COSTUL DE PERFORMANȚĂ AL PROGRAMĂRII ORIENTATE PE ASPECTE

ing. Şerban Drăgănescu serban.draganescu@gmail.com

prof. dr.ing. Nicolae Ţăpuş ntapus@cs.pub.ro

Facultatea de Automatică şi Calculatoare, Universitatea *Politehnica* din Bucureşti

Rezumat: Costul de performanță este definit ca timpul suplimentar consumat de o aplicație POA[1] comparativ cu aceeaşi aplicație care implementează manual preocupările încrucişate.

Sunt comune considerațiile teoretice despre impactul POA asupra performanței, dar această lucrare încearcă să reveleze nişte valori reale ale încărcării suplimentare, într-o manieră grafică, folosind o aplicație reală (o bază de date) şi câteva operații pe baza de date cu întreţesere implementată atât în faza de compilare, cât şi în faza de rulare într-un mediu Java.

Cuvinte cheie: aspect, AspectJ, AOP, performanță, cantitativ, cost, Java, faza rulare, faza compilare

Abstract: The overhead is defined as the additional time spent by the AOP application compared to the same application that implements the cross-cutting code manually.

There are, usually, theoretical considerations about AOP impact on performance but this paper tries to reveal real numbers on the total overhead in a graphical manner, using a real application (a database) and several database operations, with both compile-time and run-time Java AOP weaving implementations.

Keywords: aspect, AspectJ, AOP, performance, quantitative, overhead, Java, runtime, compile time

1. Introducere

Motivație

Căutând informații despre impactul POA asupra performanței nu am găsit publicații cu date cantitative despre POA. Există doar considerații teoretice, care sunt intuitive, şi doar date brute despre costul individual al diverselor operații. Este adevărat că ne-am aştepta să nu vedem niciun impact asupra performanței atunci când se foloseşte întreţesere la compilare şi să se observe un oarecare impact la întreţeserea la faza de rulare. Această lucrare încearcă să confirme această teorie şi să ofere nişte valori numerice.

Despre h2, baza de date

Baza de date h2 [h2] este o bază de date cu surse publice (open source) scrisă în Java şi publicată sub Licența Publică Mozilla (Mozilla Public License). Denumirea vine de la „Hipersonic 2”, Hipersonic fiind o bază de date anterioară, de acelaşi autor.

Baza de date poate rula integral în memorie sau pe disc şi se concentrează pe o amprentă mică de memorie.

Am ales această bază de date pentru că se potriveşte cu criteriile de a fi o aplicație mare, scrisă în Java, cu surse disponibile. H2 a rulat fără probleme câteva scripturi Oracle, fără modificări, iar sursele sunt disponibile pentru modificare şi recompilare.

---

1 Programarea Orientată pe Aspecte

Despre AspectJ

Conceptul POA a fost realizat prin AspectJ [AJ] la laboratoarele centrului de dezvoltare Xerox Parc, devenit apoi IBM. După momentul integrării cu AspectWerkz [AW], o altă platformă POA, AspectJ a devenit standardul de fapt pentru POA.

Întreţeserea în faza de compilare şi întreţeserea în faza de rulare

Sunt mai multe feluri de a face un advice să fie rulat. Advice-ul este partea suplimentară de cod care trebuie inserat în locurile definite de aspect. Procesul inserării codului e denumit întreţesere – weaving.

O cale de a întreţese advice-urile este de a utiliza un compilator special care, pe lângă surse, are în vedere şi aspectele definite, inserând advice-urile în poziţiile corespunzătoare. Rezultatul rulării unui astfel de compilator este bytecode normal Java (fişiere .class) care sunt rulate cu un executabil Java. Un astfel de compilator care face întreţesere la faza de compilare este *ajc*.

O altă cale (şi alt moment) de a întreţese advice-urile în cod este momentul în care clasa este încărcată de maşina virtuală, numită întreţesere la faza de rulare. Maşina virtuală delegă procesul de încărcare unui încărcător de clase şi de asemenea oferă unul implicit. Întreţeserea la faza de rulare funcţionează prin schimbarea încărcătorului de clase cu un altul care interceptează codul binar (bytecode) original (în mod normal încărcat dintr-un fişier .class). În versiunile de Java 1.5 şi mai recente mai există o modalitate, mai puţin cunoscută, numită „agent Java” şi pachetul `java.lang.instrument`. Aceasta permite o modalitate simplă de instrumentare a claselor Java: nu mai e necesară implementarea unui nou încărcător de clase, ci orice încărcător de clase folosit consultă agentul (dacă există) prin simpla implementare a unei metode. Agentul Java trebuie referit în linia de comandă, la apelul executabilului Java.

2. Alegerea încărcării sistemului

Această lucrare îşi propune măsurarea impactului asupra performanţei POA în cazul unei aplicaţii reale într-o situaţie tipică. În acest scop capitolul curent prezintă uneltele, programele şi configurările folosite. În subcapitolul următor descriem programul ales şi justificăm alegerea făcută, iar în al doilea variantele care sunt evaluate.

2.1 Ce se rulează în timpul testului

Testele inventate tind să fie subiective, în ciuda bunelor intenţii. Ca să evităm situaţia aceasta, folosim o bază de date scrisă în Java, *h2* iar datele de test şi scripturile SQL folosite sunt împrumutate dintr-o aplicaţie din viaţa reală care rulează pe *Oracle* şi faptul că *h2* şi *Oracle* au aceeaşi înţelegere asupra SQL a fost de ajutor. De asemenea faptul că toate scripturile SQL au fost scrise respectând înţelesul cuvântului *standard*, evitând capcanele extensiilor *Oracle* a dus la încărcarea fără probleme a schemei în *h2*.

Datele de test au câteva sute de linii de creare a schemei (tabele, indexuri, secvenţe, chei ş.a.) şi câteva zeci de mii de inserări, la origine rulate în *Oracle*.

Pe această aplicaţie am adăugat cod ca să facem nişte acţiuni tipice de POA: jurnalizarea tuturor operaţiilor de inserare SQL într-un fişier, împreună cu durata fiecărei execuţii.

De asemenea metoda „main” din aplicaţie îşi jurnalizează propria durată, pentru măsurarea performanţei. Această metodă este preferată măsurării duratei execuţiei întregii Java, care ar fi inclus şi încărcarea jar-urilor AspectJ şi, în cazul întreţeserii la rulare, punerea

în scenă a agentului Java. Am considerat că rezultatele fără această diferență de pornire sunt mai interesante.

2.2 Sunt două implementări: manual şi POA

Codul adăugat pentru jurnalizarea şi cronometrarea comenzilor SQL *insert* trebuie să fie prezent în două versiuni: una în care este făcut prin scrierea manuală a codului în interiorul claselor *h2* şi a doua în care codul este scris ca un *advice* POA şi întreţesut cu AspectJ. De asemenea a doua abordare se prezintă în două arome: una în care întreţeserea se face la faza de compilare şi a doua, în care întreţeserea se face la faza de încărcare, prin agentul Java.

În implementarea manuală, pe care o vom numi întreţesere manuală, metoda arată ca mai jos:

```
long start = System.currentTimeMillis();
```

[method initial body except for the return statement]

```
long stop = System.currentTimeMillis();
FileLogger.log("update() în clasa: Insert.java"
      + " sql=" + getSQL() + " a durat " + (stop-start));
return count;
```

În implementările POA am definit un *advice*:

```
Object around(): insertCall() {
      Object[] args = thisJoinPoint.getArgs();
      Object thisObject = thisJoinPoint.getThis();
      long start = System.currentTimeMillis();
      Object result = proceed();
      long stop = System.currentTimeMillis();
      FileLogger.log("update() in class: Insert.java, sql=" +
            ((Insert)thisObject).getSQL() + " took " + (stop-
start));
      return result;
}
```

Acest *advice* este întreţesut de compilatorul special *ajc* la faza de compilare sau de agentul AspectJ la faza de încărcare a clasei. Ca să fie întreţesut la încărcarea clasei, maşina virtuală Java are nevoie de un agent, folosind următorul parametru în linia de comandă: `-javaagent:$ASPECTJ_HOME/lib/aspectjweaver.jar"`.

3. Testele

Am selectat două tabele ale bazei de date şi constrângerile lor, tabele care conţin 210406 înregistrări, respectiv 16729.

Metoda „main” execută următoarele operaţii: crearea schemei (două tabele şi constrângerile pentru că am ales să folosim baza de date în memorie) şi încărcarea tabelelor folosind operaţiunile SQL *insert*. La sfârşitul metodei este calculată durata şi memorată într-un fişier.

În al doilea test, pe care îl vom numi *placebo*, după încărcarea tabelelor (şi rularea codului afectat de aspecte) se mai rulează două metode care efectuează operaţii SQL *select* complexe pe cele două tabele şi le măsoară durata. Aceste operaţii select nu mai sunt afectate de aspecte şi urmărim dacă performanţa este afectată de faptul că în sistem sunt şi aspecte.

3.1 Încercări, măsurări, metode

Măsurarea propriu-zisă are loc în metoda „main” prin citirea timpului la început şi la sfârşit. După fiecare rulare durata este adăugată la sfârşitul unui fişier. Acelaşi test este rulat de multe ori pentru a minimiza şansele de eroare.

Câteva cauze de erori au fost identificate şi îndepărtate în timpul testelor. Cea mai probabilă cauză este că procesorul ar putea, accidental, să se ocupe de alte procese care sunt variabile în durată. Calculatorul nu execută alte operaţiuni notabile în paralel, dar întotdeauna există alte procese şi operaţiuni ale sistemului de operare în fundal. De asemenea cache-urile (din sistemul de operare şi procesor) sunt diferite cu fiecare rulare. În scopul minimizării acestui efect, s-a efectuat un număr mare de rulări urmărindu-se pe histogramă să existe distribuţii similare în toate cazurile.

O altă posibilă cauză de eroare ar fi viteza variabilă a procesorului, IntelSpeedStep în cazul nostru[2]. Anumite tipare de încărcare a procesorului pot fi favorizate şi altele handicapate de algoritmul de scădere a vitezei procesorului în funcţie de încărcare aşa că viteza procesorului a fost blocată.

În mod inerent au fost folosite două compilatoare, fiind implicite fiecare în situaţii diferite. Pentru întreţeserea manuală, firesc, *javac* (executabilul compilatorului de Java) de la Sun a fost folosit, în timp ce pentru întreţeserea la faza de compilare a fost folosit *ajc* (executabilul Aspect Java Compiler) din cadrul proiectului Eclipse al IBM. Acestea două sunt situaţiile clare, dar în celelalte cazuri, se pot imagina căi alternative. De exemplu, în cazul întreţeserii în faza de încărcare aspectele sunt compilate cu *ajc,* în timp ce restul claselor sunt compilate cu *javac*. Alternativ clasele pot fi compilate toate cu *ajc*. Din cauza acestei situaţii, pentru a minimiza efectul compilatoarelor asupra rezultatelor, unde a fost posibil au fost folosite variante cu ambele compilatoare (*javac* şi *ajc*).

O mai puţin evidentă situaţie apare din prezenţa procesoarelor multiple. Testele au arătat că folosirea a două procesoare (sau a unui procesor dual) afectează puternic distribuţia rezultatelor şi favorizează una din implementări. Vom arăta rezultate pe o maşină cu un procesor şi câteva rezultate pentru o maşină cu două procesoare, pentru comparaţie. Pentru realizarea testelor pe un singur procesor, al doilea procesor nu este folosit prin opţiunea de kernel `maxcpus=1`. Această opţiune este potrivită pentru că păstrează acelaşi kernel, schimbând doar numărul de procesoare folosite.

3.2 Rezultate

După cum am menţionat mai sus, sunt mai multe feluri de a întreţese advice-urile şi noi testăm două din ele: întreţesere la faza de compilare şi întreţesere la faza de încărcare a claselor. Pentru a avea o bază de comparaţie, am implementat şi o întreţesere „manuală” în care codul Java conţine deja comportamentul definit ca aspecte, deci nu se mai execută nicio întreţesere asupra lui (pentru simplitate vom folosi sintagma de *întreţesere manuală*, fără ghilimele). Din perspectiva întreţeserii avem:

- Întreţesere manuală, etichetată ÎM
- Întreţesere în faza de compilare, etichetată ÎC
- Întreţesere în faza de încărcare, etichetată ÎÎ

Având în vedere că toate de mai sus, exceptând ÎC, pot fi realizate folosind şi *ajc* şi *javac*,

2 Tehnologie Intel care permite programelor (tipic sistemului de operare) să scadă viteza procesorului şi chiar mărimea cache-ului în momentele cu încărcare mică. Similar există tehnologia AMD Cool'n'Quiet.

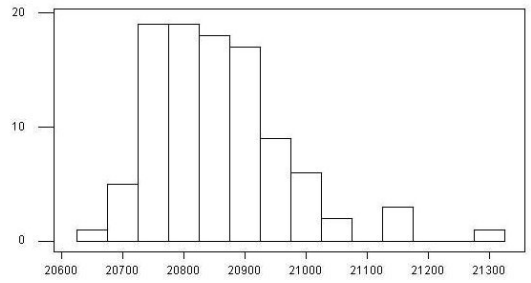
situațiile de testare devin:

- Întrețesere manuală şi compilare cu *javac*, etichetată ÎMJ
- Întrețesere manuală şi compilare cu *ajc*, etichetată ÎMA
- Întrețesere în faza de compilare şi compilare cu *ajc*, etichetată ÎCA
- Întrețesere în faza de încărcare şi compilare cu *ajc*, etichetată ÎÎA
- Întrețesere în faza de încărcare şi compilare cu *javac*, etichetată ÎÎJ

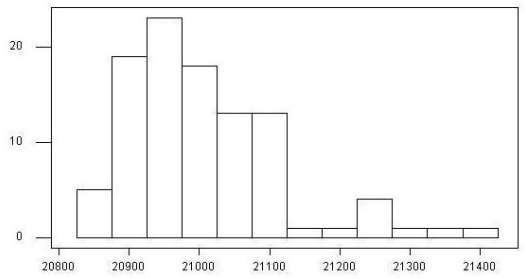
Primele valori pe care le prezentăm sunt obținute cu 100 de eşantioane pe test (fiecare test a fost rulat de 100 de ori), cu viteza procesorului blocată (fără SpeedStep), un singur procesor (opțiunea de kernel maxcpus = 1).

Distribuția eşantioanelor e importantă pentru că arată imediat anomaliile de date, în principal cele apărute din cauza încărcării nedorite a procesorului în timpul testelor. Ca să vedem că numărul de rulări este potrivit de mare şi că nu au apărut evenimente de procesor semnificative în timpul rulărilor am verificat că rezultatele tuturor testelor prezintă histograme similare.

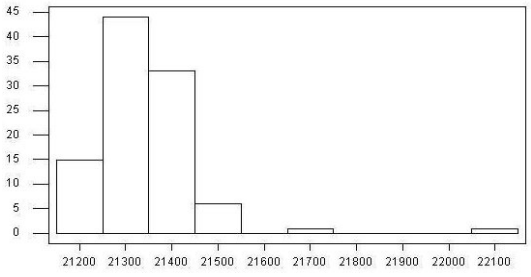
După cum ne şi aşteptam, tendința este ca mai multe eşantioane să se apropie de valoarea minimă:



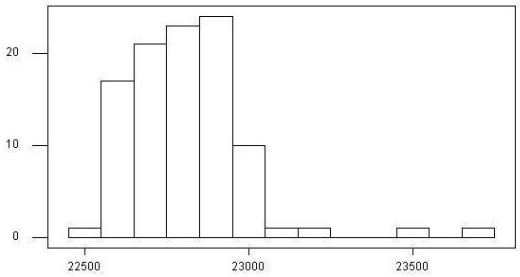
Ilustrația 1: Distribuția pentru procesor unic, ÎMJ



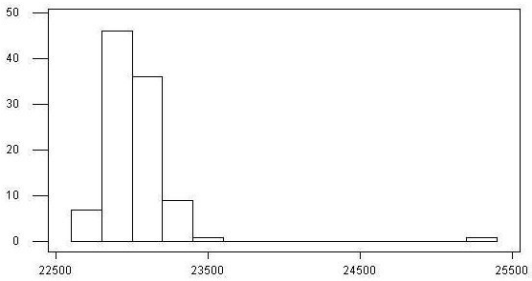
Ilustrația 2: Distribuția pentru procesor unic, ÎMA



Ilustrația 3: Distribuția pentru procesor unic, ÎCA



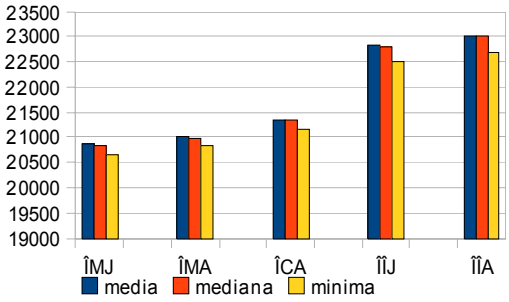
Ilustrația 4: Distribuția pentru procesor unic, ÎÎJ



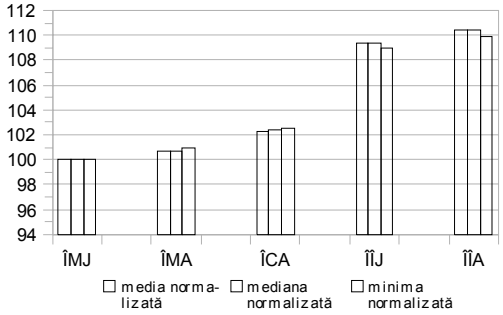
Ilustrația 5: Distribuția pentru procesor unic, ÎÎA

Considerăm câteva valori calculate statistic ca fiind relevante. Calculăm media, mediana şi minima pentru eşantioane. Din cauza caracteristicilor testelor, fiind făcute din componenta de calcul pur plus timpul în care procesorul nu e disponibil nouă, considerăm că *minima* este foarte semnificativă, mai semnificativă decât *media,* iar *maxima* nu are nicio valoare.

Media ÎMA (întreţesere manuală şi compilare cu ajc) calculată din 100 de rulări a luat 21 de secunde (Ilustraţia 6). Considerând ÎMJ ca normal, costurile relative sunt în Ilustraţia 7.



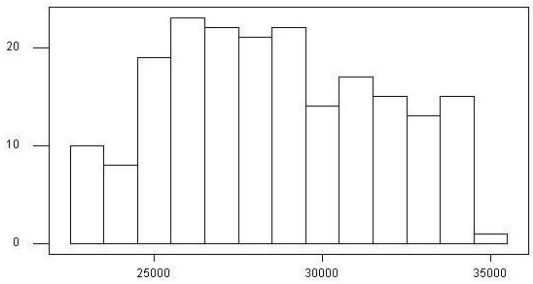
Ilustraţia 6: Timpii de rulare pentru procesor unic



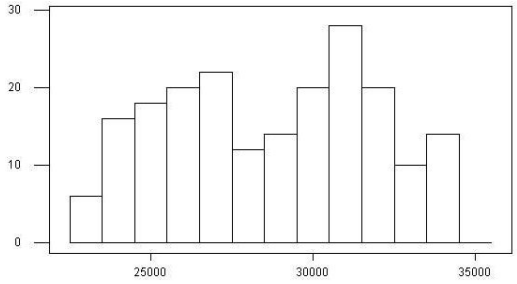
Ilustraţia 7: Costurile relative la ÎMJ

Acelaşi test a fost rulat pe două procesoare (un dual core), cu viteza procesorului tot blocată (SpeedStep dezactivat). Rezultatele care urmează sunt pentru 200 de rulări per test:

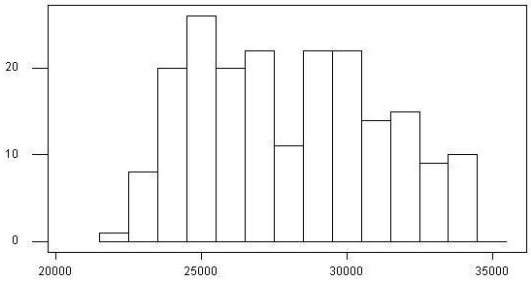
Distribuţia eşantioanelor:



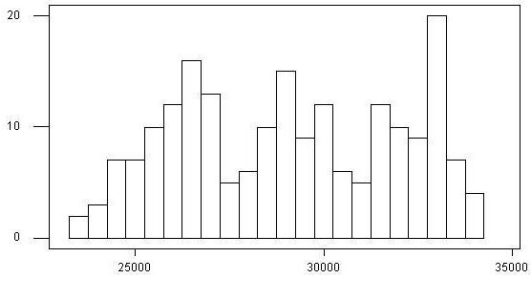
Ilustraţia 8: Distribuţia ÎMJ, două procesoare



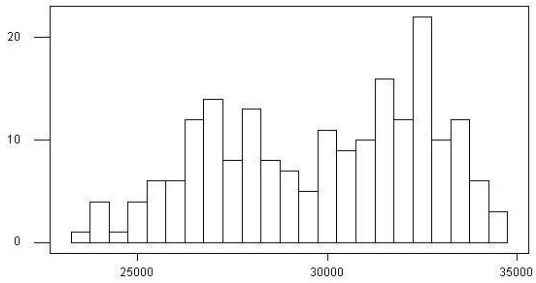
Ilustraţia 9: Distribuţia ÎMA, două procesoare



Ilustraţia 10: Distribuţia ÎCA, două procesoare

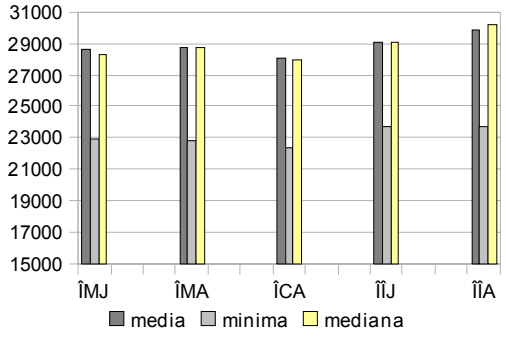


Ilustraţia 11: Distribuţia ÎÎJ, două procesoare

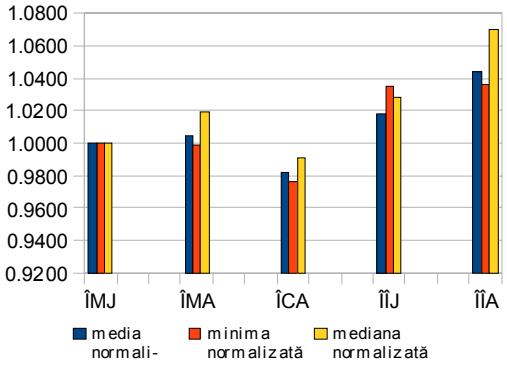


Ilustrația 12: Distribuția ÎÎA, două procesoare

Media, mediana şi minima arată de asemenea rezultate interesante (Ilustrația 13). Considerând ÎMJ ca referință: Ilustrația 14.



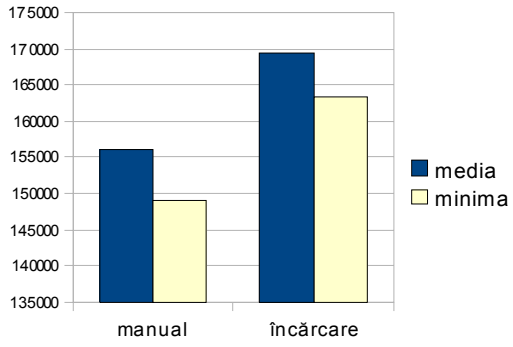
Ilustrația 13: Timpii de rulare, două procesoare



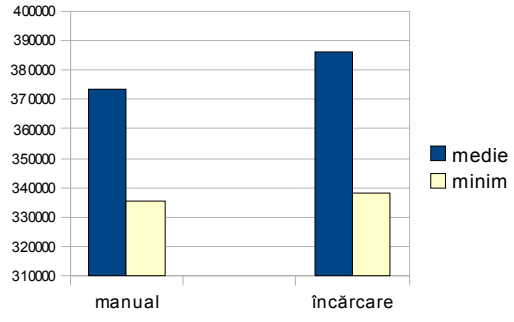
Ilustrația 14: Timpii de rulare raportați la ÎMJ, două procesoare

Următorul test este numit *placebo* pentru că evaluează efectul aspectelor atunci când nu sunt rulate aspecte. Doar două cazuri au fost considerate relevante: cazul ÎMJ, în care întreţeserea nu e folosită deloc, care constituie baza de comparație şi cazul ÎÎJ în care se face întreţesere la încărcare, dar nu pentru codul evaluat.

Am verificat că rezultatele împărtăşesc aceeaşi distribuție în ambele cazuri (aceeaşi distribuție când se rulează pe un singur procesor şi aceeaşi când se rulează pe două procesoare, nu mai sunt ilustrate) şi comparăm rezultatele:



Ilustrația 15: Timpii de rulare placebo, un procesor



Ilustrația 16: Timpii de rulare placebo, două procesoare

4. Discuție şi concluzii

Rezultatele rulării pe un singur procesor, din prima grupă, arată aşa cum ne-am aştepta:

- **întreţeserea manuală** este cea mai rapidă,
- urmată îndeaproape de **întreţeserea la compilare** cu un timp în jur de 2% mai mare: în medie 2,32%, faţă de minimă cu 2,52% comparat cu rezultatele compilatorului *javac,* 1,6% în medie şi 1,52% la minime folosind acelaşi compilator (*ajc*),
- iar cea mai *scumpă* este **întreţeserea la încărcare**, cu un cost suplimentar în jur de 10%, 9,38% în medie şi 9,02 la minime pentru comparaţie *javac* la *javac* şi 9,62 în medie şi 8,87 la minime pentru comparaţie *ajc* la *ajc*.

Numerele de mai sus sunt reflectate în Ilustraţia 7 şi în tabela următoare:

	MWJ	MWA	CWA	LWJ	LWA
media normalizată	100	100.71	102.32	109.38	110.4
mediana normalizată	100	100.69	102.39	109.4	110.35
minima normalizată	100	100.99	102.52	109.02	109.95

Întreţeserea manuală şi la compilare au performanţe cvasi egale. Diferenţa de 1,6% în medie, folosind acelaşi compilator -ajc, nu e insignifiantă dar foarte mică. Acest cost suplimentar ar putea fi teoretic aproape de 0 dar probabil că sunt sofisme sintactice care se cumulează şi crează costul real. Iată codul generat de ajc pentru un advice de tip around. În primul rând corpul metodei devine[3]:

```
JoinPoint joinpoint = Factory.makeJP(ajc$tjp_0, this, this);

return Conversions.intValue(update_aroundBody1$advice(this, joinpoint,
InsertLogger.aspectOf(), null, joinpoint));
```

şi apoi mai sunt două metode care se apelează una pe alta: `update_aroundBody1$advice` şi `update_aroundBody0`. Considerăm că aceste apeluri sunt responsabile de mica diferenţă între ÎMA şi ÎCA şi un comportament adiţional implementat. Totuşi diferenţa este minoră şi de mică importanţă practică.

O distanţă importantă apare între întreţeserea manuală şi cea la încărcare, în jur de 10%. Întreţeserea la încărcare se face în maniera următoare: încărcătorul de clase (classloader) permite unui plugin, denumit „agent Java" să intercepteze codul binar (bytecode) în timpul încărcării de pe disc[4] în memorie. Agentul apoi parsează codul binar şi îl modifică conform cu aspectele definite, în cazul nostru, în aop.xml. Ar putea fi considerat că aceasta modificare realizată o singură dată pe ciclul de viaţă al unei clase, la încărcare, nu ar trebui să afecteze semnificativ performanţa generală a aplicaţiei. Totuşi această întreţesere are nevoie de un întreg sistem în spate care, pe lângă bibliotecile de cod ale logicii, trebuie să creeze şi menţină structura de date care ţine contul aspectelor şi claselor deja încărcate. Această structură poate creşte considerabil şi astfel contribuie la mărirea distanţei dintre întreţeserea manuală şi cea la încărcare.

Diferenţa de performanţă între ÎMJ şi ÎMA (aceeaşi modalitate de întreţesere, dar compilatoare diferite) a fost generată de folosirea parametrilor impliciţi la compilare, care s-au dovedit a fi diferiţi. În cazurile ÎMJ şi ÎMA clasele generate au avut dimensiuni sensibil diferite, de exemplu: 5354 pentru ÎMJ şi 7308 pentru ÎMA. La o privire sumară, diferenţele par a fi

3 Clasa generată a fost decompilată folosind *jad*: http://en.wikipedia.org/wiki/JAD_(JAva_Decompiler)

4 Discul e cazul obişnuit, dar încărcarea se poate face din diverse surse

generate de lucruri ca:

```
int count;//la ÎMA
```

devine

```
int i;//la ÎMJ
```

Rezultate foarte surprinzătoare au fost obținute la rulare pe procesoare duale. Trebuie să subliniem că testele noastre sunt operațiuni secvențiale şi puține operații pot fi implementate în paralel de baza de date, ne aşteptăm niciuna. În aceste teste se pare că cea mai rapidă implementare este ÎCA. Cum am mai zis, ÎCA diferă, de ÎMA de exemplu, prin faptul că mai există ceva operații de executat, deci aceste operații suplimentare s-ar executa într-un timp negativ. Pentru că aceasta nu se poate, însemnă că anumite operații se execută în paralel în cazul întreţeserii la compilare, ÎCA, pe procesorul disponibil în plus. Acest al doilea procesor este disponibil în sensul că nu e folosit explicit de aplicație, dar el execută ocazional operații ale sistemului de operare.

Iată valorile raportate la ÎMJ:

	ÎMJ	ÎMA	ÎCA	ÎÎJ	ÎÎA
media normalizată	1.0000	1.0043	0.9818	1.0178	1.0443
minima normalizată	1.0000	0.9988	0.9762	1.0349	1.0357
mediana normalizată	1.0000	1.0188	0.9908	1.0281	1.0698

ÎCA este mai rapidă datorită procesării paralele permise de codul inserat de întreţeserea la faza de compilare.

De asemenea întreţeserea la încărcare are un câştig uriaş, acum fiind cu doar 1,78% mai lentă (ÎÎJ comparat cu ÎMJ) datorită aceluiaşi motiv. De fapt costul ÎÎA şi ÎÎJ este acum (în utilizarea a două procesoare) la fel de mic ca şi costul de la ÎCA la ÎMA din mediu cu un singur procesor.

Nelegat de scopul acestei lucrări, am mai constatat un rezultat remarcabil: diferența de viteză în execuție între un mediu cu un singur procesor şi un mediu cu două procesoare în cazul operațiunilor secvențiale. ÎMJ este în medie 28584.07s pentru două procesoare şi 20855s pentru un singur procesor, o diferență de 37%. Credem că diferența este în cea mai mare parte legată de sistemul de operare (Linux 2.6.27) şi într-o anumită măsură de maşina virtuală Java (Sun 1.6.0_15). În cazul în care am observat un beneficiu din partea procesării paralele, ÎCA, diferența este de doar 31,5%. Aceste diferențe sunt datorate nevoii sincronizării celor două procesoare, care se face în sistemul de operare. Aceste diferențe au fost obținute păstrând acelaşi kernel şi limitând numărul de procesoare (`maxcpus=1`). Alternativ, kernelul pentru un singur procesor (`nosmp`) obține diferențe mai radicale.

Al doilea caz de test, numit de noi placebo, studiază dacă costul întreţeserii la rulare este, împrumutând termeni din economie, un cost fix sau variabil în sensul că este un cost per advice întreţesut (variabil) sau un cost per joinpoint posibil (fix). Costul în cazul procesorului unic este 7% în medie şi 8% la minime iar în cazul procesorului dual costul este de 3% pe medie şi 1% pe minimă.

La testele monoprocesor s-a folosit opțiunea de kernel `nosmp` (în loc de `maxcpus=1`), variantă mai eficientă în care nu se folosesc deloc mecanismele de sincronizare inter-procesor şi care justifică diferențele între cele două grafice. Comparația între cele două grafice (un procesor comparat cu două procesoare) nu e discutată aici.

5. Concluzii finale

Costul introdus de POA este revelat de testele rulate pe un singur procesor. Diferența este foarte mică pentru întreţeserea la compilare (ÎCA), în jur de 1-2% şi semnificativă pentru întreţeserea la încărcare (ÎÎJ şi ÎÎA), în jur de 10%.

O situație interesantă apare când în sistem mai există un procesor nefolosit, caz frecvent în calculatoarele moderne unde procesoarele duale sunt standardul, iar cele triple şi cvadruple sunt adesea găsite. În aceste situații, cazul întreţeserii la compilare (ÎCA) este, surprinzător, mai rapid decât întreţeserea manuală (ÎMJ şi ÎMA). De asemenea întreţeserea la încărcare (ÎÎA şi ÎÎJ) este foarte favorizată de prezența unui procesor nefolosit, performanța lor fiind cu doar în jur de 2-4% mai proastă decât cea a întreţeserii manuale. Totuşi ÎÎA este în continuare cu 8% mai lent decât ÎCA, care este cel mai rapid în cazul prezenței celui de-al doilea procesor.

Prin efectuarea testului *placebo* se constată că, la utilizarea întreţeserii la încărcare, costul POA este general asupra aplicației, şi nu doar asupra *advice*-urilor inserate. Nu putem compara procentele (între testul obişnuit şi placebo) datorită profilului diferit al încărcării procesorului: în cazul *placebo* testul a avut calcule intensive, iar în primul caz operații cu memoria, dar putem spune că cele două costuri sunt similare: folosirea întreţeserii la rulare implică costuri de performanță similare şi componentelor care nu o folosesc efectiv.

BIBLIOGRAFIE

[h2] H2 database, http://www.h2database.com

[AJ] AspectJ, http://eclipse.org/aspectj

[AW] AspectWerkz, http://aspectwerkz.codehaus.org/

[AWB] AOP call overhead, http://docs.codehaus.org/display/AW/AOP+Benchmark

[METR] AOP Metrics http://aopmetrics.tigris.org/

[OBH08] Ortiz, Guadalupe; Bordbar, Behzad; Hernández, Juan - Evaluating the Use of AOP and MDA in Web Service Development, 2008

[BO09] De Borger, Wouter; Lagaisse, Bert; Joosen, Wouter - A Generic and Reflective Debugging Architecture to Support Runtime Visibility and Traceability of Aspects, 2009

[XU05] Xu, Weifeng; Xu, Dianxiang - A Model-Based Approach to Test Generation for Aspect-Oriented Programs, 2005

[BHAT08] Bhatti, Muhammad Usman; Ducasse, Stéphane - Mining and Classification of Diverse Crosscutting Concerns, 2008

[PLDI05] Avgustinov, Pavel; Christensen, Aske Simon ş.a. - Optimising AspectJ, 2005