Proiect la Simularea si Optimizarea Arhitecturilor de Calcul

Autor: Barbu Paul – Gheorghe
Semigrupa: 242/1
Indrumator: Prof.univ.dr.ing. Adrian FLOREA

Enunt

Implementarea unui Automatic Design Space Exploration aferent unei arhitecturi superscalare (vezi simulatorul PSATSim – arhitectura PowerPC), pentru analiza multiobiectiv (performanță, consum de energie) folosind tehnici de optimizare Pareto: Multi Objective Particle Swarm Optimization.

Resurse necesare

Sistemul de operare: Windows, GNU/Linux

Pentru a rula aplicatia avem nevoie de:

- JRE 8.0
- simulatorul PSATSim, acesta necesita la randul sau bibliotecile:
  - libgtk+
  - zlib1
  - iconv
  - libxml2

Pentru a modifica aplicatia avem nevoie de:

- JDK SE 8.0
- Limbajul Scala 2.11.8
- Mediul de dezvoltare Scala IDE
- Biblioteca Scala XML 1.0.3
- Biblioteca JFreeChart 1.0.19
- Codul sursa al bibliotecii jMetal 4.5 (inclus deja in proiect)

Particle Swarm Optimization

Algoritmul de Particle Swarm Optimization este o tehnica din categoria “Artificial Life” prin care o populatie de indivizi urmareste optimul prin experienta colectiva cat si experienta individuala.

In cazul optimizarii unui procesor, dorim sa optimizam numarul de instructiuni rulate raportate la ciclii de procesor si energia consumata, deci dorim performanta maxima cu energie minima, asadar ne trebuie un algoritm pentru optimizare multiobiectiv (doua obiective in acest caz). Algoritmul PSO se aplica foare bine pe probleme de optimizare multiobiectiv de minimizare.

Asadar primul nostru obiectiv, va trebui inversat, in loc de maximizarea IPC-ului (instructions per cycle), va trebui sa minimizam CPI (cycles per instruction). Acest lucru nu reprezinta o problema, doarece $CPI = \frac{1}{IPC}$.

In linii mari algoritmul PSO, modeleaza miscarea unui stol de pasari sau al unui banc de pesti care urmaresc sa ajunga la o destinatie.

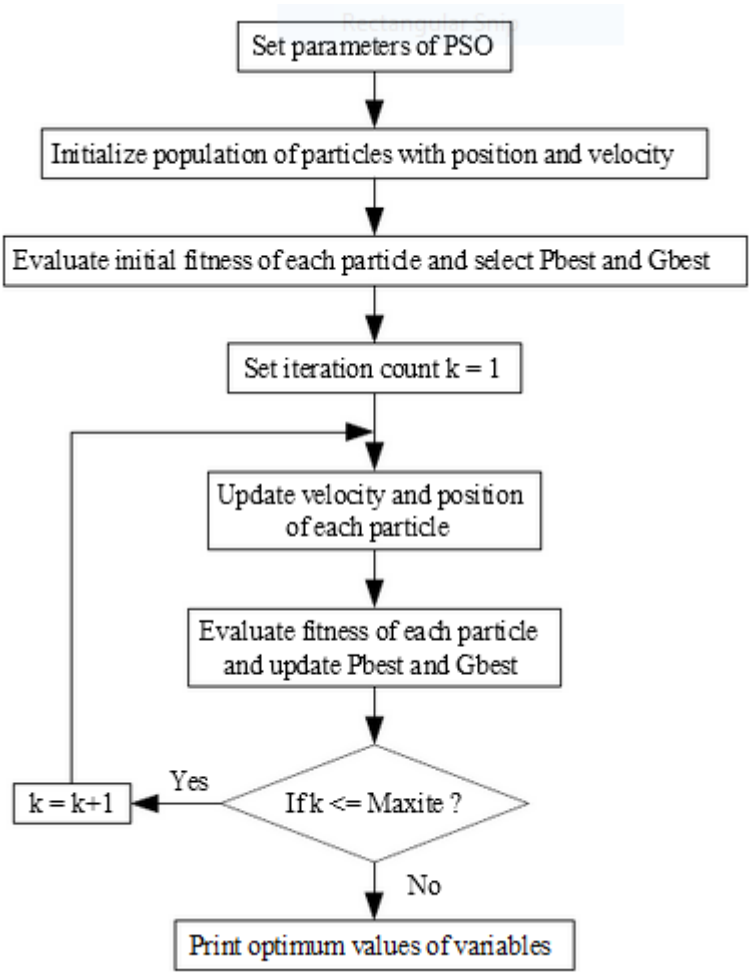
In mod plastic, in cazul optimizarii procesoarelor, vom defini “destinatia” ca fiind punctul din planul XOY unde ambele obiective, CPI si energia, sunt minime. Asadar, punctul (0, 0).

In pseudocod algoritmul ar fi:

```
initializare aleatoare a particulelor din populatie

cat-timp nr. maxim de iteratii nu a fost atins sau hipervolumul nu s-a modificat
semnificativ

      pentru fiecare particula

            evaluare particula

            daca locatia particulei este cea mai apropiata de obiectiv

                  retinere configuratie particula ca personal-best

            sf-daca

      sf-pentru

      global-best = maxim personal-best

      pentru fiecare particula

            schimbare viteza in functie de personal-best si global-best

            schimbare locatie in functie de locatia anterioara si noua viteza
      sf-pentru

sf-cat-timp
```

Se poate observa ca fiecare particula tine cont atat de experienta celorlalte particule din populatie cat si de experienta proprie, datorita faptului ca viteza se schimba atat in functie de cea mai buna configuratie gasita la nivel global cat si personal.

Schema logica a algoritmului:



In realitate, programul foloseste un algoritm PSO usor modificat: SMPSO, detaliat de autorii jMetal pe pagina: http://jmetal.sourceforge.net/smpso.html.

Pe scurt, pseudocodul pentru acest algoritm este:

```
1: initializeSwarm()
2: initializeLeadersArchive()
3: generation = 0
4: while generation < maxGenerations do
5:    computeSpeed()
6:    updatePosition()
7:    mutation() // Turbulence
8:    evaluation()
9:    updateLeadersArchive()
10:   updateParticlesMemory()
11: generation ++
12: end while
13: returnLeadersArchive()
```

In urma studierii pseudocodului apare necesitatea de a echivala configuratia unei particule cu configuratia unui procesor. Acest lucru va fi descris in ghidul de dezvoltare, luand in considerare si notiunile din sectiunea urmatoare.

Despre simulatorul PSATSim

Fiind dificil sa implementam fizic sute, chiar mii de procesoare evaluate de-a lungul rularii algoritmului SMPSO se foloseste un simulator care ne ofera posibilitatea configurarii parametrilor procesorului si evaluarea acestuia din punct de vedere al IPC-ului si al energiei.

Dintre toti parametrii pe care ii ofera simulatorul exploratorul nostru va modifica doar:

Nume	Domeniu	Descriere
superscalar	Integers 1-16	Superscalar factor
rename	Integers 1-512	Register renaming table size
reorder	Integers 1-512	Reorder buffer size
RSB	centralized or hybrid or distributed	Reservation station architecture
Separate dispatch	true or false	True = decode and dispatch stages are separated.
vdd	Floating point greater than zero	Processor supply voltage. Should kept in the range of 1.8-3.3.
freq	Floating point greater than zero	Processor clock frequency, in MHz
integer	Integers 1-8	The number of integer functional units
floating	Integers 1-8	The number of floating point functional units
branch	Integers 1-8	The number of branch functional units
memory	Integers 1-8	The number of memory functional units

Restul de parametrii vor ramane constanti, precum se va arata in ghidul de dezvoltare.

Ghid utilizare

In urma compilarii software-ului utilizatorul va trebui sa porneasca o fereastra de consola, de unde va rula programul.

Comanda care trebuie folosita este:

```
java -jar cpuPso.jar --psatsim-name psatsim_con --psatsim-path
/cale/spre/psatsim/ --swarm-size 5 --max-iterations 10 --archive-
size 100 –-epsilon 0.001
```

Aplicatia ar trebui sa fie utilizabila atat pe Windows cat si pe GNU/Linux, iar semnificatia parametrilor este urmatoarea:

1. psatsim-name este numele executabilului aferent simulatorului PSatSim, pe Windows “psatsim_con.exe”, iar pe GNU/Linux, “psatsim_con” (fara extensie).
2. psatsim-path este calea pe hard disk unde se gaseste executabilul indicat cu parametrul anterior, exemplu: "E:/localhost/CpuParticleSwarmOptimization/psatsim/PSATSim".
3. swarm-size este numarul de indivizi ai populatiei de particule (cu alte cuvinte nr. de procesoare de simulat), pe procesoare mai slabe un numar mic este ideal, de exemplu 5, pe procesoare mai bune, puntem mari acest numar chiar si la 20-30.
4. max-iterations este numarul maxim de populatii pe care programul are voie sa le genereze, acesta ruleaza oricum pana cand hipervolumul populatiilor nu mai evolueaza semnificativ. Totusi daca numarul maxim de iteratii este atins, atunci explorarea se opreste. Daca rulam pe un sistem constrans dpdv. al resurselor si dorim sa ne oprim dupa ce s-au generat cateva populatii, atunci acest numar poate fi mic (exemplu: 50), dar daca dorim sa pastram conditia de oprire pe modificarea nesemnificativa a hipervolumului atunci etse bine sa folosim un numar mare, 1000, pentru a evita atingerea lui.
5. archive-size este un parametru specific jMetal si reprezinta numarul maxim de indivizi non-dominati de pe frontul Pareto care trebuie pastrati in timpul simularilor, asadar fisierele de output descrise mai jos nu vor putea contine mai multi indivizi fata de cat indica acest parametru. Un maxim pentru acest parametru il reprezinta swarm-size*max-iterations, desi este recomandat sa fie pastrat la 100.
6. epsilon reprezinta diferenta dintre doua hipervolume pentru a considera schimbarea nesemnificativa si deci pentru a declansa oprirea explorarii. O valoare indicata ar fi 0.001 sau 0.0001.

Output

In urma rularii cu succes a aplicatiei, se va crea un director cu data si ora curenta in formatul YYYY-MM-DD HH.MM.SS (ex. “2016-12-17 10.15.49”). S-a ales acest format datorita usurintei sortarii lui si a garantiei unicitatii numelui, deoarece o simulare putem fi siguri ca dureaza mai mult de o secunda. Acest director nu va fi creat in cazul unei rulari cu erori!

Continutul directorului este:

- fisierul FUN – care reprezinta obiectivele atinse de cei mai buni indivizi la finalul explorarii. In acest caz o linie din fisier corespunde unui individ non-dominat la finalul executiei. O line consta din doua numere de tip double care reprezinta CPI-ul unui procesor si energia consumata de acesta. Aceste numere sunt de fapt agregate ca media aritmetica pe cele 10 trace-uri livrate cu simulatorul PSATSim.
- fisierul plot-hv.png reprezinta evolutia hipervolumului de-a lungul celor max-iterations sau pana cand acesta nu a mai evoluat semnificativ (la a 3a zecimala)
- fisierul plot-solutions.png este reprezentarea grafica a indivizilor non-dominati (frontul Pareto)
- SOMPSO.log este output-ul din program, care tine logul celor mai importante evenimente, cum ar fi valoarea parametrilor, timpul de rulare, valorile hipervolumelor de-a lungul explorarii, si motivul pentru care programul s-a oprit (fie ca a atins numarul maxim de iteratii, fie ca hipervolumul nu s-a mai modificat considerabil)
- fisierul VAR arata configuratia fiecarui individ non-dominat. Cu alte cuvinte, daca luam fiecare linie din acest fisier si folosim valorile in simulatorul PSatSim, acestea vor duce la obtinerea rezultatelor de pe aceeasi linie din fisierul FUN.

Fisierele FUN si VAR ar putea arata dupa cum urmeaza:

	Continutul fisierului FUN		Continutul fisierului VAR										
#	CPI	Energy	Superscalar	Rename	Reorder	RSB	Separate dispatch	VDD	Freq	Integer FU	Floating FU	Branch FU	Memory FU
1	1.040	2.480	14	473	3	2	1	1.929	1459	3	6	4	3
2	0.286	12.455	16	85	448	2	1	2.582	1403	4	5	6	5
3	0.565	4.395	13	27	17	2	1	3.116	1469	7	5	6	1
4	0.743	2.853	13	282	6	2	1	2.366	1461	3	5	4	2
5	0.323	7.738	15	164	261	2	1	2.448	1418	3	5	5	4
6	2.486	2.274	6	512	1	2	1	1.800	1570	1	6	1	1
7	0.323	8.283	15	152	286	2	1	2.456	1413	3	5	5	4
8	2.486	1.808	1	512	1	2	1	1.800	1646	1	6	1	1
9	0.351	7.421	14	175	266	2	1	2.416	1416	3	5	4	3

Ghid de dezvoltare

In ceea ce priveste dezvoltarea aplicatiei de explorare folosind algoritmul Particle Swarm Optimization s-a folosit limbajul Scala (un dialect al Java) cat si limbajul de programare Java. Scala a fost ales din considerente de productivitate, acesta oferind un mod foarte usor pentru lucrul cu fisiere XML (mai ales scrierea acestora).

Lucrul cu fisiere XML este important, deoarece pentru a automatiza simulatorul PSATSim sunt necesare fisiere de configuratie. Aceste fisiere de configuratie sunt in format XML.
Scala ne permite sa scriem si sa parametrizam fisiere XML similar cu scrierea codului normal, programatorul neobservand vreo diferenta intre codul care aplica logica aplicatiei, algoritm, si partea de XML.

De exemplu:

```
private def getXml = {
    def generalNode(trace: String) =
      <general
            superscalar={_superscalar.toString}
            rename={_rename.toString}
            reorder={_reorder.toString}
            rsb_architecture={_rsb_architecture.toString}
            separate_dispatch={_separate_dispatch.toString}
            seed="0"
            trace={trace}
            output={output}
            vdd={"%1.2f".format(vdd)}
            frequency={freq.toString}
      />;

    // this XML will contain 10 general nodes, one for every trace file
    <psatsim>
      <config name={name}>
            {traces map(t => generalNode(t))}
            <execution
                  architecture="standard"
                  integer={integerFu.toString}
                  floating={floatingFu.toString}
                  branch={branchFu.toString}
                  memory={memoryFu.toString}
            />
            <memory architecture="12">
                  <l1_code hitrate="0.990" latency="1" />
                  <l1_data hitrate="0.970" latency="1" />
                  <l2 hitrate="0.990" latency="3" />
                  <system latency="20" />
            </memory>
       </config>
    </psatsim>
  }
```

Aceasta metoda privata a clasei PSATSimCfg se ocupa cu crearea (in memorie) a intregului fisier de configuratie necesar rularii simulatorului. Observam in prima jumatate a metodei, o functie

imbricata, care genereaza o configuratie generala pentru fiecare fisier de trace in parte. Aceasta functie este folosita in a doua parte a metodei pentru a popula nodul “config” al XML-ului.

Se poate observa modul in care parametrii precizati la sectiunea “Despre simulatorul PSATSim” sunt configurabili din exterior, iar restul, sunt constanti, acestia nefiind influentati de algoritm.

In urma generarii continutului acesta va fi scris pe disc de metoda “save” a aceleasi clase:

```
def save(path: String) = {
    XML.save(Paths.get(path, name + ".xml").toString, getXml)
  }
```

O particula sau un individ creat de algoritmul PSO va fi o instanța a clasei CpuSolutionType. Instanța face maparea efectiva a unui procesor în reprezentarea specifica PSO.

```
override def createVariables(): Array[Variable] = {
    val variables = new Array[Variable](11)
    variables(0) = new Int(1, 16) // superscalar
    variables(1) = new Int(1, 512) // rename
    variables(2) = new Int(1, 512) // reorder
    variables(3) = new Int(1, 2) // RSB
    variables(4) = new Int(0, 1) // separate dispatch
    variables(5) = new Real(1.8, 3.3) // vdd
    variables(6) = new Int(10, 5000) // freq
    variables(7) = new Int(1, 8) // interger FU
    variables(8) = new Int(1, 8) // floating FU
    variables(9) = new Int(1, 8) // branch FU
    variables(10) = new Int(1, 8) // memory FU

    variables
  }
```

In acest caz, o soluție este compusa din mai multe variabile, variabilele corespunzand caracteristicilor procesorului cu constrangerile date de domeniul lor de valori. Valoarea acestor variabile va fi folosită mai apoi pentru a calcula viteza și poziția indivizilor (prin simulare).

Este totusi necesar si limbajul Java deoarece biblioteca jMetal este scrisa in Java si a fost necesara modificarea aceastei biblioteci pentru a satisface cerintele proiectului. Datorita faptului ca Scala este un dialect al Java, interactiunea cu cod Java din interiorul codului Scala nu a fost o problema. Motivul pentru care a fost necesara modificarea jMetal este faptul ca, in urma implementarii clasei CpuSolutionType, clasa XReal din pachetul jmetal.util.wrapper nu este capabila de a prelua informatia din noul tip de solutie. Clasa XReal stie sa citeasca doar dintr-un numar limitat de tipuri de solutii, acestea fiind hard-codate in interiorul clasei XReal. Acest lucru este probabil un defect al bibliotecii jMetal, dar nu reprezinta totusi o problema deoarece, avand sursele bibliotecii, defectul a fost remediat astfel:

```
public double getValue(int index) throws JMException {
            if ((type_.getClass() == CpuSolutionType.class) ||
                        (type_.getClass() == RealSolutionType.class) ||
                        (type_.getClass() == BinaryRealSolutionType.class)){
                  return solution_.getDecisionVariables()[index].getValue() ;

            }
```

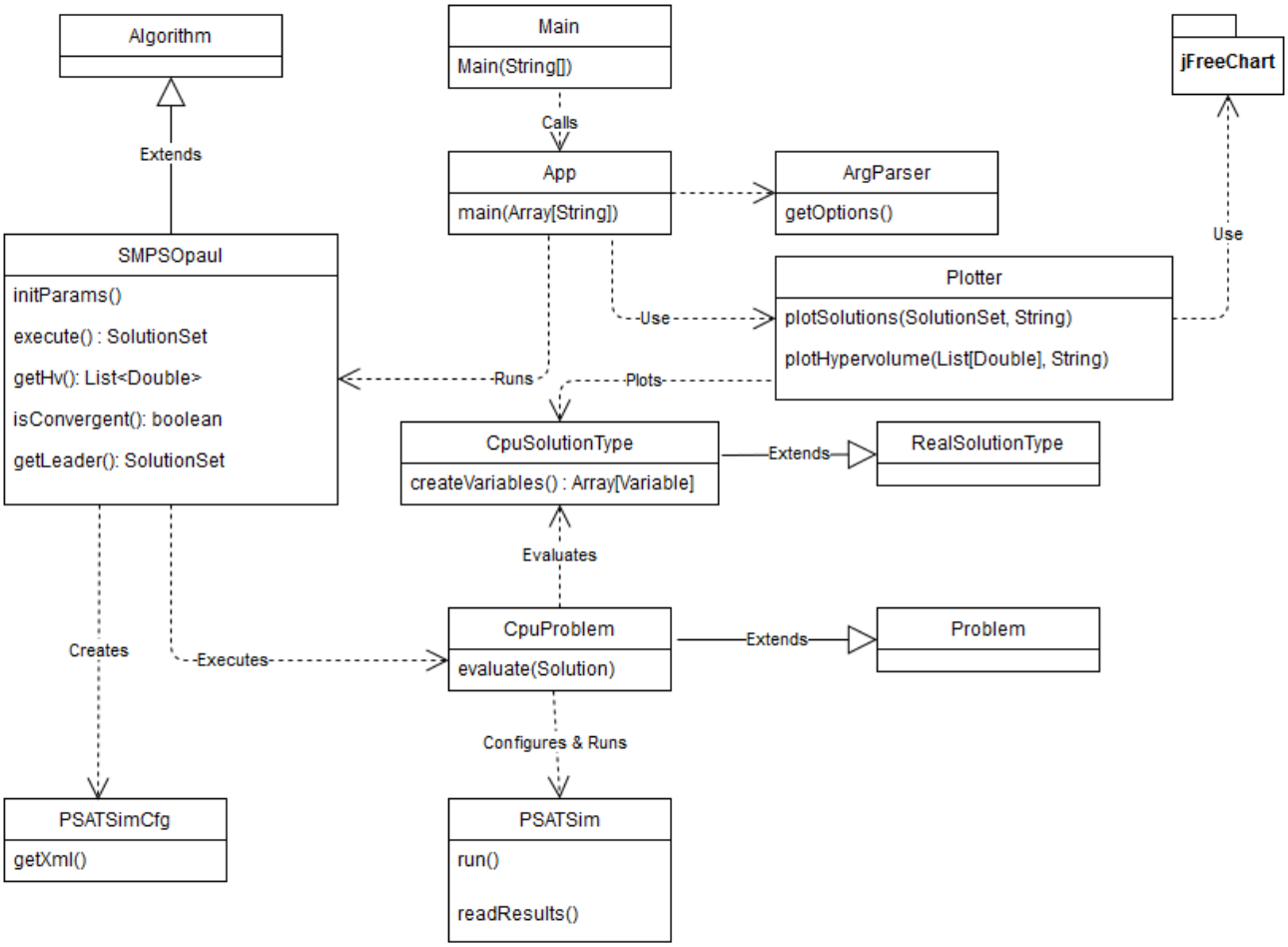
```
        else if //…
        //…
        return 0.0 ;
    }
```

De asemenea tot la categoria “modificari aduse bibliotecii jMetal” ar putea intra si implementarea clasei „SMPSOpaul”, care aduce în plus conditia de oprire atunci când hipervolumul unei populatii nu mai difera semnificativ fata de hipervolumul anterior. În acest caz „nesemnificativ” reprezintă schimbari ale ariei dincolo de a treia zecimala (**ε** = 0.001 sau parametrul din linia de comanda).

Pentru a desena graficele (plot-solutions.png și plot-hv.png) s-a folosit biblioteca jFreeChart, și aceasta fiind implementata tot în Java. Aceasta biblioteca este folosită în clasa Plotter. În aceasta clasa se creaza și se salvează graficele pe disc. Tot aici se normalizeaza datele pentru graficul hipervolumului, toate ariile fiind raportate la aria maxima.

Așadar rezultatele sunt generate și salvate automat în urma procesului de explorare prin intermediul clasei Plotter (pentru grafice) și a clasei SMPSOpaul (care este doar clasa SMPSO din jMetal cu adaugirile mentionate mai sus).
Paralelismul în aceasta aplicație este exploatat la nivelul simulatorului PSATSim. Acesta este capabil de a rula doua sau mai multe configuratii de procesor în paralel. Așadar, când se evalueaza un individ, simulatorul ruleaza trace-urile în paralel, reducandu-se astfel timpul de rulare și exploatandu-se la maxim resursele hardware ale sistemului de calcul.

Pentru a aduce împreuna conceptele prezentate sumar anterior se prezinta în cele ce urmează diagrama UML a proiectului (clasele goale apartin jMetal):

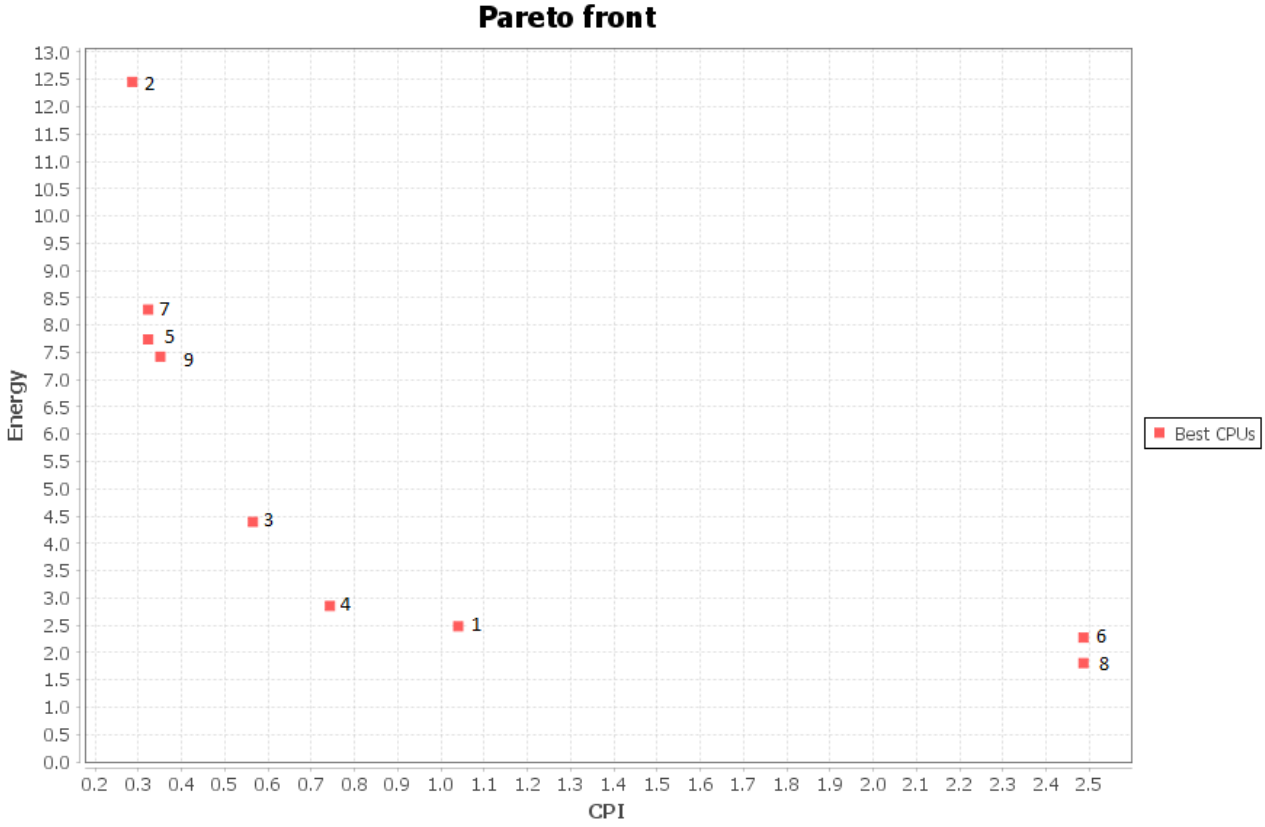


Se poate vedea in diagrama de mai sus ca exista câteva puncte cheie în arhitectura exploratorului automat care pot fi modificate sau inlocuite, în funcție de nevoi:

- Dacă se dorește schimbarea algoritmului, atunci putem inlocui clasa SMPSOpaul cu o alta clasa care implementeaza alt algoritm, este important ca aceasta noua clasa sa mosteneasca clasa Algorithm din jMetal.
- Pentru a modifica problema careia sa ii găsim optimul, trebuie inlocuite clasele CpuSolutionType și CpuProblem, care trebuie sa extinda obligatoriu o clasa din pachetul jmetal.encodings.solutionType și respectiv clasa Problem (din pachetul jmetal.core). Eventual modificarea problemei poate atrage după sine și un nou simulator, caz în care trebuie inlocuite clasele PSATSim și PSATSimCfg.
- În cazul în care se doresc alte tipuri de grafice, trebuie modificata clasa Plotter.

Concluzii și interpretari

În urma unor versiuni initiale ale programului (înainte de implementarea conditiei pe hipervolum), după 18 minute de rulare, 11 generații și 11 interatii ale algoritmului PSO s-a obținut acest rezultat:



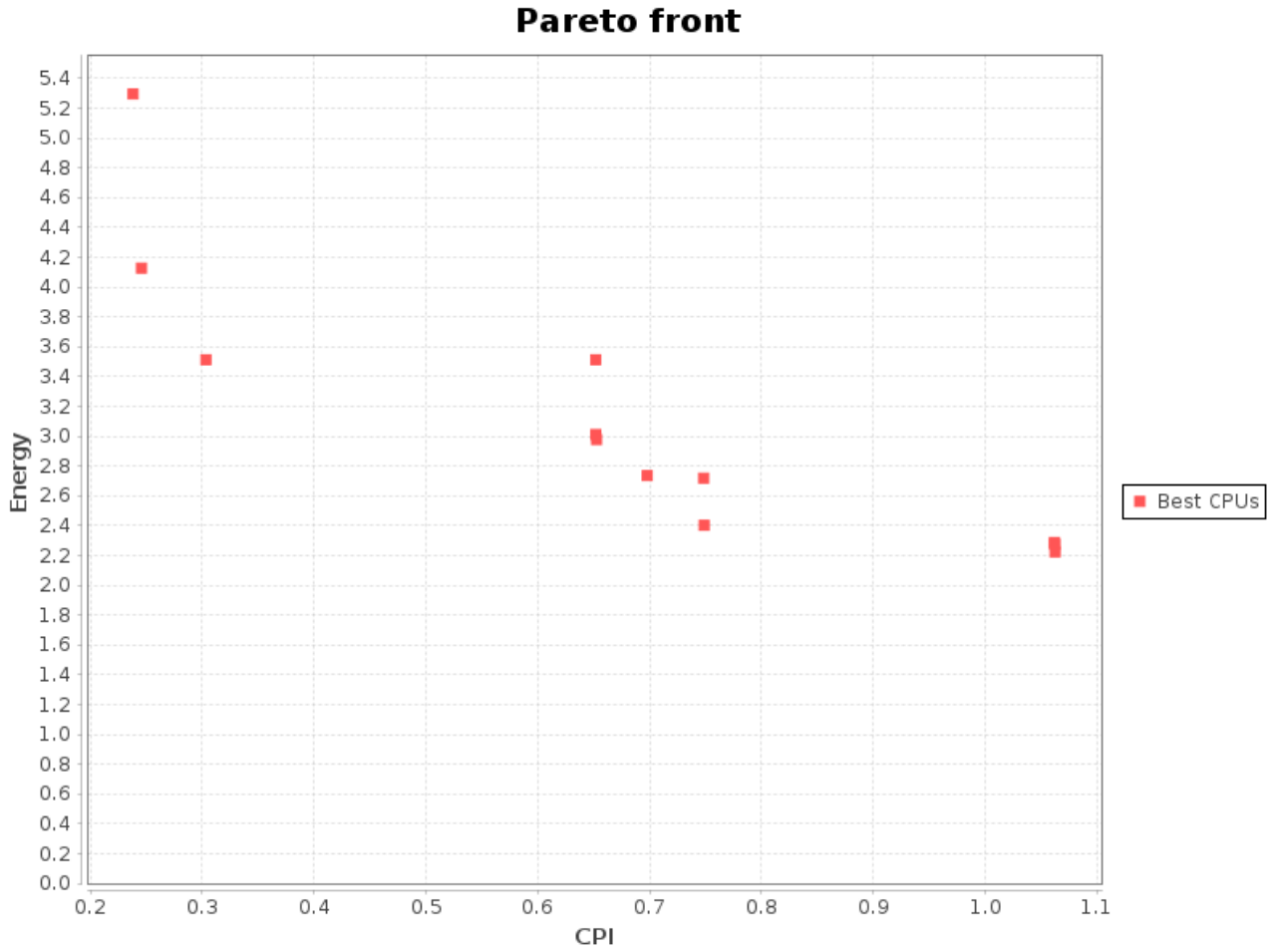
Verificand fișierul VAR, deducem ca procesoarele noastre, găsite în urma explorării automate folosind particle Swarm Optimization au următoarele configuratii:

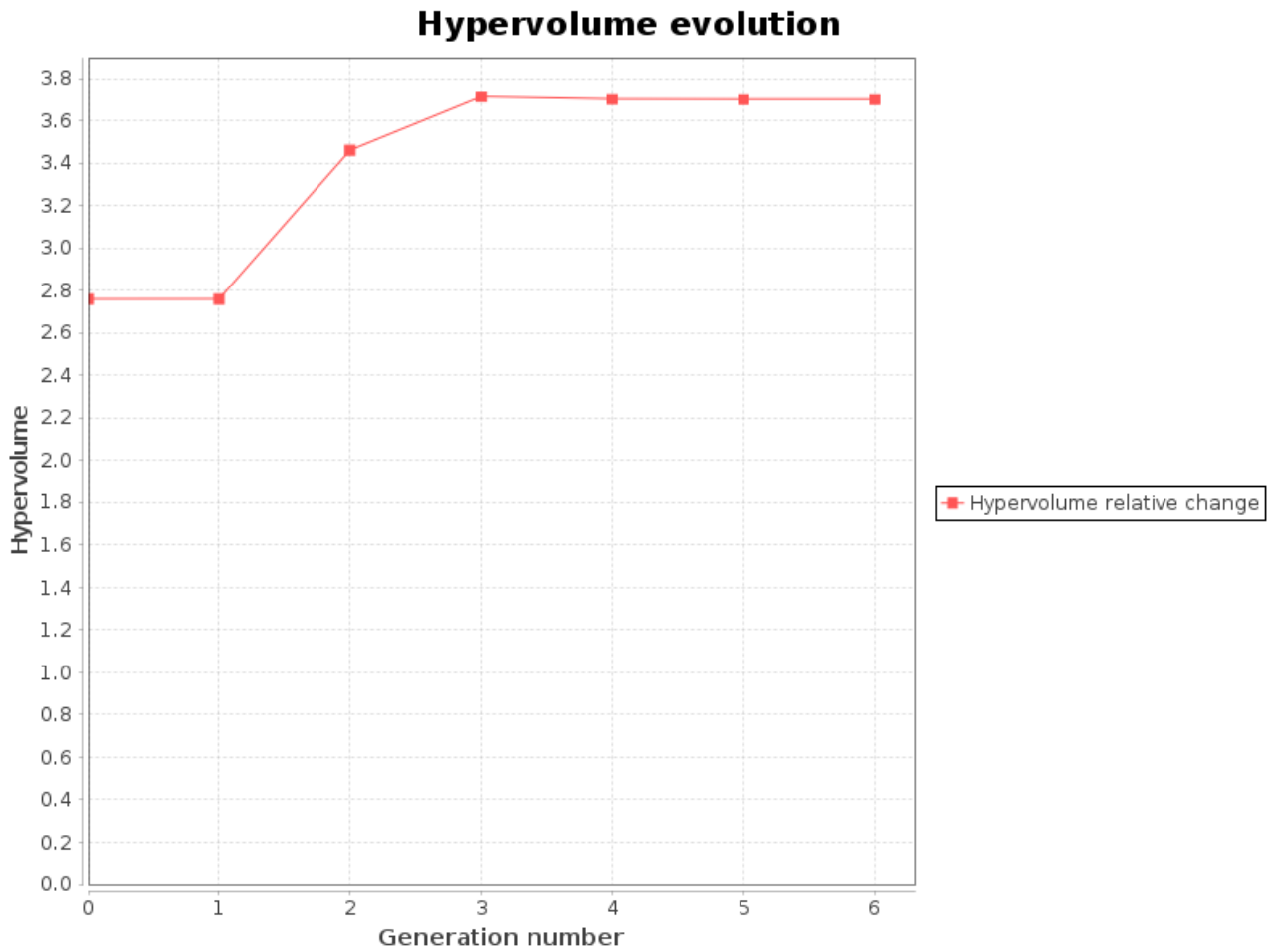
#	Superscalar	Rename	Reorder	RSB	Separate dispatch	VDD	Freq	Integer FU	Floating FU	Branch FU	Memory FU
1	14	473	3	2	1	1.929	1459	3	6	4	3
2	16	85	448	2	1	2.582	1403	4	5	6	5
3	13	27	17	2	1	3.116	1469	7	5	6	1
4	13	282	6	2	1	2.366	1461	3	5	4	2
5	15	164	261	2	1	2.448	1418	3	5	5	4
6	6	512	1	2	1	1.800	1570	1	6	1	1
7	15	152	286	2	1	2.456	1413	3	5	5	4
8	1	512	1	2	1	1.800	1646	1	6	1	1
9	14	175	266	2	1	2.416	1416	3	5	4	3

Urmărind liniile din fișierul FUN vedem urmatoarele valori pentru cele doua obiective optimizate:

#	CPI	Energy
1	1.03971206	2.480243
2	0.2860141291	12.45488
3	0.5652037391	4.394594
4	0.7433339221	2.853436
5	0.3228312836	7.73752
6	2.4861783131	2.273825
7	0.3227228404	8.283152
8	2.4861891932	1.808303
9	0.3507120139	7.421373

Mai jos putem vedea frontul Pareto, cât și evolutia hipervolumului în ultima versiune a programului:





Observam ca aria hipervolumului se mărește pe cât avansam în numărul de iteratii ale algoritmului. Indivizii apropiindu-se tot mai mult de optimul dorit.
Acest exemplu a fost rulat în aproximativ 15 minute, cu o populatie de 15 indivizi și **ε** = 1.0E-7.

Ambele exemple au fost rulate pe un procesor Intel Core i3 cu 4 GB de RAM.
Se poate observa ușor faptul ca adaugarea conditiei pe hipervolum ne garantează faptul ca ne vom opri atunci când nu mai putem găsi indivizi care sa aducă imbunatatiri semnificative ariei hipervolumului, deci putem spune cu confidenta ca nu se mai aduc nici imbunatatiri semnificative asupra configuratiilor procesoarelor. Acest lucru nu era valabil la oprirea conditionata doar de numărul de iteratii, acolo riscand sa irosim resurse chiar și după ce indivizii nu se mai schimbau semnificativ.

Alte documente:

- Documentatia jMetal se poate gasi la adresa: http://jmetal.sourceforge.net/javadoc/index.html
- Ghid jMetal: https://sourceforge.net/projects/jmetal/files/jmetal4.5/jmetal4.5.userManual.pdf/download
- PSATSim: http://homepages.udayton.edu/~ttaha1/psatsim/
- Documentatie PSATSim: /calea/catre/directorul/PSATSim/cli_usage.html
- Tutorial PSO: http://www.swarmintelligence.org/tutorials.php
- PSO interactiv: https://ccl.northwestern.edu/netlogo/models/ParticleSwarmOptimization
- Descriere PSO: https://www.researchgate.net/publication/297245624_Particle_Swarm_Optimization_Algorithm_and_its_Codes_in_MATLAB