fbubble, fsort, ftower, ftree, fpuzzle, fmatrix, fperm*) folosind arhitectura HSA (*Hatfield Superscalar Architecture*). Tratarea modului de funcționare și de implementare a acestui simulator nu este obiectivul acestei lucrări, urmărind în schimb procesul de optimizare a microarhitecturilor și metode de distribuție și execuție în paralel a simulărilor. Proiectul prezentat în această lucrare face parte dintr-un ansamblu care conține și simulatorul de față și care a fost dezvoltat lucrând într-o echipă.

3.4. Server-ul FOCAP

Serverul FOCAP este aplicația centrală a cadrului de optimizare și are rolul de a rula algoritmii evolutivi. O singură instanță a aplicației server poate controla mai multe aplicații FOCAP client și o singură instanță server este necesară pentru fiecare sistem client-server FOCAP. Funcționarea sistemului de calcul pe care rulează server-ul FOCAP este critică pentru funcționarea întregului sistem din acest motv. În cazul în care apare o problemă în funcționarea sistemului server procesul de optimizare nu mai poate continua. În cazul în care un client ar întâmpina o problemă procesul de optimizare microarhitecturală poate continua fără acesta.

3.4.1. Funcționalitate

Aplicația server a cadrului FOCAP este compusă la fel ca și aplicația client din mai multe module care îndeplinesc funcții diferite. Aplicația server este mai complexă decât cea client și

este formată din cinci module principale și încă trei module secundare care implementează algoritmii de optimizare Hill Climbing, Simulated Annealing și un algoritm genetic.

3.4.1.1. Modulul de comunicare cu interfața grafică

Server-ul FOCAP este doar o aplicație de consolă care nu necesită intervenția utilizatorului pentru a funcționa după ce a fost pornit un proces de optimizare. De asemenea aplicația server ar putea fi lansată în execuție și pe sisteme la care utilizatorul nu are acces direct pentru a monitoriza progresul. Din aceste motive server-ul FOCAP este o aplicație care poate fi controlată prin rețea folosind un protocol care va fi descris ulterior și care este implementat în acest modul. Acest mod de control permite monitorizarea server-ului de pe orice alt sistem de calcul sau dispozitiv care are implementată o interfață grafică FOCAP.

Acest modul suportă mai multe interfețe grafice conectate în același timp și poate accepta sarcini de simulare de la oricare dintre ele. De asemenea tot acest modul are rolul de a trimite informații referitoare la progresul proceselor de optimizare în momentul în care o interfață grafică cere aceste informații.

La fel ca în cazul modulelor client și acest modul este format dintr-o parte de inițializare executată când aplicația server este pornită și dintr-un fir de execuție care este folosit pentru comunicare prin rețea și pentru tratarea cererilor primite de la interfețele grafice conectate. Pentru a diferenția mesajele se alocă atât un identificator unic pentru fiecare interfață cât și un identificator unic pentru fiecare mesaj primit.

Partea de inițializare a acestui modul presupune deschiderea unui soclu TCP care așteaptă conexiuni de la interfețe grafice și crearea unui fir de execuție pentru comunicare. În cazul în care acest soclu nu poate fi deschis procesul de inițializare este întrerupt. Având în vedere că protocolul folosit este TCP se memorează o listă de socluri deschise care sunt folosite pentru a comunica cu interfețele grafice conectate. Pentru a gestiona într-un mod asincron comunicarea folosind mai multe socluri este folosită funcția Windows API `select()`.

Firul de execuție al acestui modul funcționează în următorul mod:

1. Folosind funcția `select()` se așteaptă primirea unui mesaj pe un soclu sau se așteaptă ca o nouă instanță de interfață grafică să se conecteze
2. Dacă o nouă instanță încearcă să se conecteze este acceptată și se adaugă soclul rezultat în lista de socluri active
3. Se parcurge lista de socluri, fiecare soclu care a fost închis este eliminat din listă
4. Se recepționează mesajele de pe fiecare soclu cu date disponibile și se interpretează aceste mesaje
   4.1. Dacă mesajul este o sarcină de simulare se execută următorii pași
      4.1.1. Se caută identificatorul unic al mesajului în lista de sarcini de simulare primite
      4.1.2. Dacă acesta nu este găsit, este adăugat în listă și sarcina de optimizare este transmisă modulului de optimizare
      4.1.3. Se trimite un mesaj de confirmare spre interfața grafică
   4.2. Dacă mesajul este o cerere de informații referitoare la sarcinile de optimizare actuale se construiește un mesaj de răspuns și se trimite acest mesaj
5. Se trece la pasul 1 pentru a aștepta alte mesaje sau conexiuni

Această implementare permite gestionarea comunicării cu mai multe interfețe grafice folosind mai multe conexiuni TCP utilizând un singur fir de execuție.

3.4.1.2. Modulul de optimizare microarhitecturală

Rolul acestui modul este execuția sarcinilor de optimizare primite de la modulul de comunicare cu interfața grafică. Acest modul folosește o coadă FIFO sincronizată pentru a stoca sarcinile de optimizare care nu au fost încă terminate și un fir de execuție care este folosit pentru a rula algoritmii de optimizare implementați. Firul de execuție așteaptă ca un eveniment să fie declanșat pentru a prelua sarcini de simulare din coada FIFO a modulului.

Inițializarea acestui modul presupune crearea evenimentului Windows care este folosit de firul de execuție folosind funcția `CreateEvent()` și apoi crearea firului de execuție în sine

folosind funcția `CreateThread()`. În cazul în care sunt întâmpinate erori este întrerupt procesul de inițializare și server-ul FOCAP devine inactiv deoarece funcționarea acestui modul este esențială pentru funcționalitatea aplicației. Firul de execuție are rolul de a prelua sarcinile de optimizare din coadă și de a rula algoritmul de optimizare corespunzător.

Firul de execuție urmărește succesiunea de pași:

1. Așteaptă folosind `WaitForSingleObject()` ca evenimentul creat la inițializare să fie declanșat
2. Se extrage din coadă o sarcină de simulare
3. Se lansează în execuție un algoritm de optimizare (Hill Climbing, Simulated Annealing sau algoritmul genetic) în funcție de cerințele sarcinii de optimizare extrasă
4. După finalizarea execuției algoritmului de optimizare se sare la pasul 2 dacă mai sunt alte sarcini în coada FIFO
5. Se trece la pasul 1 pentru a aștepta alte sarcini de optimizare

Acest modul alături de implementările algoritmilor de optimizare care vor fi descrise în subcapitole ulterioare joacă rolul central în funcționarea procesului de optimizare microarhitecturală. Se folosește un fir de execuție separat pentru a păstra modularitatea aplicației.

3.4.1.3. Modulul de control al distribuției simulărilor

Modulul de control al distribuției simulărilor joacă un rol foarte important în asigurarea performanței întregului sistem FOCAP. Pentru a putea utiliza eficient resursele de procesare paralelă puse la dispoziție de toate mașinile client conectate la un server trebuie ca fiecare procesor disponibil să ruleze simulări cu întreruperi minime, deci server-ul trebuie să asigure trimiterea unei simulări noi în momentul în care un client intră într-o stare de așteptare, deci în momentul în care a trimis un rezultat. Acest modul de asemenea are rolul de a garanta simularea tuturor configurațiilor microarhitecturale cerute și astfel implementează un mecanism de realocare a simulărilor în cazul apar probleme la un client și sunt pierdute rezultate.

Modul în care simulările sunt alocate clienților poate de asemenea să aibă un impact asupra performanței sistemului, în special în care sunt necesare doar puține simulări. În acest caz utilizarea celor mai rapizi clienți liberi conduce la creșterea performanței sistemului prin reducerea timpului de așteptare. Rezultatele benchmark-urilor inițiale ale clienților sunt folosite pentru a organiza clienții liberi într-un heap. Acest heap este actualizat de fiecare dată când un client termină execuția unei simulări, când un client nou se conectează și când un client primește o sarcină de simulare și devine astfel ocupat. Simulările sunt întâi alocate clienților cu performanță cât mai mare.

Rezultatul benchmark-ului unui client are de asemenea rolul de a indica timpul maxim aproximativ în care o simulare este finalizată. Această estimare este folosită pentru a determina timpul maxim de așteptare pentru rezultate. De asemenea estimarea este specifică fiecărui client. Când rezultatele unei simulări nu sunt primite în intervalul de timp așteptat această simulare este trecută din nou în coada de simulări care nu au fost rulate și va fi trimisă altui client. Clientul care a întârziat este trecut într-o listă și nu îi mai sunt trimise alte simulări pentru o anumită perioadă de timp. Dacă același client întârzie de mai multe ori decât o limită fixă (3 ori) acesta este eliminat de server deoarece întârzieri repetate indică fie o problemă la comunicarea prin rețea fie o problemă cu sistemul pe care rula acel client și este nevoie de intervenția utilizatorului pentru a rezolva această problemă. Alt motiv pentru care sunt eliminați acești clienți care dau dovadă de fiabilitate redusă este faptul că afectează negativ performanța întregului sistem prin așteptările introduse.

Pentru a exploata capacitatea de execuție paralelă a unui sistem client-server FOCAP modulul de distribuție permite organizarea simulărilor necesare în seturi numite *unități de lucru* (*work units*). O simulare în acest caz este considerată o sarcină independentă (*work*) din cadrul setului. Un *work unit* este deci compus din mai multe simulări (*work*). Un *work unit* ar putea conține spre exemplu toate simulările necesare pentru a evalua populația unei generații a unui algoritm genetic.

În figura 3.2 este prezentată funcționarea modulului de distribuție printr-un exemplu care conține un server FOCAP și trei clienți de performanță diferită. În acest caz procesoarele sistemelor exemplificate ar oferi aceeași performanță la aceeași frecvență clock, deci sistemul care funcționează la 3GHz este cel mai rapid, iar sistemul care funcționează la 1GHz este cel

mai lent. De asemenea în exemplu trebuie distribuite trei seturi de simulări, reprezentate prin culorile roșu, verde și albastru. Primul set conține doar 3 simulări, al doilea conține 5 simulări, iar al treilea conține n + 2 simulări.

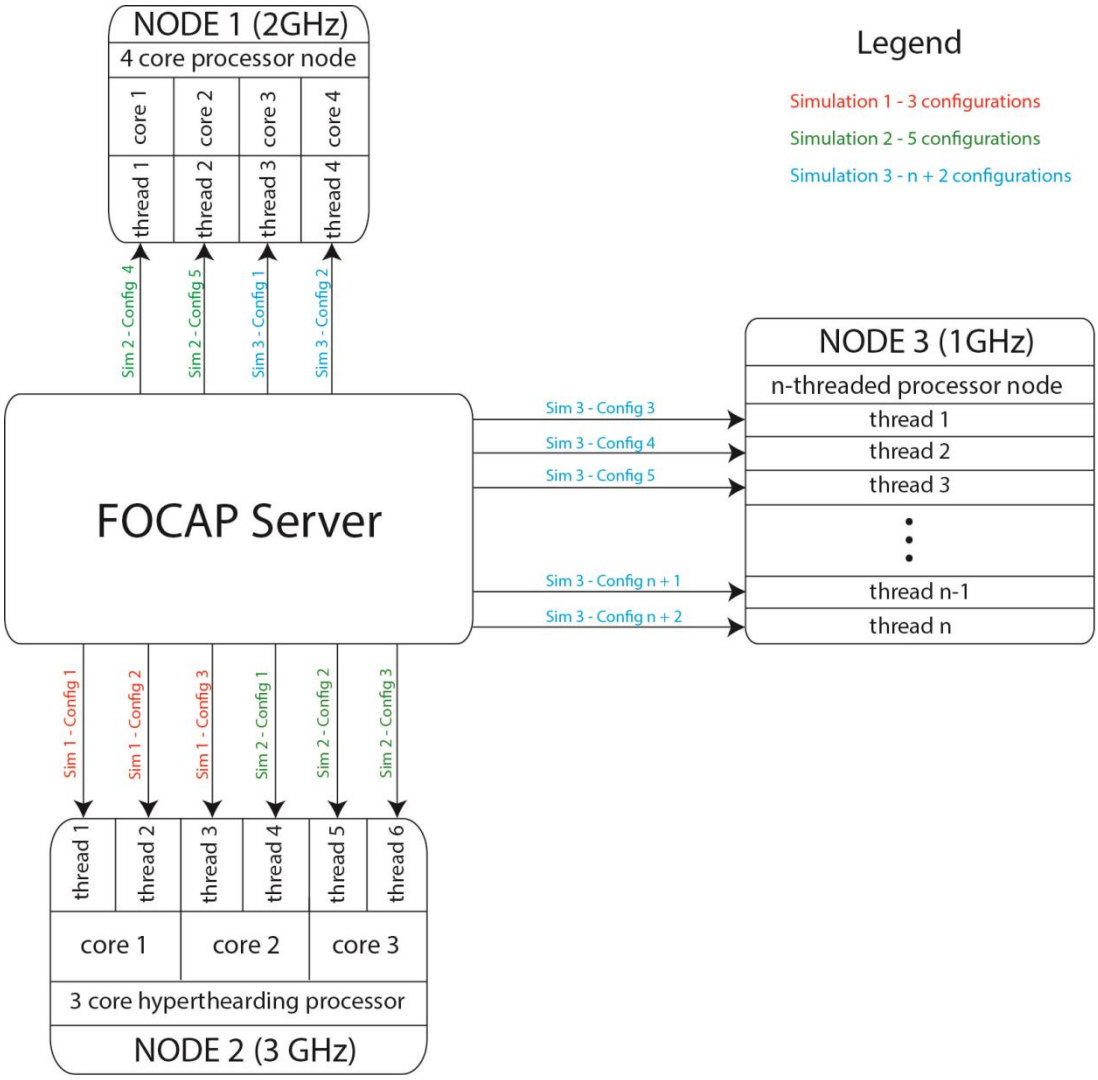

Figura 3.2 Distribuția simulărilor

Pentru a distribui simulările trebuie creat și populat un *work unit*. După ce un work unit a fost construit se poate apela modulul de distribuție prin funcția `dispatch_work_unit()` pentru a trimite simulările clienților. De asemenea este pusă la dispoziție o funcție care cauzează firul de execuție apelant să aștepte până când toate simulările din work unit au fost rulate. Această funcție este implementată folosind un eveniment Windows API declanșat doar când a fost recepționat rezultatul ultimei simulări din set.

Sistemul de distribuție este bazat pe o coadă FIFO în care se află toate simulările care trebuie rulate și pe heap-ul care conține clienții liberi. Acest modul de asemenea comunică cu modulul de interfață cu baza de date pentru a prelua rezultatele simulărilor care au mai fost rulate. Dacă simularea nu este găsită în baza de date va fi alocată unui client. Pentru fiecare client liber este extrasă o simulare din coadă, până când nu mai sunt simulări în coadă sau până când nu mai sunt clienți liberi.

Acest modul este construit asemănător modulelor descrise anterior, folosind o funcție de inițializare apelată când server-ul este lansat și un fir de execuție. Firul de execuție este folosit pentru a implementa mecanismul descris de realocare a simulărilor și de monitorizare a clienților.

În cadrul acestui modul firul de execuție urmărește următorii pași:

1. Se parcurge lista de simulări active și se incrementează timpul trecut pentru fiecare simulare
2. Dacă acest timp a depășit valoarea limită, simularea este adăugată din nou în coada de simulări iar clientul care a întârziat este adăugat listei de clienți întârziați dacă nu a depășit limita, altfel este eliminat
3. Se parcurge lista de clienți întârziați și se incrementează timpul petrecut în această listă pentru fiecare client
4. Dacă acest timp a depășit sau a atins limita de așteptare, clientul este trecut în heap-ul de clienți liberi
5. Se așteaptă 5 secunde
6. Se trece la pasul 1

Acest modul este implementat în așa fel încât funcționalitatea sa poate fi utilizată ușor prin intermediul unor funcții apelate din implementările algoritmilor de optimizare.

3.4.1.4. Modulul de comunicare prin rețea

În cadrul acestui modul este implementată funcționalitatea server a protocolului de comunicare prin rețea descris în subcapitolul 3.2. Construcția modulului este asemănătoare cu cea a componentei echivalente din cadrul clientului. Acest modul conține interpretorul de mesaje primite și codul pentru trimiterea mesajelor spre clienți.

Inițializarea acestui modul presupune creare unui eveniment pentru trimiterea mesajelor, crearea unui eveniment și asocierea acestuia cu recepția datelor pe soclul UDP folosind `WSAEventSelect()` și crearea unui fir de execuție. Singura diferență importantă față de implementarea modulului echivalent din clientul FOCAP este că mesajele trimise nu sunt adresate doar unei singure mașini deoarece server-ul comunică cu toți clienții.

Firul de execuție este folosit pentru a permite altor module să trateze comunicarea prin rețea într-un mod asincron. Acesta folosește o coadă FIFO pentru mesaje care trebuie trimise iar mesajele primite sunt interpretate direct urmărind următorii pași:

1. Se așteaptă ca unul din evenimentele de recepție a mesajelor sau de trimitere să fie declanșat
2. Dacă evenimentul declanșat a fost cel de citire, se execută următorii pași:
   2.1. Dacă mesajul primit este unul de conectare se adaugă clientul în lista de clienți și în heap-ul de clienți liberi apoi se trimite un mesaj de confirmare clientului
   2.2. Dacă mesajul primit este rezultatul unei simulări acesta este transmis modulului de distribuție pentru a fi tratat
3. Dacă evenimentul declanșat a fost cel de trimitere a mesajelor, se execută următorii pași:
   3.1. Se extrage din coada FIFO un element
   3.2. Se determină adresa clientului și mesajul folosind informațiile extrase anterior
   3.3. Se trimite mesajul clientului folosind funcția `sendto()`
   3.4. Dacă mai sunt elemente în coadă, se trece la pasul 3.1
4. Se trece la pasul 1 pentru a aștepta alte evenimente

3.4.1.5. Modulul de comunicare cu server-ul MySQL

Server-ul FOCAP folosește o bază de date MySQL pentru a stoca rezultatele simulărilor deja rulate. În cazul în care o simulare este găsită deja în baza de date rezultatul poate fi obținut mult mai rapid fără a mai fi necesară rularea benchmark-ului aferent. Prin acest mecanism se reduce timpul necesar procesului de optimizare și astfel crește performanța. Baza de date de asemenea are rolul de a stoca rezultate cât timp server-ul FOCAP nu rulează, deci informația nu este pierdută nici în cazul în care apare o problemă în funcționarea server-ului.

Pentru a identifica o simulare în baza de date se folosesc ca și cheie primară numele benchmark-ului și o valoare întreagă fără semn pe 64 biți. Această valoare reprezintă o configurație comprimată. Rezultatele sunt stocate în baza de date prin serializare într-un câmp de tip blob. Acest mod de reprezentare a fost ales deoarece permite modificarea configurației microarhitecturale și a informației reprezentate într-un rezultat fără a aduce schimbări tabelei folosite. De asemenea un alt avantaj al utilizării compresiei este cantitatea mai mică de informație transmisă server-ului MySQL și căutarea mai rapidă în baza de date deoarece nu sunt necesare interogări care să conțină condiții pentru toți cei 21 de parmetrii microarhitecturali (84B), aceștia fiind înlocuiți cu o singură valoare de 8B. Tabela folosită este următoarea:

Câmp	Tip	Null	Cheie primară	Valoare implicită
config	bigint unsigned	Nu	*	NULL
benchmark	char (21)	Nu	*	NULL
result	blob	Da		NULL

Compresia repezentării parametrilor microarhitecturali se bazează pe faptul că fiecare parametru arhitectural poate lua doar un număr relativ mic de valori diferite. Având în vedere această proprietate fiecare parametru poate fi reprezentat printr-un număr mult mai mic de biți, reprezentând practic numărul valorii parametrului din șirul valorilor posibile. Un parametru care spre exemplu poate lua 4 valori diferite poate fi reprezentat folosind doar 2 biți de informație deoarece valorile posibile în sine sunt cunoscute. Valoarea pe 64 biți folosită este obținută prin concatenarea parametrilor reprezentați fiecare printr-un număr mic de biți.

Acestă metodă reduce vectorul de 84B folosit pentru reprezentarea explicită a parametrilor la o valoare de 8B din care sunt folosiți efectiv doar 53 din cei 64 de biți.

În cadrul inițializării acestui modul este stabilită conexiunea cu server-ul MySQL și este creat un fir de execuție pentru operațiile de inserare a rezultatelor în baza de date. În cazul în care această inițializare eșuează server-ul FOCAP va funcționa fără facilitățile puse la dispoziție de baza de date.

Acest modul pune la dispoziție o funcție pentru compresia parametrilor unei simulări și pentru căutarea rezultatului unei simulări în baza de date. Folosind această funcție modulul de distribuție preia direct din baza de date rezultatele care sunt găsite și nu le mai trimite clienților pentru simulare. Modulul de distribuție are de asemenea nevoie să insereze în baza de date rezultatele nou obținute. Pentru a evita așteptarea operației de inserare aceasta nu este executată de modulul de distribuție unde performanța și viteza de reacție sunt foarte importante. Din acest motiv a fost implementată o coadă FIFO pentru rezultate care trebuie inserate. Firul de execuție al modulului de față preia din această coadă rezultatele și efectuează operațiile de inserare în paralel cu execuția altor module.

În cadrul firului de execuție sunt urmați pașii:

1. Așteaptă să fie declanșat un eveniment care indică prezența unui element în coada FIFO de inserare
2. Se extrage un element din coada FIFO
3. Se inserează în baza de date simularea și rezultatul din elementul extras
4. Dacă mai sunt elemente în coada FIFO se trece la pasul 2
5. Se trece la pasul 1

3.4.2. Implementarea algoritmilor de optimizare

Pentru optimizarea microarhitecturală au fost implementați trei algoritmi în cadrul server-ului FOCAP. Acești algoritmi sunt Hill Climbing, Simulated Annealing și un algoritm genetic. Microarhitectura este optimizată din punct de vedere al performanței, acești algoritmi

urmărind minimizarea numărului de ciclii procesor necesari execuției unui set de benchmark-uri.

Implementările acestor algoritmi salvează rezultatele optimizării în două fișiere de tip `csv`. Primul fișier conține toate configurațiile microarhitecturale care au fost simulate iar al doilea fișier conține doar configurațiile cele mai bune găsite în fiecare iterație a algoritmului. Rezultatele parțiale pot fi vizualizate în aceste fișiere în timp ce algoritmii de optimizare își desfășoară activitatea.

Algoritmii de optimizare sunt implementați în așa fel încât orice parametru microarhitectural poate avea o valoare fixă setată de utilizator. Aceste informații sunt reprezentate printr-un vector de octeți, o valoare 0 semnificând că parametrul corespunzător nu este o valoare fixă, deci poate fi modificat de algoritmul de optimizare.

3.4.2.1. Algoritmul Hill Climbing

Implementarea algoritmului Hill Climbing urmărește descrierea teoretică a acestuia din subcapitolul 2.3.1. Spre deosebire de algoritmul Hill Climbing standard implementarea pentru server-ul FOCAP a trebuit dezvoltată în așa fel încât să poată beneficia de posibilitatea simulării paralele și distribuite. Folosind facilitățile puse la dispoziție de modulul de distribuție descris în capitolul 3.4.1.3 algoritmul Hill Climbing implementat construiește seturi de simulări care conțin toate configurațiile microarhitecturale care vor fi evaluate în iterația curentă. Aceste configurații sunt *vecinii superiori* (vecini în care un parametru a trecut la o treaptă superioară) și *vecinii inferiori* (vecini în care un parametru a trecut la o treaptă inferioară) ai configurației microarhitecturale actuale.

Algoritmul se oprește în momentul în care a fost atins numărul maxim de iterații sau când nu se mai poate obține o îmbunătățire. Configurația microarhitecturală inițială este specificată folosind interfața grafică.

Algoritmul Hill Climbing implementat funcționează urmând pașii:

1. Se creează un nou *work unit* (un set de simulări)

2. Se adaugă setului de simulări toți vecinii superiori și toate benchmark-urile aferente
3. Se adaugă setului de simulări toți vecinii inferiori și toate benchmark-urile aferente
4. Este expediat setul de simulări și se așteaptă rezultatele
5. Se parcurg rezultatele și se obține astfel configurația cea mai bună
6. Configurația actuală devine configurația găsită la pasul 5
7. Se trece la pasul 1 pentru a începe o nouă iterație dacă nu a fost îndeplinită o condiție de oprire

3.4.2.2. Algoritmul Simulated Annealing

Algoritmul Simulated Annealing a fost implementat urmărind conceptele teoretice prezentate în capitolul 2.3.2. La fel ca în cazul implementării algoritmului Hill Climbing și algoritmul Simulated Annealing a trebuit implementat în așa fel încât să poată utiliza facilitățile de execuție paralelă puse la dispoziție de sistemul FOCAP prin intermediul modulului de distribuție. Acest mod de implementare a presupus din nou construirea unor seturi de simulări care să poată fi executate în paralel. În cazul algoritmului Simulated Annealing seturile menționate sunt formate din simulări generate pornind de la benchmark-urile folosite în cadrul procesului de optimizare aplicate unor vecini aleatori ai configurației actuale.

Parametrii algoritmului (temperatura inițială, temperatura minimă și numărul de iterații) sunt configurabili, iar configurația microarhitecturală inițială este specificată folosind interfața grafică. Funcția de răcire aleasă este cea exponențială cu un factor de răcire calculat în așa fel încât temperatura minimă să fie atinsă în numărul de iterații specificat. Acest factor este calculat folosind următoarea formulă:

$$c = e^{\ln(T_{min}/T_{max})/(N-1)}, \qquad \text{unde:}$$

c este factorul de răcire

T_{min} este temperatura minimă

T_{max} este temperatura maximă (inițială)

N este numărul de iterații

Funcția de acceptare implementată este funcția *P* descrisă în capitolul 2.3.2.2.

Condițiile de oprire ale algoritmului sunt atingerea numărului maxim de iterații sau lipsa îmbunătățirilor după un anumit număr de iterații. Acest algoritm funcționează în modul următor:

1. Se creează un nou set de simulări
2. Se adaugă setului simulări generate pornind de la vecini aleatori ai configurației curente
3. Este expediat setul de simulări și se așteaptă rezultatele
4. Se parcurg rezultatele; configurațiile mai bune sunt acceptate, configurațiile inferioare sunt acceptate cu probabilitatea dată de funcția de acceptare P
5. Ultima configurație acceptată devine configurația actuală
6. Se reduce temperatura: $T \leftarrow T \cdot c$
7. Se trece la pasul 1 pentru a începe o nouă iterație dacă nu a fost îndeplinită o condiție de oprire

3.4.2.3. Algoritmul genetic

Algoritmul genetic implementat realizează o optimizare mono-obiectiv, obiectivul fiind obținerea unei configurații cât mai performantă. Având în vedere faptul că este necesară evaluarea întregii populații înainte de a începe procesul de selecție, algoritmul genetic permite paralelizarea ușoară a etapei de simulare. În cadrul acestei implementări se creează un set de simulări care conține toți indivizii din populația actuală. Numărul de simulări din acest set depinde atât de numărul de indivizi din populație cât și de numărul de benchmark-uri folosite pentru evaluarea performanței.

Configurațiile microarhitecturale au fost reprezentate ca valori întregi pe 64 biți folosind sistemul de compresie descris în capitolul 3.4.1.5. Această reprezentare permite implemetarea operatorilor de încrucișare și mutație pe biți. Operatorul de încrucișare implementat folosește un singur punct generat aleator pentru a realiza încrucișarea, iar operatorul de mutație parcurge reprezentarea unui individ și cu o anumită probabilitate inversează bitul curent.

Metoda de selecție implementată este *stochastic universal sampling*. Această metodă a fost aleasă pentru a menține diversitatea populației astfel încât indivizii cu fitness mic să primească o șansă mai bună de a fi selectați. De asemenea a fost implementat un mecanism de elitism pentru a putea garanta că individul cel mai bun nu se pierde.

Algoritmul este configurabil folosind interfața grafică. Parametrii care pot fi modificați sunt probabilitatea de încrucișare, probabilitatea de mutație, numărul minim de generații, numărul maxim de generații și opțiunea de elitism. În acest caz populația inițială este generată aleator, respectând valorile setate de utilizator pentru parametrii setați fix.

Algoritmul genetic urmărește pașii:

1. Se generează aleator o populație
2. Se creează un set de simulări
3. Se adaugă în set toate simulările necesare pentru a evalua întreaga populație
4. Se expediază setul de simulări și se așteaptă rezultatele
5. Folosind rezultatele se realizează procesul de selecție părinților
6. Se aplică operatorul de încrucișare cu probabilitatea specificată
7. Fiecărui individ rezultat din încrucișare i se aplică operatorul de mutație
8. Populația nouă este formată din indivizii rezultați din pașii 5-7
9. Dacă nu a fost îndeplinită o condiție oprire se trece la pasul 2 pentru a începe următoarea generație

4. Rezultate

În subcapitolele 4.1-4.4 vor fi prezentate rezultatele obținute prin aplicarea celor trei algoritmi de optimizare implementați unei microarhitecturi evaluate cu simulatorul HSA dezvoltat. În capitolul 4.5 este prezentat un rezultat preliminar obținut aplicând algoritmul Hill Climbing unei arhitecturi SimpleScalar simulată folosind Sim-Outorder.

4.1. Rezultate obținute cu Hill Climbing

Algoritmul Hill Climbing a fost rulat pornind de la o configurație generată aleator iar singurii parametri cu valori fixe au fost latențele caracteristice sistemului de memorie și ale unităților de înmulțire respectiv împărțire. Configurația inițială a fost:

Parametru	**Valoare inițială**	**Valoare fixă**
Latență memorie principală	30	Da
Latență memorie cache	1	Da
Latență înmulțire	4	Da
Latență împărțire	12	Da
Seturi cache instrucțiuni	2	Nu
Dimensiune set cache instrucțiuni	4	Nu
Seturi cache date	1	Nu
Dimensiune set cache date	1	Nu
Dimensiune linie cache date	32	Nu
Dimensiune buffer instrucțiuni	16	Nu
Dimensiune buffer date	8	Nu
Fetch rate	4	Nu
Issue Rate Maxim	16	Nu
Unități execuție relaționale	3	Nu
Unități execuție aritmetice	7	Nu
Unități execuție deplasare	7	Nu
Unități execuție înmulțire	3	Nu
Unități execuție împărțire	4	Nu
Unități execuție pentru scriere în memorie (store)	8	Nu
Unități execuție pentru citire din memorie (load)	2	Nu

În figurile 4.1 și 4.2 sunt prezentate rezultatele optimizării Hill Climbing. Figura 4.1 prezintă toate configurațiile care au fost evaluate în ordinea în care acestea au fost. Figura 4.2 prezintă doar performanța celei mai bune configurații din fiecare iterație Hill Climbing.

Cea mai bună configurație a avut nevoie de un număr mediu de 145241 de ciclii procesor pentru a rula cele 8 benchmark-uri Stanford. Algoritmul Hill Climbing s-a oprit după 17 iterații.

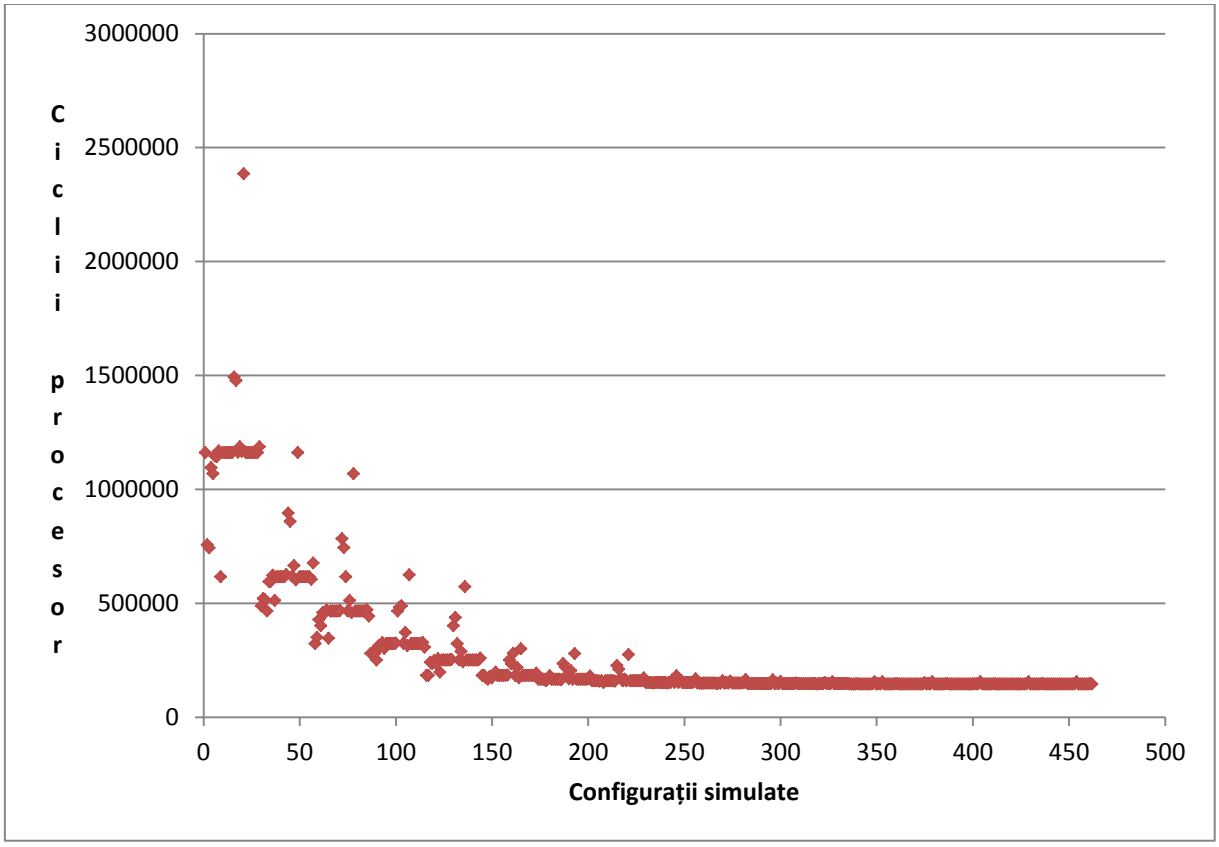

Figura 4.1 Optimizare folosind Hill Climbing

Figure 4.2 Performanța celei mai bune configurații la fiecare iterație Hill Climbing

4.2. Rezultate obținute cu Simulated Annealing

Algoritmul Simulated Annealing a pornit de la aceeași configurație ca algoritmul Hill Climbing. În figurile 4.3 și 4.4 sunt prezentate toate configurațiile întâlnite de Simulated Annealing respectiv configurațiile actuale pentru fiecare iterație.

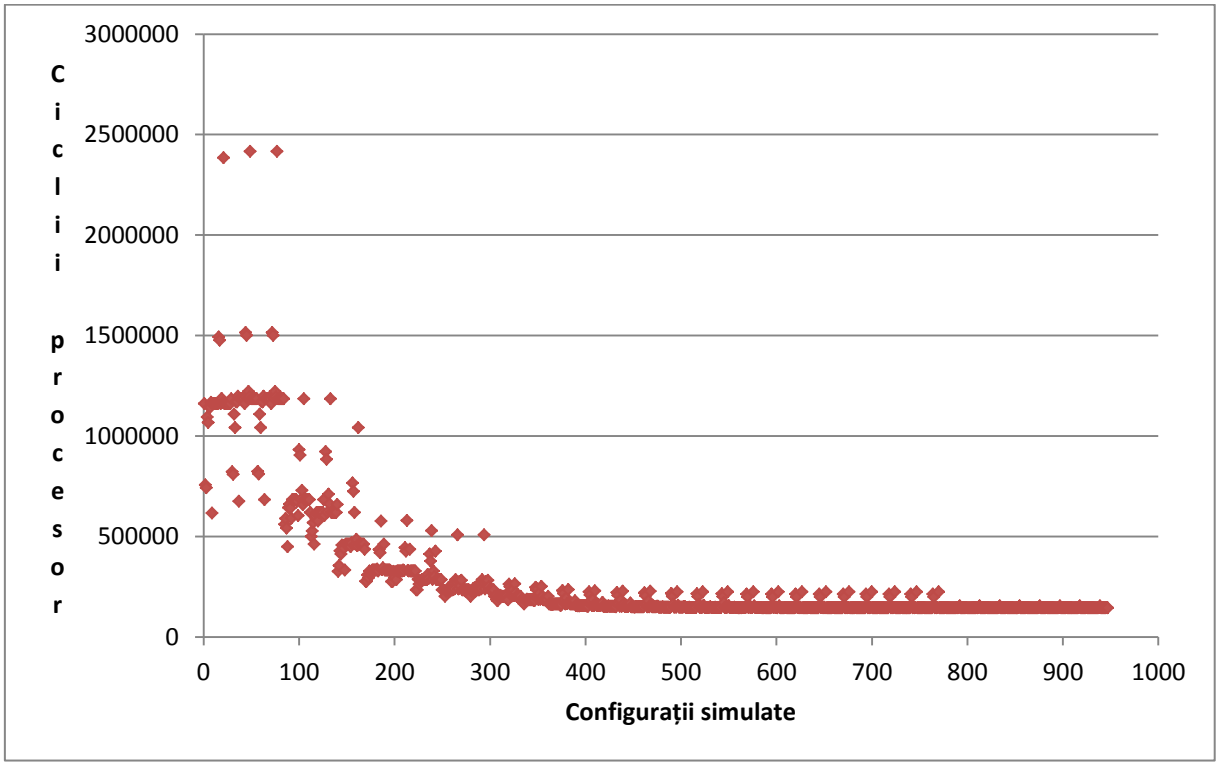

Figura 4.3 Optimizare folosind Simulated Annealing

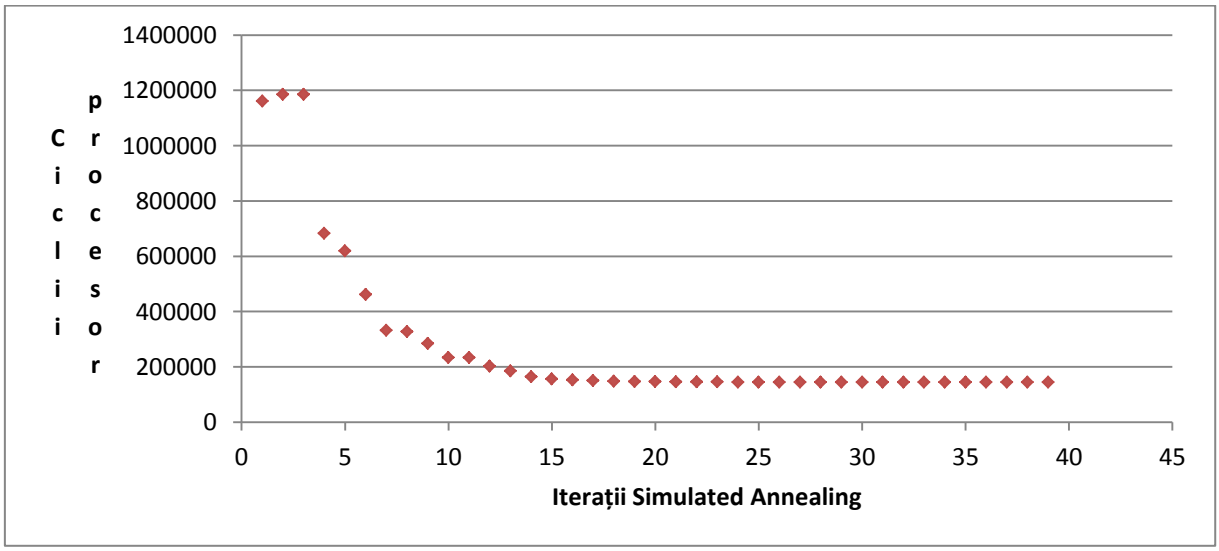

Figura 4.4 Performanța configurației actuale la fiecare iterație Simulated Annealing

Cea mai bună configurație a avut nevoie de un număr mediu de 145199 de ciclii procesor pentru a rula cele 8 benchmark-uri Stanford, observându-se o mică îmbunătățire față de rezultatul obținut cu Hill Climbing. Algoritmul s-a oprit după 37 de iterații deoarece configurația curentă nu s-a mai schimbat pentru 5 iterații succesive.

4.3. Rezultate obținute cu algoritmul genetic

Algoritmul genetic rulat a folosit o populație de 100 de indivizi cu o probabilitate de încrucișare de 85% și o probabilitate de mutație de 0.5%. De asemenea algoritmul a fost configurat în așa fel încât să funcționeze într-un mod elitist, iar numărul minim de generații evaluate a fost 15, numărul maxim fiind 30. Populația inițială a fost generată aleator, dar pentru a putea compara rezultatele cu cele obținute cu Hill Climbing și Simulated Annealing au fost setate manual aceleași latențe. Figurile 4.5 și 4.6 prezintă rezultatele obținute. În figura 4.5 este prezentată performanța tuturor indivizilor în ordinea în care aceștia au fost întâlniți iar figura 4.6 reprezintă performanța celor mai buni indivizi din fiecare generație.

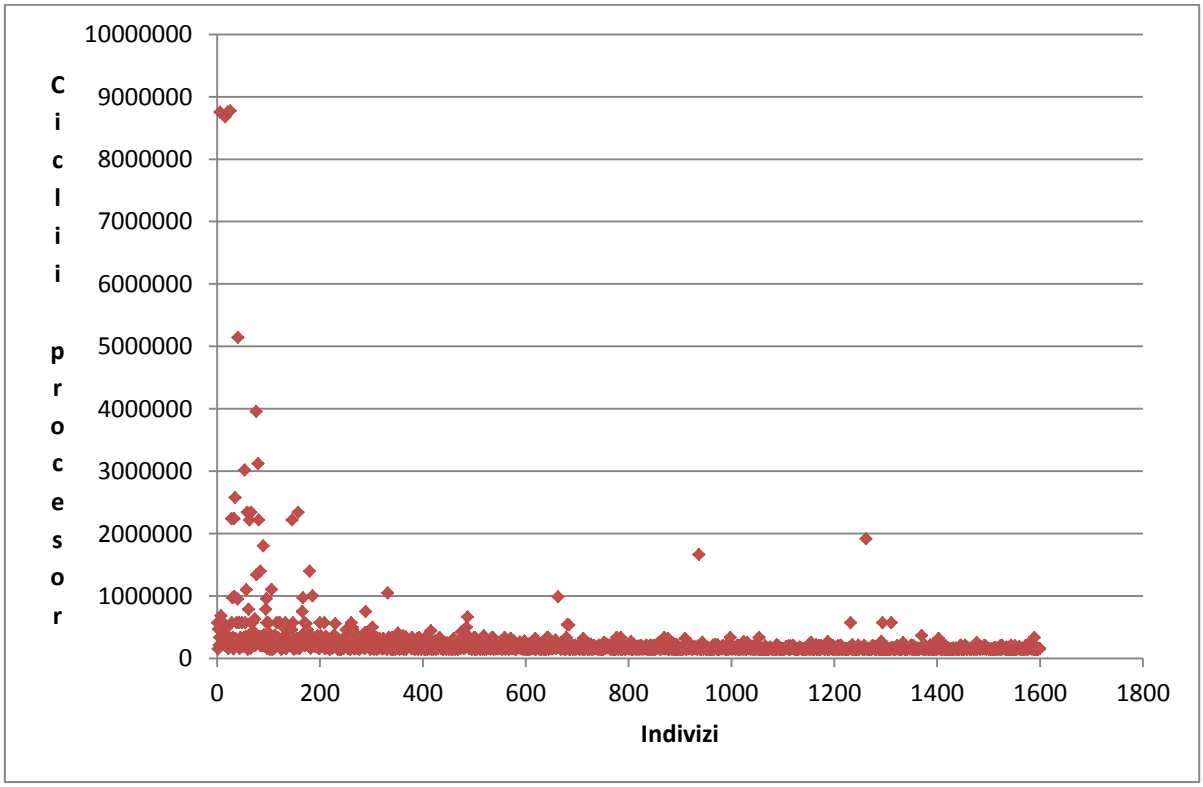

Figura 4.5 Optimizare folosind algoritmul genetic

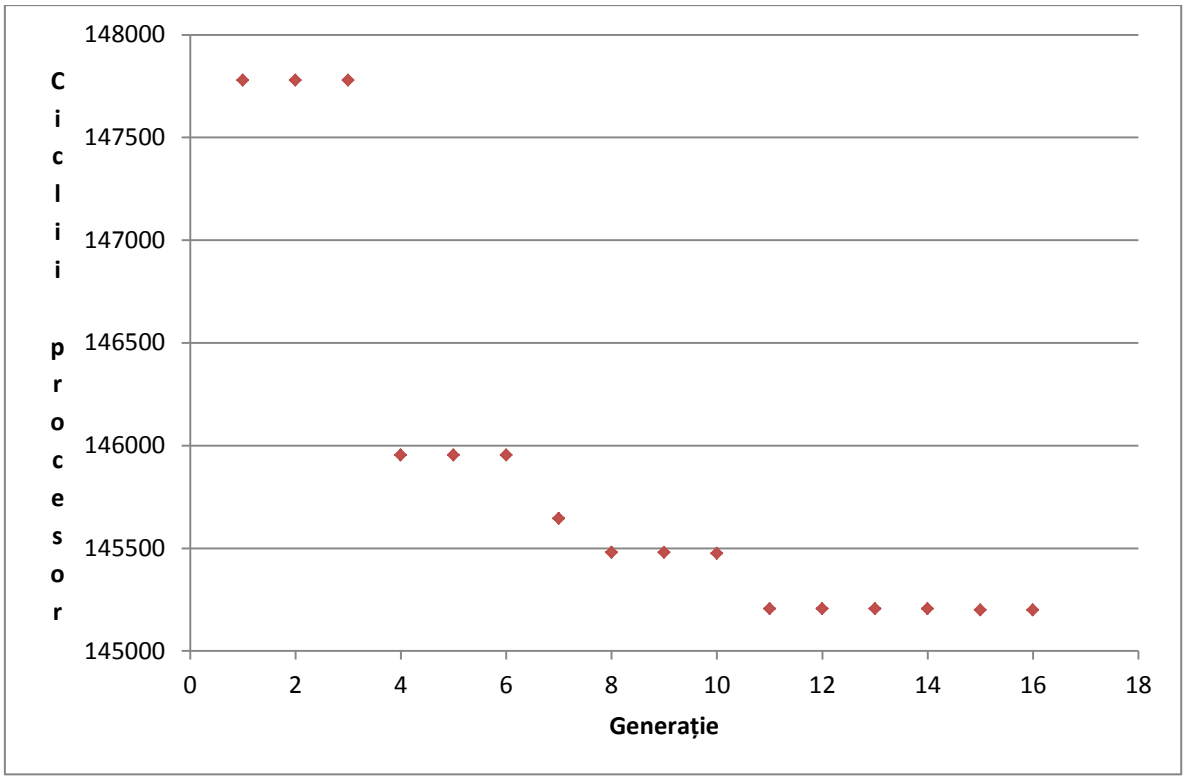

Figura 4.6 Performanța celui mai bun individ din fiecare generație

Algoritmul genetic a obținut o configurație cu aceeași performanță ca algoritmul Simulated Annealing (145199 ciclii).

4.4. Comparație a configurațiilor obținute

În tabelul următor sunt prezentate configurațiile obținute cu cei trei algoritmi aplicați. Se poate observa că în cazul utilizării simulatorului HSA împreună cu benchmark-urile Stanford de mică dimensiune chiar și algoritmul Hill Climbing are rezultate bune și foarte apropiate de algoritmul genetic și de Simulated Annealing. De asemenea în ciuda diferențelor dintre configurațiile obținute acestea dau dovadă de performanță aproape identică. Totuși se poate observa că unii parametri au un impact mai important asupra performanței iar acești parametri iau valori identice în toate cele trei configurații (valorile fetch rate, issue rate, dimensiunea buffer-ului de instrucțiuni, numărul de unități de scriere și citire din memorie). Pentru a observa diferențe mai mari între cei trei algoritmi ar trebui utilizat un set mai complex de benchmark-uri și un simulator mai performant.

Parametru	Hill Climbing	Simulated Annealing	Algoritm Genetic
Latență memorie principală	30	30	30
Latență memorie cache	1	1	1
Latență înmulțire	4	4	4
Latență împărțire	12	12	12
Seturi cache instrucțiuni	16	4	4096
Dimensiune set cache instrucțiuni	4	8	8
Seturi cache date	4	8	16384
Dimensiune set cache date	8	8	1
Dimensiune linie cache date	32	64	64
Dimensiune buffer instrucțiuni	64	64	64
Dimensiune buffer date	2	2	2
Fetch rate	16	16	16
Issue Rate Maxim	16	16	16
Unități execuție relaționale	3	6	7
Unități execuție aritmetice	7	8	6
Unități execuție deplasare	7	8	3
Unități execuție înmulțire	3	3	1
Unități execuție împărțire	4	4	6
Unități execuție pentru scriere în memorie (store)	8	8	8
Unități execuție pentru citire din memorie (load)	8	8	8
Număr mediu ciclii	***145241***	***145199***	***145199***

Configurațiile obținute

4.5. Rezultate Hill Climbing obținute cu Sim-Outorder

La fel ca în cazul utilizării simulatorului HSA algoritmul Hill Climbing a pornit de la o configurație microarhitecturală generată aleator. Latențele sistemului de memorie au fost de asemenea considerate valori fixe. Configurația inițială a fost:

Parametru	**Valoare inițială**	**Valoare fixă**
Latență memorie principală	40	Da
Latență memorie cache	2	Da
Seturi cache instrucțiuni	256	Nu
Dimensiune set cache instrucțiuni	4	Nu
Seturi cache date	16	Nu
Dimensiune set cache date	4	Nu
Dimensiune linie cache date	32	Nu
Dimensiune buffer instrucțiuni	8	Nu
Dimensiune buffer date	4	Nu
Fetch rate	8	Nu
Issue Rate Maxim	2	Nu
Număr ALU	6	Nu
Unități înmulțire/împărțire	7	Nu
Unități de acces la memorie	2	Nu

În figurile 4.7 și 4.8 sunt prezentate rezultatele obținute. Figura 4.7 reprezintă performanța tuturor configurațiilor microarhitecturale simulate iar în figura 4.8 sunt prezentate rezultatele celor mai bune configurații din fiecare iterație Hill Climbing.

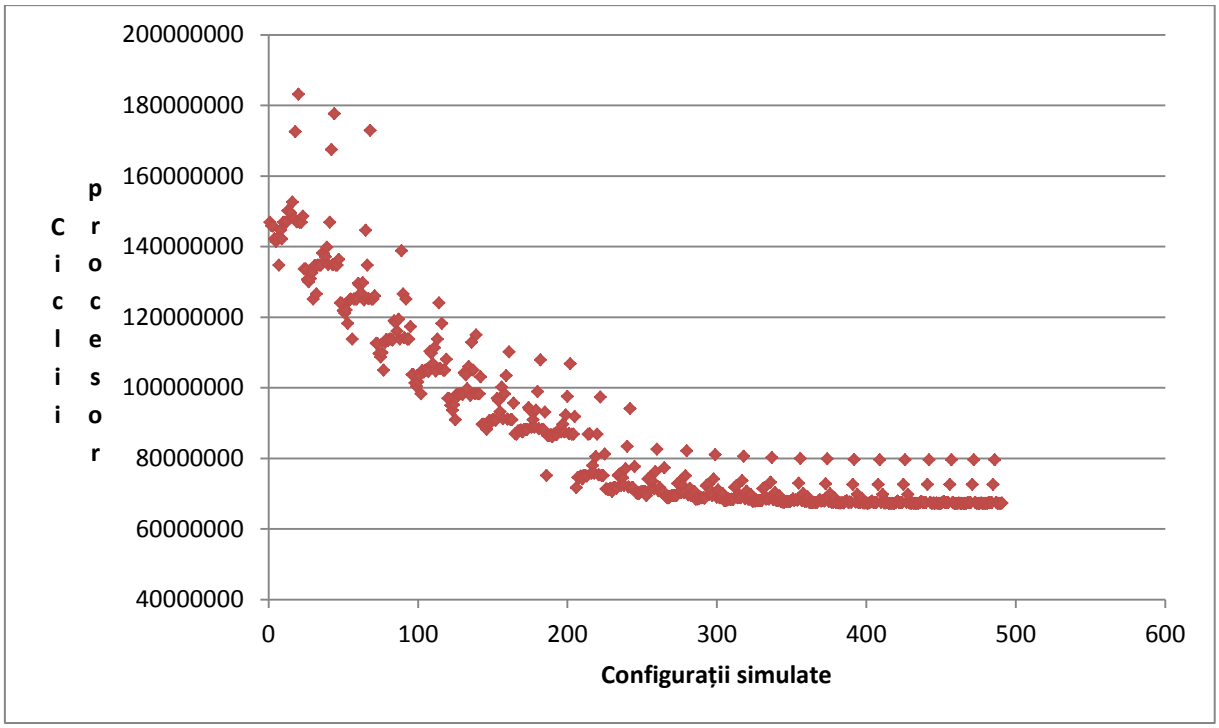

Figura 4.7 Optimizare folosind Hill Climbing și Sim-Outorder

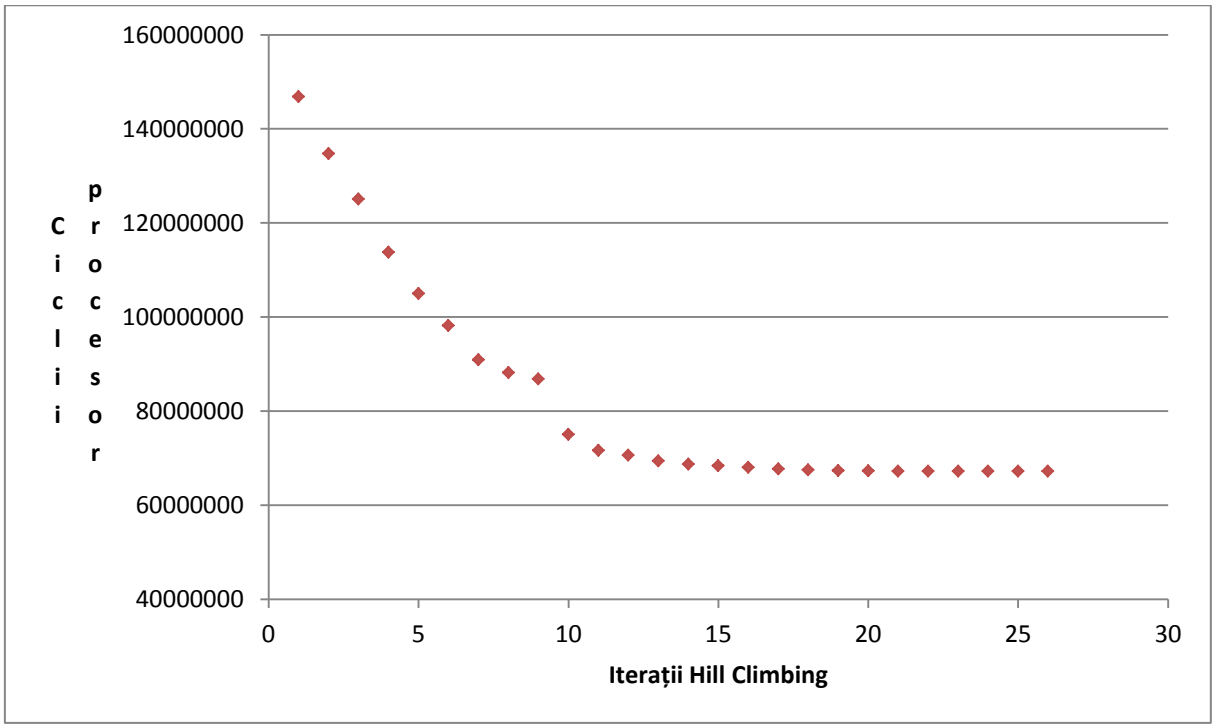

Figura 4.8 Performanța celei mai bune configurații la fiecare iterație Hill Climbing folosind Sim-Outorder

Algoritmul Hill Climbing a obținut în 25 de iterații de iterații o configurație care a rulat setul de benchmark-uri într-un număr mediu de 67251896 de ciclii procesor, cea inițială având nevoie de 146892523 de ciclii. Astfel s-a obținut o creștere IPC de la 0.68 la 1.486. Configurația finală a fost:

Parametru	**Valoare**
Latență memorie principală	40
Latență memorie cache	2
Seturi cache instrucțiuni	4096
Dimensiune set cache instrucțiuni	8
Seturi cache date	128
Dimensiune set cache date	8
Dimensiune linie cache date	64
Dimensiune buffer instrucțiuni	64
Dimensiune buffer date	16
Fetch rate	16
Issue Rate Maxim	16
Număr ALU	8
Unități înmulțire/împărțire	8
Unități de acces la memorie	8

5. Concluzii și dezvoltări ulterioare

Dezvoltarea cadrului de optimizare FOCAP a pornit doar de la un proiect pentru disciplina Simularea și Optimizarea Arhitecturilor de Calcul dar a devenit o aplicație mult mai complexă și flexibilă capabilă să optimizeze performanța mai multor arhitecturi de calcul folosind atât un simulator propriu dezvoltat pentru arhitectura HSA cât și simulatorul extern Sim-Outorder pentru arhitectura SimpleScalar.

Rezultatele obținute folosind simulatorul HSA trace-driven propriu și setul de benchmark-uri Stanford au dezvăluit necesitatea utilizării unor benchmark-uri mai complexe, cu mai multe

instrucțiuni, și a unui simulator mai performant. Rezultatele simulărilor care au folosit benchmark-urile Stanford au dezvăluit existența a multor configurații diferite cu performanță aproape identică, fapt cauzat de simplitatea benchmark-urilor și a numărului redus de instrucțiuni ale acestora. Simulatorul folosit ca înlocuitor este numit Sim-Outorder și este un un simulator execution-driven pentru arhitectura SimpleScalar care folosește benchmark-uri mult mai lungi din care sunt executate doar 100000000 de instrucțiuni. Totuși, chiar folosind acest număr redus de instrucțiuni din cauza constrângerilor de timp nu a fost posibilă prezentarea unor rezultate complete obținute folosind acest simulator în cadrul aceste lucrări.

Proiectarea cadrului de optimizare în așa fel încât dependența dintre simulatorul folosit și sistemul FOCAP să fie minimă și-a dovedit eficiența prin integrarea ușoară a simulatorului extern SimpleScalar Sim-Outorder. Pentru utilizarea Sim-Outorder a fost necesară doar implementarea unui simplu modul de adaptare a datelor de intrare și ieșire și de gestionare a execuției simulatorului, restul sistemului FOCAP funcționând fără să-i fie adusă nici o modificare.

În momentul de față procesul de optimizare vizează doar performanța microarhitecturii studiate. Pentru a obține configurații cu aplicabilitate practică mai mare se dorește extinderea optimizării la una de tip multiobiectiv și implementarea algoritmilor aferenți. Pentru a simula puterea consumată de procesor instrumente externe precum CACTI vor fi integrate iar informația pusă la dispoziție de acestea va fi folosită ca un al doilea obiectiv de optimizare [2, 8]. De asemenea se dorește implementarea unui sistem care să permită descrierea unei microarhitecturi folosind fișiere de configurare externe pentru a evita necesitatea modificării structurilor folosite în implementarea curentă. Pentru a permite adăugarea simplă de noi algoritmi de optimizare server-ului FOCAP modulul de optimizare va fi extins în așa fel încât să permită încărcarea algoritmilor din fișiere .dll externe, fără a fi necesară modificarea codului sursă al acestuia.

Un ultim obiectiv propus este portarea atât a clientului cât și a server-ului FOCAP pe sisteme de operare bazate pe Linux și pe sisteme HPC care ar permite o viteză de simulare mult mai mare.

BIBLIOGRAFIE

[1] K. de Bosschere et al., "High-Performance Embedded Architecture and Compilation Roadmap", *Transactions on HiPEAC I, Lecture Notes in Computer Science 4050, Springer-Verlag*, 2007, pp 5-29.

[2] Brooks D., Tiwari V., Martonosi M., "Wattch: a framework for architectural-level power analysis and optimizations". *Proceedings of International Symposium on Computer Architecture*, pp. 83–94, 2000.

[3] H. Calborean, R. Jahr, T. Ungerer, L. Vintan, "Optimizing a Superscalar System using Multi-objective Design Space Exploration", *Proceedings of the 18th International Conference on Control Systems and Computer Science*, Bucharest, Romania, May 2011, pp. 339-346.

[4] H. Calborean, "Multi-Objective Optimization of Advanced Computer Architectures using Domain-Knowledge", PhD Thesis, "Lucian Blaga" University of Sibiu, Sibiu, Romania, 2011.

[5] V. Desmet, A. Ramirez, O. Temam, A. Vega, "ArchExplorer for Automatic Design Space Exploration". IEEE Micro 30(5): 5-15, 2010.

[6] A. Florea, L. Vintan, "Simularea și optimizarea arhitecturilor de calcul în aplicații practice", *Editura MatrixRom*, Bucuresti, 2003, ISBN 973-685-605-4.

[7] S. Kang, R. Kumar, "Magellan: a search and machine learning-based framework for fast multi-core design space exploration and optimization", *in Proceedings of the Conference on Design, Automation and Test in Europe*, Germany, 2008, pp. 1432-1437.

[8] N. Muralimanohar, R. Balasubramonian, N.P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches", HP Laboratories, Technical Report HPL-2009-85, Utah, 2009.

[9] G. Palermo, V. Zaccaria, G. Mariani, F. Castro, "Multicube Explorer User Manual Release 1.1", Politecnico di Milano, Italy, 2010.

[10] T. Taghavi, A.D. Pimentel, "VMODEX: A visualization tool for multi-objective Design Space Exploration", *International Conference on Field-Programmable Technology (FPT)*, 2010, pp. 357 - 360.

[11] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, „Optimization by Simulated Annealing", Science, New Series, Vol. 220, No. 4598. (May 13, 1983), pp. 671-680

[12] Moore A., "*Iterative Improvement Search*: *Hill Climbing, Simulated Annealing, WALKSAT, and Genetic Algorithms*", 2002, http://www.autonlab.org/tutorials/hillclimb02.pdf

[13] H. Calborean, „An overview of the multiobjective optimization methods", University of Sibiu, Sibiu, Romania, 2010

[14] D. Whitley, „A Genetic Algorithm Tutorial", Computer Science Department, Colorado State University

[15] F. Busetti, „Simulated annealing overview", http://163.18.62.64/wisdom/Simulated%20annealing%20overview.pdf

[16] C. Brachem, K. Kröninger, „Implementation and test of a simulated annealing algorithm in the Bayesian Analysis Toolkit (BAT)", Fakultät für Physik, Georg-August-Universität Göttingen, 2009