Profilovanie aplikácií spúšťaných vo virtuálnych strojoch spĺňajúcich Java špecifikáciu

Diplomová práca

Igor Ináš **2006**

Profilovanie aplikácií spúšťaných vo virtuálnych strojoch spĺňajúcich Java špecifikáciu

Diplomová práca

Igor Ináš

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY
KATEDRA INFORMATIKY

Informatika

Školiteľ záverečnej práce
Ing. Radovan Sninský

BRATISLAVA 2006

Čestne vyhlasujem, že som diplomovú prácu vypracoval samostatne, s použitím literatúry a zdrojov uvedených v závere práce.

Bratislava, máj 2006

…...............................

Ďakujem svojmu diplomovému vedúcemu Ing. Radovanovi Sninskému za odborné vedenie práce, cenné rady a pripomienky.

Ďakujem firme Business Global Systems, a.s., jej zamestnancom a mojím kolegom za poskytnutie prostriedkov, priestoru a mnohých podnetných postrehov pri písaní práce.

Abstrakt

Práca prezentuje rôzne pohľady na problematiku profilovania aplikácií spúšťaných v Java virtuálnych strojoch. Objasňuje súvislosti medzi rôznymi technológiami Java platformy, ktoré s profilovaním priamo súvisia. Zameriava na jednak na tie časti špecifikácie, ktoré definujú Javu ako nástroj na vývoj softvéru a popisujú tie vlastnosti Java technológie, ktoré sa odrážajú na štruktúre no najmä behu Java programov. Na druhej strane zase popisuje časti špecifikácie technológie Java, ktoré umožňujú okrem spúšťania programov získavať o ich behu údaje, ktoré sú potrebné na vytváranie profilov. Okrem prehľadu súčasných možností je popísaný aj vývoj, ktorý viedol k súčasnému stavu špecifikácie a dôvody, ktoré sprevádzali rozhodnutia pri jej tvorbe. Práca porovnáva niekoľko profilovacích nástrojov a objasňuje možnosti tvorby vlastných. Odpovedá na otázku, aký prínos môže profilovanie priniesť do procesu vývoja a údržby softvéru. Posledná kapitola zachytáva nasadenie profilovania v praxi, tak ako ho bolo realizované na niekoľkých komerčných softvérových projektoch.

Kľúčové slová: Java, virtuálny stroj, profilovanie, optimalizácia, profilovacie nástroje

Obsah

1 Úvod 3

1.1 Definícia pojmu profilovanie 3

1.2 Členenie práce 3

2 Typy profilov pre Java aplikácie 5

2.1 Základná špecifikácia technológie Java 5

2.2 Java ako systémový proces 6

2.2.1 Profily procesu 7

2.3 Procedurálne programovanie 8

2.3.1 Dynamic call tree 9

2.3.2 Calling context tree 10

2.3.3 Tabuľka volaní metód 12

2.3.4 Method back traces tree 13

2.3.5 Aproximačné verzie profilov 14

2.4 Viacvláknové programovanie 15

2.4.1 Prehľad stavu vlákien v čase 17

2.5 Automatická správa pamäte 19

2.5.1 Graf haldy 19

2.5.2 Tabuľka inštancií tried 20

2.5.3 Vytváranie objektov v metódach 20

2.5.4 Prehľad stavu haldy a GC procesu v čase 21

2.6 Špecializované profily 24

3 Technológie a postupy na získavanie profilovacích údajov 26

3.1 Zdroje profilovacích informácií 26

3.2 Rozšírenia virtuálneho stroja 28

3.3 Inštrumentácia kódu 31

3.3.1 Inštrumentácia zdrojového kódu 33

3.3.2 Inštrumentácia bajtkódu 35

3.4 Podpora profilovania na úrovni Java API 38

3.4.1 Základná výpočtová reflexia 38

3.4.2 Rozšírenia výpočtovej reflexie podľa JSR-163.....41
3.5 Aspektové programovanie.....42
4 Profilovacie nástroje.....44
4.1 HPROF a JHAT.....44
4.2 YourKit profiler.....46
4.3 JProfiler.....47
4.4 NetBeans profiler.....50
4.5 Eclipse TPTP.....52
4.6 Java Interactive Profiler.....53
4.7 Porovnanie profilovacích nástrojov.....53
5 Začlenenie profilovania do procesu vývoja softvéru.....55
5.1 Hľadanie chýb v aplikácii.....55
5.2 Optimalizácia výkonu aplikácie.....56
5.3 Predikcia budúceho správania aplikácie.....57
5.4 Dynamická analýza neznámeho kódu.....57
6 Prípadové štúdie.....58
6.1 Prepaid Billing Server.....58
6.1.1 Popis produktu.....58
6.1.2 Konfigurácia testu.....59
6.1.3 Profilovanie a zistené skutočnosti.....60
6.1.4 Riešenie problému.....60
6.1.5 Záver.....62
6.2 Identifikácia pamäťových problémov.....63
6.3 Identifikácia výkonnostných problémov.....64
7 Záver.....66
8 Zoznam použitej literatúry.....67

1 Úvod

1.1 Definícia pojmu profilovanie

Profil je súbor vlastností, čŕt, množina informácií, často prezentovaná v grafickej forme, ktorá znázorňuje významné vlastnosti pozorovaného objektu [1][2]. V tejto diplomovej práci bude slovo profilovanie používané ako ekvivalent anglického *profiling* s významom „vytváranie profilu".

Profilovanie v počítačovej terminológii je analýza bežiaceho počítačového programu za účelom získania jeho skutočného a nie predpokladaného správania [3]. Je to proces zberu charakteristík programu, ktorý je realizovaný zaznamenávaním udalostí, ktoré prebiehajú v počítačovom systéme počas behu programu. Tieto udalosti môžu byť hardvérového typu (hardvérové prerušenia, systémové hodiny) alebo softvérového typu (volanie funkcií, alokácia pamäte).

Výsledkom je sled zaznamenaných udalostí, ktorý sa označuje pojmom *trace*. Tieto dáta sú ďalej analyzované, zobrazované v zrozumiteľnej forme, aplikujú sa na ne rôzne metriky a finálnym produktom procesu profilovania je profil spusteného programu.

Aký typ informácie o behu programu získaný profil poskytuje samozrejme závisí od triedy programov, do ktorej sledovaný program spadá, od vnútornej architektúry cieľového operačného systému a hardvéru, na ktorom je program vykonávaný. Ak sa porovnajú rôzne triedy programov, napríklad SQL (deklaratívne programovanie), Assembler (neštruktúrované programovanie), LISP (funkcionálne programovanie), Pascal (procedurálne programovanie), Java (objektovo-orientované programovanie), je zrejmé, že vnútorné charakteristiky behu programov z rôznych programových tried sa budú navzájom líšiť. Napríklad v klasickom procedurálnom programe je v prvom rade zaujímavý podiel časov pre jednotlivé procedúry, v ktorých program trávi čas. V SQL príkazoch, ktoré sú spúšťané v prostredí databázových serverov, to je zase reálny počet prístupov k dátam uloženým na fyzickom médiu v porovnaní s prístupom k dátam uložených vo vyrovnávacej pamäti (*cache hit/miss/insert*). Rôzne operačné systémy sa od seba líšia spôsobom správy procesov, prideľovania systémových prostriedkov, správou pamäte. To má za následok rôznorodosť udalostí, ktoré možno v danom OS pozorovať.

1.2 Členenie práce

Druhá a tretia kapitola popisujú špecifikácie a vlastnosti technológie Java. Druhá kapitola sa zameriava na tie časti technológie, ktoré definujú Javu ako nástroj na vývoj softvéru. Popisuje

vlastnosti, ktoré získava aplikácia vyvinutá technológiou Java. Popísané vlastnosti sa odrážajú na jej štruktúre a najmä behu. V kapitole sú definované štruktúry, ktorými sa dajú jednotlivé vlastnosti behu programov formálne popísať. Zavedené štruktúry predstavujú základné profily Java programov.

Tretia kapitola popisuje tie časti špecifikácie technológie Java, ktoré umožňujú okrem spúšťania programov získavať o ich behu údaje, ktoré sú potrebné na vytváranie profilov. Okrem prehľadu súčasných možností je popísaný aj vývoj, ktorý viedol k súčasnému stavu špecifikácie a dôvody, ktoré sprevádzali rozhodnutia pri jej tvorbe. Cieľom kapitoly je objasniť princípy získavania profilovacích údajov, aby si čitateľ mohol vytvoriť predstavu, do akej miery je normálny beh programu ovplyvnený aktiváciou profilovacích mechanizmov. Druhým cieľom je popísať možnosti tvorby vlastných profilovacích nástrojov.

V štvrtej kapitole je opísaných niekoľko v súčastnosti dodávaných profilovacích nástrojov. Pre každý nástroj je uvedená jeho základná architektúra, spôsoby použitia a typy profilov, ktoré poskytuje. Záver kapitoly sa venuje porovnaniu jednotlivých nástrojov na základy rôznych kritérií.

Piata kapitola odpovedá na otázku, aký prínos môže profilovanie priniesť do procesu vývoja a údržby softvéru. Uvádza problémy spojené s vývojom a údržbou, ktoré môže profilovanie pomôcť odstrániť ale aj riziká, ktoré profilovanie prináša.

Šiesta kapitola zachytáva nasadenie profilovania v praxi, tak ako ho realizoval autor tejto práce na skutočných komerčných softvérových projektoch. Okrem niekoľkých optimalizácii lokálneho charakteru, ktoré vychádzali zo skutočností odhalených profilovaním, sa kapitola venuje aj pomerne rozsiahlemu procesu profilovania produktu, ktorý obsahoval viacero dizajnových a implementačných nedostatkov.

2 Typy profilov pre Java aplikácie

2.1 Základná špecifikácia technológie Java

Základom technológie Java je definícia abstraktného výpočtového prostredia, ktoré sa označuje pod názvom Java virtuálny stroj (*Java virtual machine*), ďalej len JVM. Rovnako ako fyzický hardvér, aj JVM má svoju množinu inštrukcií, disponuje pamäťou na vytváranie a manipuláciu s dátami, má definovaný prístup k vstupno-výstupným zariadeniam [4]. Samotné detaily implementácie JVM nie sú nijako predpísané, jedinou podmienkou je dodržanie pôvodnej špecifikácie. Od vzniku technológie sú jej typické implementácie softvérového charakteru. V poslednom čase, kedy sa Java stáva populárnou platformou pre mobilné a iné zariadenia, sa výskum zameriava aj na hardvérové implementácie platformy [5].

Ďalej špecifikácia definuje samotný kód určený na spustenie v JVM, označovaný pojmom bajtkód (*bytecode*). Jeho formát je binárny, a jedna ucelená časť bajtkódu definuje tzv. triedu (*class*). Zvyčajne je bajtkód programu dodávaný v množine súborov s príponou *class,* v každom súbore je definícia práve jednej triedy. Aj keď bajtkód nie je programovacím jazykom v pravom slova zmysle, sú v ňom definované vyššie štruktúry v súlade s objektovo orientovaným programovaním.

Nakoniec definuje špecifikácia štandardnú množinu tried, ktoré majú byť spolu so samotnou JVM dodávané a byť k dispozícii pri každom spustení JVM. Táto množina tried sa nazýva API (*application programming interface*). V súčastnosti existujú tri štandardné množiny API tried, ktoré sa od seba líšia svojím rozsahom a účelom, a to J2SE (*standard edition*), J2EE (*enterprise edition*) a J2ME (*micro edition*) API. Táto práca sa venuje primárne profilovaniu aplikácií určených pre edíciu J2SE, aj keď prestavené nástroje a techniky profilovania sú rovnako použiteľné pre J2EE.

Samotný programovací jazyk Java je popísaný v samostatnom dokumente [6]. Striktne povedané, JVM nevie nič o programovacom jazyku Java, pre JVM je jediným známym formátom bajtkód. V JVM je možné spustiť program napísaný v ľubovoľnom programovacom jazyku, pre ktorý existuje kompilátor do bajtkódu. Jazyk Java však bol samozrejme vyvinutý na implementáciu programov určených primárne pre JVM, preto syntax jazyka Java definuje rovnaké štruktúry, aké definuje bajtkód. Štandardne dodávané súčasné verzie kompilátorov jazyka Java vyrábajú zo všetkých tried, ich metód a polí ich reprezentáciu v bajtkóde. Zachované sú ich presné mená. Formát bajtkódu môže navyše obsahovať pomocné informácie[1] o štruktúre zdrojového kódu, z ktorého vznikol.

1 Názov zdrojového súboru a pre každú inštrukciu riadok v pôvodnom súbore.

Proces dekompilácie je potom priamočiary a jeho výsledkom je zdrojový kód skoro úplne zhodný s pôvodným kódom. Samozrejme je možné vytvoriť kompilátor, ktorý štruktúru kódu nezachová. Príkladom sú staršie verzie kompilátora dodávaného s JVM od firmy SUN, ktoré mali možnosť optimalizovať výsledný bajtkód[2]. Iným príkladom je tzv. *obfuscator* bajtkódu, ktorým možno premenovať mená objektov tak, aby sa sťažila analýza a modifikácia zdrojového kódu po dekompilácii. Sú to však len špeciálne prípady a implicitne sa bude predpokladať jednoduchá kompilácia do bajtkódu bez dodatočných modifikácií.

Profilovanie sa zaoberá dynamickou analýzou bežiaceho kódu. V prípade JVM je teda analyzovaný bajtkód a výsledky profilovania popisujú jeho vlastnosti. Z práve uvedeného vyplýva, že všetky odvodené vlastnosti bajtkódu sú zároveň vlastnosťami zdrojového kódu napísaného v jazyku Java. Toto je veľmi dôležitý záver. Profilovanie môže odhaliť neželané vlastnosti programu a za účelom ich odstránenia je výhodné vedieť, ktoré časti zdrojového kódu zistenú vlastnosť spôsobujú. Fakt, že kód, ktorý vstupuje do JVM popisuje väzby na pôvodný zdrojový kód, je dôležitým predpokladom toho, že výsledok profilovania priamo poukáže na problematické miesta zdrojového kódu, bez toho, aby bolo nutné tieto väzby identifikovať dodatočne na základe manuálnej analýzy. Okrem toho popisuje druhá kapitola Javu z programátorského pohľadu, ako nástroj na vývoj aplikácií. V ďalších kapitolách bude preto používaný pojem „príkaz jazyka Java“ ako synonymum pojmu „inštrukcia bajtkódu“.

2.2 Java ako systémový proces

Princíp virtuálneho stroja umožňuje Java aplikáciam ich platformovú nezávislosť. V praxi to znamená to, že aplikáciu napísanú v jazyku Java stačí raz skompilovať do bajtkódu a výsledný bajtkód potom možno spustiť na ľubovoľnom hardvéri a pod ľubovoľným operačným systémom, pre ktorý existuje implementácia JVM. Hardvér spolu s nasadeným operačným systémom sa označuje pojmom platforma. Na to, aby bolo možné na cieľovej platforme úspešne implementovať JVM v plnej miere, platforma musí poskytovať funkcionalitu, ktorú si vyžaduje špecifikácia JVM.

V prvom rade obsahuje Java príkazy na riadenie behu programu (cykly, podmienky, ...), príkazy na definovanie a manipuláciu s dátami uloženými v pamäti (primitívne typy, objekty, premenné, polia, …), aritmetické operácie a iné. Vykonávanie týchto príkazov vyžaduje na cieľovej platforme prítomnosť jedného alebo viacerých procesorov a operačnej pamäte, tak, ako ich definuje von Neumannovská architektúra Súčasťou J2SE API sú ďalej triedy, ktoré poskytujú funkcionalitu pre

2 Dôvodom bol fakt, že prvé implementácie JVM fungovali čisto na princípe interpretácie bajtkódu [7]. Nové verzie JVM kompilujú bajtkód za behu do natívneho kódu pre daný operačný systém, na ktorom je JVM nasadený a optimalizácie prebiehajú aj na základe dynamických vlastností doterajšieho behu programu.

prácu so súborovým systémom, prístup k vstupno výstupným zariadeniam (klávesnica, myš, konzola) a pre prístup k sieťovým jednotkám prostredníctvom vyšších protokolov (TCP, UDP).

Image Name	CPU	Mem Usage	Handles	Threads	I/O Read Bytes	I/O Write Bytes
soffice.bin	00	66 484 K	190	8	28 517 205	17 712 660
firefox.exe	00	51 696 K	245	9	15 208 324	23 559 436
javaw.exe	22	44 372 K	632	14	14 723 053	610
explorer.exe	00	25 032 K	419	11	145 671 009	22 929
svchost.exe	00	24 400 K	1 484	66	8 626 314	6 509 905
nod32krn.exe	00	19 544 K	215	16	335 809 502	35 250 899
Apache.exe	00	14 844 K	586	254	146 706	227
winamp.exe	02	10 704 K	293	9	302 432 535	1 056 747

Obrázok 1: Task Manager operačného systému WindowsXP

V konečnom dôsledku je JVM s bežiacim Java programom jeden alebo viacero procesov v rámci hosťovského operačného systému cieľovej platformy. Ako každý iný proces, aj JVM procesy sa uchádzajú o všetky spomenuté zdroje, ktorých rozdeľovanie je zvyčajne v réžii operačného systému (obrázok 1).

2.2.1 Profily procesu

Prvé typy profilov uvádzané v tejto práci popisujú podiel JVM procesu na celkovom zaťažení systému. Formálne možno tieto profily zadefinovať ako funkcie s parametrom času, kde $t=0$ je začiatok behu JVM procesu.

- Zaťaženie procesora popisuje funkcia *ProcessorTime(t)*, ktorej funkčnou hodnotou je čas, ktorý procesor od štartu aplikácie strávil vykonávaním inštrukcií JVM procesu. Funkcia je neklesajúca.

- Využitie pamäte popisuje funkcia *AlocatedMemory(t)*, ktorej obor hodnôt je počet bajtov, ktoré mal JVM proces v danom momente alokovaný v operačnej pamäti.

- Prístup k súborovému systému a sieťovým zdrojom popisuje dvojica funkcií *DiskIOBytes(t)* a *NetworkIOBytes(t)*, ktorých obor hodnôt sú dvojice s počtom bajtov, ktoré proces počas celej doby svojho behu čítal alebo zapísal zo súborového systému alebo siete. Aj tieto funkcie majú neklesajúci charakter.

Monitorovanie systémových procesov je rozsiahla, samostatná problematika. Bežiace procesy sú charakterizované mnohými inými, platformovo závislými vlastnosťami. Štyri popísané profily boli uvedené ako profily procesov, ktoré sú spoločné pre rôzne platformy a implementácie JVM. Možné sú ich modifikácie a úrovne detailov informácií.[3]

3 Doplňujúce informácie o práci so súbormi a sieťou sú napríklad počet otvorených súborov, sieťových spojení,

Z uvedeného vyplýva prvý dôležitý fakt, ktorý sa týka profilovania. Bez toho aby boli skúmané vnútorné procesy v JVM, beh JVM má dopad na celkové zaťaženie cieľovej platformy. Profilovaný program môže vykazovať slabý výkon a pomalý beh aj v niektorých prípadoch, kedy proces JVM zaťažuje procesorový čas oveľa menšou mierou, ako by sa pri prvých úvahách čakalo. Vo všeobecnosti je dôležité vedieť dať všetky zistené skutočnosti o vnútornom behu JVM procesu do súvislosti s jeho prejavmi na úrovni operačného systému. Negatívne vonkajšie prejavy JVM procesu sú prvými signálmi toho, že sa aplikácia, ktorá v JVM beží, nespráva podľa očakávaných predstáv.

2.3 Procedurálne programovanie

Program napísaný v jazyku symbolických adries (*assembly language*) je typickým zástupcom neštruktúrovaného programovania. Program je jednoduchý zoznam po sebe idúcich inštrukcií, skok iný, ako na nasledujúcu inštrukciu zabezpečujú inštrukcie skoku. Ak by sme chceli analyzovať beh takéhoto programu, výsledkom by mohla byť postupnosť inštrukcií tak, ako boli vykonávané v čase. Na základe takejto postupnosti by bolo potom možne vytvoriť jej kompaktnejšie reprezentácie. Príkladom by mohla byť tabuľka, ktorá by pre každú inštrukciu obsahovala počet jej vykonaní (hovoríme o konkrétnej inštrukcii, nie type inštrukcie). Ak by sme chceli analyzovať skupinu inštrukcií ako celok, museli by sme upresniť to, čo pojem celok predstavuje. Jazyk symbolických adries pojmy ako blok alebo funkcia nepozná. Presnejšie povedané, nie sú súčasťou syntaxe jazyka, aj keď programátor môže pri organizácii zdrojového kódu zvoliť metodiku štruktúrovaného programovania [8].

Vlastnosťou programov napísaných vo vyšších jazykoch ako C alebo Pascal je procedurálny prístup. Ako už bolo naznačené v úvode, pri profilovaní je najdôležitejším dôsledkom tohto prístupu hľadisko organizácie kódu. Príkazy v procedurálnych programoch sú rozdelené do procedúr a volanie jednej procedúry z iných miest programu je základným konceptom. Jedna procedúra spravidla obsahuje príkazy, ktoré implementujú nejakú vyššiu funkcionalitu. Procedúra môže volať ďalšie procedúry (aj samú seba) a po vykonaní posledného príkazu vracia riadenie na miesto, odkiaľ bola zavolaná. Java je objektovo-orientovaný jazyk, z hľadiska štruktúry kódu je však len špeciálnym prípadom procedurálneho programovania. V Java programe sú procedúry podľa konvencie nazývané metódy, navyše sú zaradené do hierarchie podľa príslušnosti k triedam a balíkom a majú definovanú viditeľnosť z iných miest programu.

---

veľkosť prenesených dát pre jednotlivé spojenia, atď...

2.3.1 Dynamic call tree

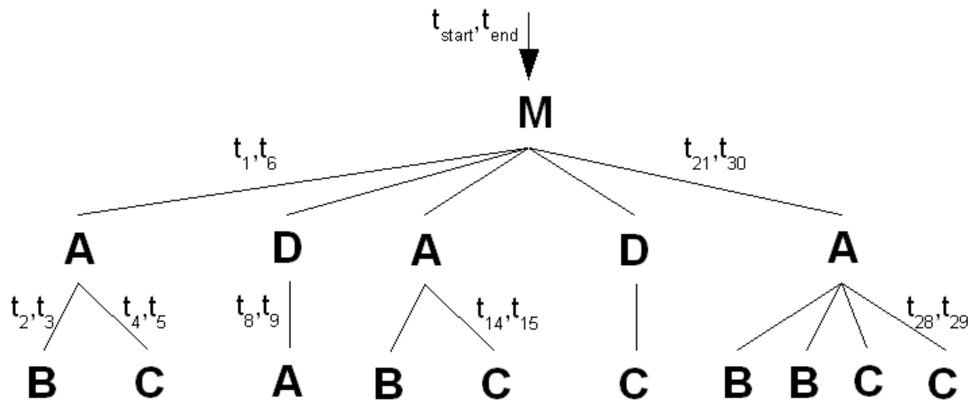
Nech sú všetky metódy programu označené symbolmi $m_1, m_2, \dots m_n$. Nech je volanie procedúry m_i označené trojicou $<t, enter, m_i>$ a ukončenie trojicou $<t, exit, m_i>$, kde t je čas, ktorý uplynul od štartu programu, hodnoty *enter* a *exit* druhého parametra označujú vstup do metódy (respektíve jej zavolanie) a ukončenie metódy. Trojice sa budú označovať ako udalosti. Beh ľubovoľného programu z hľadiska volania procedúr možno potom popísať konečnou postupnosťou udalostí, ktorá spĺňa nasledovné podmienky:

1. Postupnosť sa začína udalosťou $<0, enter, m_{main}>$ a $<t_{end}, exit, m_{main}>$, kde m_{main} je zvolená vstupná metóda programu a t_{end} je čas ukončenia behu programu.

2. Pre každé dve po sebe nasledujúce udalosti $<t_1, \{enter \mid exit\}, m_i>$ a $<t_2, \{enter \mid exit\}, m_j>$ platí $t_1 \leq t_2$.

3. Udalosti medzi prvou a poslednou udalosťou možno rozdeliť na nula až viacero podpostupností takých, že prvým prvkom každej podpostupnosti je $<t_1, enter, m_i>$ a posledným $<t_2, \{exit\}, m_i>$ pre nejakú metódu m_i. Pre udalosti medzi prvým a posledným prvkom podpostupnosti platí opäť podmienka č.3.

Na základe rekurzívnej aplikácie pravidla č.3 možno povedať, že každá postupnosť, ktorá opisuje beh programu je „správne uzátvorkovaná". Vyplýva to z toho, že žiadna metóda sa nemôže ukončiť, pokiaľ neboli ukončené všetky metódy ňou zavolané. Postupnosť má aj grafovú reprezentáciu, ktorú definoval J. R. Larus [9] pod názvom *dynamic call tree (DCT)*. Vrcholmi v *DCT* strome sú metódy, koreňom je vstupná metóda programu, hrany medzi vrcholmi reprezentujú volanie metód. Metóda, ktorá je bližšie ku koreňu je vždy volajúcou metódou. Každá hrana ma priradenú dvojicu časových údajov. Prvý údaj zaznamenáva čas volania, druhý údaj čas návratu. Rozdiel dvoch časových údajov vyjadruje čas, ktorý trvalo vykonávanie volanej metódy (obrázok 2). *DCT* strom má konečnú hĺbku, ohraničuje ju maximálna veľkosť zásobníka pre volania metód.

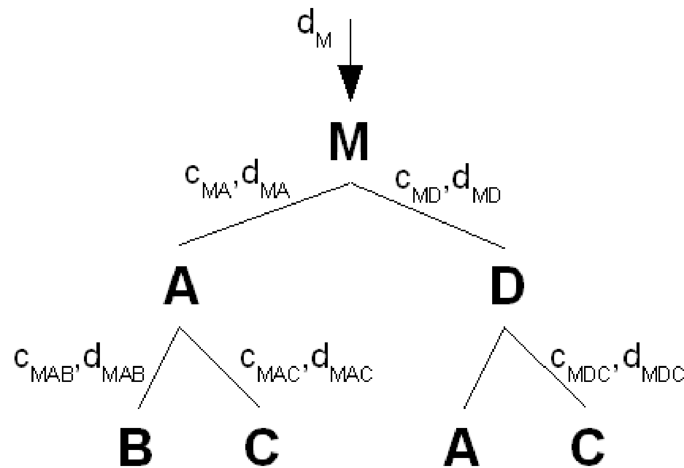


Obrázok 2: Dynamic call tree (DCT)

2.3.2 Calling context tree

Postupnosť udalostí[4] alebo *DCT* strom možno nazvať profilom aplikácie. Exaktne popisuje celé správanie aplikačného kódu. Problém však predstavuje množstvo dát, ktoré tieto štruktúry predstavujú. V bežnom prípade ide o rádovo tisíce volaní metód a návratov z nich. Aby bolo možné z týchto údajov vyčítať relevantné informácie, je potrebné previesť ich do nejakej kompaktnej formy. Larus definoval kompaktnú verziu *DCT* stromu, ktorý nazval *calling context tree (CCT)*. Pojem kontext má v tomto prípade význam stavu zásobníka volaní metód, v Jave označený pojmom *stacktrace*. V *CCT* sú všetky cesty od koreňa k jeho nasledovníkom jedinečné postupnosti volaní metód. Každá cesta zjednocuje cesty z *DCT*, ktoré patria do rovnakej triedy ekvivalencie podľa postupnosti volaní. Jedná cesta v *CCT* však stále agreguje veľké množstvo dvojíc časových údajov, ktoré patrili k cestám v *DCT*. Priradená jej bude preto nová dvojica, a to počet ciest v pôvodnej triede ekvivalencie a časový údaj, ktorého hodnota je suma rozdielov časových dvojíc pre všetky pôvodné cesty. Prvý údaj reprezentuje počet volaní metódy z daného kontextu, druhá suma reprezentuje čas, ktorý volajúca metóda v danom kontexte strávila pri všetkých volaniach tej istej metódy (obrázok 3).

4 Postupnosť udalostí sa označuje anglickým pojmom *trace*.



Obrázok 3: Context call tree (CCT)

CCT strom ako typ profilu adoptovali všetky v súčastnosti dostupné profilovacie nástroje (obrázok 4). Miera informácii, ktoré poskytuje o výkone aplikácie sa ukazuje ako postačujúca. *CCT* strom rozdeľuje beh aplikácie podľa kontextu volaní metód. Situácie, kedy sa výpočet nachádza v tom istom kontexte sú zvyčajne charakterizované stavmi, ktoré zdieľajú rovnaké alebo podobné atribúty. Ak nejaká vetva výpočtu vykazuje pomalú prácu, je veľmi pravdepodobné, že údaje, ktoré do tejto vetvy vstupujú na spracovanie majú rovnaké charakteristiky. Agregovanie nameraných metrík pre volania v tom istom kontexte preto zvyčajne nespôsobuje stratu hodnoty informácií o výkone v danom mieste programu[5].

Call Tree - Method	Time [%]	Time	Invocations
All threads		929 ms (100%)	1
main		929 ms (100%)	1
com.toy.anagrams.ui.Anagrams.**main** (String[])		929 ms (100%)	1
com.toy.anagrams.ui.Anagrams.<**init**> ()		750 ms (80,7%)	1
javax.swing.JFrame.<**init**> ()		383 ms (41,2%)	1
com.toy.anagrams.ui.Anagrams.**initComponents** ()		192 ms (20,7%)	1
java.awt.Window.**pack** ()		113 ms (12,2%)	1
javax.swing.text.JTextComponent.**setText** (String)		54.8 ms (5,9%)	1
javax.swing.JComponent.**requestFocusInWindow** ()		1.61 ms (0,2%)	1
java.lang.ClassLoader.**loadClassInternal** (String)		1.59 ms (0,2%)	4
javax.swing.JRootPane.**setDefaultButton** (javax.swing.JButton)		1.27 ms (0,1%)	1
Self time		0.639 ms (0,1%)	1
java.awt.Component.**setLocation** (java.awt.Point)		0.550 ms (0,1%)	1
com.toy.anagrams.lib.WordLibrary.<**clinit**>		0.224 ms (0%)	1
sun.awt.SunToolkit.**getScreenSize** ()		0.133 ms (0%)	1

Obrázok 4: Netbeans Profiler - call tree

2.3.3 Tabuľka volaní metód

Alternatívou k deleniu výpočtu podľa kontextu volaní metód je delenie podľa trávenia času v jednotlivých metódach bez ohľadu na kontext. Profil definuje funkcia *MethodProfile(m)*. Jej

5 Výnimku môže predstavovať situácia, kedy sa v tele metódy nachádza ako viacero samostatných príkazov volanie tej istej metódy. Je zaujímavé, že *CCT* strom podľa Larusa a všetky jeho súčasné implementácie v profilovacích nástrojoch medzi týmito volaniami nerozlišujú a ich volania definujú ten istý kontext. V tejto situácii môže zjednotenie nameraných metrík pre dva rôzne príkazy volania spôsobiť spriemerovanie dvoch rôznych sád výsledkov a zakryť to volanie, ktorého priemerná dĺžka volania predstavuje vážny výkonnostný problém.

vstupným parametrom je metóda programu, výstupom dvojica čísel. Prvým je počet volaní metódy z ľubovoľného kontextu, druhým je súhrnný čas, ktorý trvali všetky volania zadanej metódy. Prípady, kedy sa v programe vyskytne priame alebo nepriame rekurzívne volanie (stav, kedy sa v zásobníku volaní objaví tá istá metóda viackrát) komplikujú presnú definíciu funkcie. Priamočiare odvodenie funkcie z *CCT* stromu znamená sčítať počty a dĺžky všetkých volaní metódy zo všetkých kontextov. Toto by však zaviedlo do výsledkov anomáliu pre rekurzívne volané funkcie. Problematické sú kontexty, v ktorých sa volanie jednej metódy na ceste smerom od koreňa v CCT strome vyskytuje viackrát. Definíciu funkčných hodnôt preto treba upraviť tak, aby sa pri prechádzaní stromu bralo pri sčítavaní do úvahy len prvé nájdené volanie zadanej metódy v každom kontexte. Funkčnej hodnote funkcie môžno stále pridať tretiu zložku, a to počet volaní funkcie vrátane rekurzívnych volaní.

Profil sa zvyčajne graficky znázorňuje tabuľkou volaní metód. Vôbec jeden z prvých profilovacích nástrojov, ktoré vznikli, je *prof* dodávaný pre *UNIX*-ové platformy [10]. Nástroj vytvára jednoduchú tabuľku. Každý riadok tabuľky zobrazuje jednu procedúru, počet jej volaní, priemerný čas trvania behu, percentuálne vyjadrený čas, ktorý v nej program trávil (obrázok 5). Moderné profilovacie nástroje spravidla poskytujú rovnaký typ tabuľky, či už v textovej alebo grafickej forme.

Name	%Time	Seconds	Cumsecs	#Calls	msec/call
.scatter	17.7	10.35	10.35	28668877	0.0004
.ran3	17.2	10.06	20.41	86286632	0.0001
.__mcount	16.2	9.48	29.89		
.sqrt	16.1	9.44	39.33		
.doAnisotropicScatte	7.4	4.36	43.69	7	623.
.getRandomNumber	7.1	4.15	47.84	86286631	0.0000
._log	6.5	3.81	51.65		
.path	5.7	3.36	55.01	28738877	0.0001
.cos	4.2	2.46	57.47		
.sin	1.1	0.65	58.12		

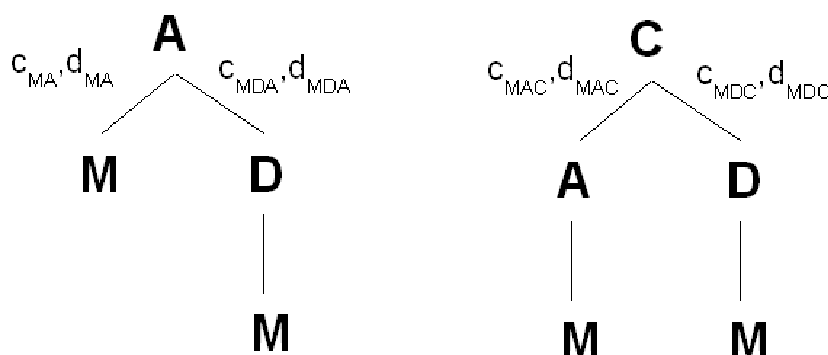
Obrázok 5: UNIX prof tool - tabuľka volaní procedúr

Tabuľka volaní procedúr bola svojho času jediným typom dostupného profilu pre analýzu výkonnosti kódu. Ako spomínajú tvorcovia ďalšieho profilovacieho nástroja *gprof* [11], uvedený profil sa z odstupom času stál menej použiteľným. Komplexita programov sa postupne zvyšovala a prirodzenou reakciou bolo delenie programu na znovupoužiteľné časti kódu, zdieľané na početných miestach v programe. Na pochopenie správania programu už nestačil jednoduchý profil, ktorý poskytoval *prof*. Nezohľadňoval rozdiely medzi volaním jednej konkrétnej funkcie z rôznych miest programu. Ak *prof* označil procedúru za problematickú, nijako nepoukázal na miesta v programe, odkiaľ je procedúra volaná. *Gprof* ako náhrada starého nástroja *prof* už vedel vytvárať aj *CCT* strom. Napriek tomu ostáva dodnes tabuľka volaní procedúr dôležitým profilom. Oproti ostatným profilom prináša tú výhodu, že zjednotením výsledkov metrík pre jednu procedúru a ich

zobrazením na jednom mieste vie v niektorých prípadoch poukázať na problém, ktorého indikácie by boli inak roztrúsené na rôznych miestach v CCT strome.

2.3.4 Method back traces tree

Nakoniec možno z *CCT* stromu odvodiť typ stromu, ktorý sa zameriava na jednú zvolenú metódu. Profilovacie nástroje grafickú reprezentáciu tohto stromu označujú ako *method back traces view* (obrázok 7). Ako dátová štruktúra bude v tejto práci zadefinovaný profil *method back traces tree (MBTT)*. Koreňom *MBTT* stromu je zvolená metóda a vychádza z nej hrana pre každú metódu, odkiaľ bola zavolaná. Rovnaké pravidlo platí pre volajúce metódy. Listami stromu sú preto vždy vrcholy, ktoré reprezentujú vstupnú metódu programu (obrázok 6). Cesty od koreňa k listom sú obrátené kontexty. Hrany pri koreňovom vrchole majú rovnakú dvojicu atribútov, ako boli definované v CCT strome pre príslušný kontext. Ostatné hrany sú neoznačené. Aj v tomto prípade komplikujú situáciu priame, či nepriame rekurzívne volania. Vtedy je výhodnejšie vytvárať *MBTT* strom priamo z *DCT* stromu a pre každý kontext ignorovať všetky volania zvolenej metódy, okrem posledného na ceste smerom od koreňa v *DCT* strome.



Obrázok 6: MBTT strom pre metódy A a C

Name	Time (ms)		Invocation Count	
Intersection$Demo.paint(Graphics)	10 735	100 %	1 049	100 %
javax.swing.JComponent.paintWithOffscreenBuffer(JComponent, Graph	10 735	100 %	1 048	100 %
javax.swing.JComponent.paintDoubleBuffered(JComponent, Compor				
javax.swing.JComponent._paintImmediately(int, int, int, int)				
javax.swing.JComponent.paintImmediately(int, int, int, int)				
javax.swing.RepaintManager.paintDirtyRegions()				
javax.swing.SystemEventQueueUtilities$Componer				
java.awt.event.InvocationEvent.dispatch()				
javax.swing.JComponent.paintChildren(Graphics)	0	0 %	1	0 %

Obrázok 7: YourKit Profiler - method back traces view

2.3.5 Aproximačné verzie profilov

CCT strom, *MBTT* strom a tabuľka volaní metód boli všetky odvodené z *DCT* stromu. *DCT* strom možno vytvoriť, len keď je k dispozícii postupnosť udalostí o volaniach a návratoch z metód. Táto postupnosť musí byť v konzistentnom stave, nesmú v nej chýbať náhodne vybrané udalostí. Pri neúplnej postupnosti nemožno správne skonštruovať *DCT* strom a teda ani *CCT* a *MBTT* stromy, pretože chýbajúce udalosti by znemožnili správne skonštruovať aktuálny kontext volania metód. Vo všeobecnom prípade by nebolo možné ani skonštruovať tabuľku volaní metód s približnými hodnotami. Z postupnosti udalostí by stačilo odstrániť prvú a poslednú udalosť o volaní metódy, ktorá sa v programe vykonala napríklad len raz, ale trvala veľmi dlho. V tabuľke by nakoniec vôbec nefigurovala, hoci by predstavovala miesto, kde výpočet strávil dlhý čas.

Verzie troch zavedených profilov, ktoré by aplikáciu popisovali so zníženou presnosťou, možno skonštruovať z iného zdroja ako *DCT* stromu. Týmto zdrojom je vzorkovacia funkcia *SamplingFunction(t)*. Definičný obor funkcie je postupnosť časov, v ktorých boli vzorky odoberané. Postupnosť je pravidelná, medzi jednotlivými hodnotami je rovnaký rozdiel. Rozdiely charakterizuje zvolená frekvencia vzorkovania. Jej funkčné hodnoty sú aktuálne kontexty volania metód v zadanom čase. Z údajov vzorkovacej funkcie možno priamočiaro odvodiť aproximácie každého z troch profilov. V prípade tabuľky volania metód by dokonca stačilo, ak by vzorkovacia funkcia vracala len metódu, v ktorej sa výpočet nachádzal. Vo všetkých profiloch by úplne chýbal údaj o počte volaní metód. Údaj o súhrnnej dĺžke volania metód by bol vyrátaný ako podielu z celkového času trvania vzorkovania (obrázok 8). Ak napríklad vzorkovanie trvá jednu sekundu, vzorky sú odoberané každých 50 milisekúnd a z 20 odobraných vzoriek polovica ukazuje, že aplikácia sa práve nachádza v tej iste metóde, možno z toho odvodiť záver, že aplikácia trávila vykonávaním príkazov metódy súhrnný čas približne pol sekundy.

[index]	secs	%	cum.%	samples	function (dso: file, line)
[1]	1020.180	43.6%	43.6%	102018	__pow (libm.so: pow.c, 198)
[2]	721.570	30.8%	74.4%	72157	__log (libm.so: log.c, 139)
[3]	358.100	15.3%	89.7%	35810	__exp (libm.so: exp.c, 102)
[4]	235.820	10.1%	99.8%	23582	asa (asa_run: asa.c, 59)
[5]	1.310	0.1%	99.9%	131	write_last_best (asa_run: getpd.c,566)
[6]	0.990	0.0%	99.9%	99	loadlists (asa_run: getpd.c, 172)
[7]	0.960	0.0%	99.9%	96	cost_derivatives (asa_run: asa.c,2229)
[8]	0.540	0.0%	100.0%	54	print_state (asa_run: asa.c, 2762)
[9]	0.210	0.0%	100.0%	21	_free (libc.so.1: malloc.c, 903)
[10]	0.100	0.0%	100.0%	10	write_profile (asa_run: getpd.c,629)

Obrázok 8: Prof - tabuľka volania metód, aproximačná verzia

Dôvod prečo sa vôbec s aproximačnými verziami profilov zaoberať je fakt, že získať úplnú

postupnosť udalostí môže byť výpočtovo náročná operácia. V niektorých prípadoch sa programátor nezaujíma o presné počty volaní procedúr, ale iba o približné rozdelenie časov pre jednotlivé procedúry. Konkrétne ide o hrubú výkonnostnú analýzu, kedy sa hľadajú takzvané horúce miesta (*hotspots*), teda miesta, kde program trávi najviac času. Opačným prípadom sú situácie kedy nie sú zaujímavé časové charakteristiky procedúr, a ide len o presné počty ich volaní. Pri získavaní postupnosti udalostí by takto nebolo potrebné zaznamenávať čas výskytu udalosti, čo znamená ušetrenie práce. Extrémne hodnoty pri volaniach niektorých procedúr môžu opäť signalizovať výkonnostné problémy, prípadne nevhodne navrhnutý dizajn. Primárne sa však programátor zameriava na správnosť programu. Pri daných vstupných dátach, ktoré má program spracovať, má programátor predstavu o správaní programu a profil s presným počtom volaní procedúr mu umožní predpokladané správanie porovnať so správaním skutočným.

2.4 Viacvláknové programovanie

Java je jazyk, ktorý priamo podporuje vytváranie tzv. programových vlákien (*threads*). Vlákna predstavujú abstrakciu paralelne bežiacich úloh. Vytvorenie vlákna je veľmi jednoduché a prirodzené. *Thread* je pre programátora v Jave obyčajná trieda zo štandardného API. V programe stačí vytvoriť novú inštanciu tejto triedy a ako parameter definovať časť kódu, ktorá má byť vo vlákne vykonávaná. Konkrétna realizácia JVM zvyčajne implementuje beh vlákien technikou *time-slicing* v rámci jedného systémového procesu, vo viacprocesorovom prostredí môžu byť jednotlivé vlákna priraďované jednotlivým procesorom, možná je kombinácia predchádzajúcich [12][13]. Pri štarte vytvára JVM jediné hlavné vlákno, v ktorom sa zavolá *main* metóda zvolenej triedy. V hlavnom vlákne možno vytvoriť nové vlákna a spustiť v nich zvolené metódy. Po vykonaní posledného príkazu sa činnosť vlákna končí. JVM beží dovtedy, pokiaľ v ňom existuje aspoň jedno vlákno s aktívne bežiacim kódom.

Beh Java programu preto v skutočnosti nepopisuje jedna postupnosť udalostí, ale práve toľko postupností, koľko vlákien počas behu JVM existovalo. Pre všetky vlákna iné ako hlavné vlákno platí, že ich vstupná metóda je *Thread.run()*. Na základe tejto skutočnosti je potrebne modifikovať profily zadefinované v predchádzajúcich kapitolách. Existujú dve možnosti. Pre každé vlákno vytvoriť samostatný profil alebo jeden spoločný profil, či už ide o *CCT* strom, *MBTT* strom alebo tabuľku metód. V spoločnom profile sa potom akumulujú výsledky nameraných metrík pre kontexty volania metód, ktoré sa vyskytujú vo viac ako jednom vlákne. Väčšina profilovacích nástrojov poskytuje obidve možnosti.

Využívanie vlákien však prináša svoje nástrahy. Objekt *Thread* sa označuje ako *heavy-weighted*,

kedže jeho vytvorenie spravidla súvisí s alokáciou systémových prostriedkov. Neustále vytváranie nových inštancií nie je časovo a výpočtovo najlacnejšia operácia. Taktiež situácia, kedy je v Java aplikácii v jednom momente vytvorených príliš veľa vlákien znamená nevyhnutné zníženie výkonu. Aj v prípade, že mnohé s vlákien sú v nečinnom stave, ich správa sa nikdy nezaobíde bez nutnej réžie. Riešením problému, ktorý bol načrtnutý, je znovupoužitie *Thread* objektu, ktorý skončil vykonávanie svojho kódu. Namiesto toho, aby bol zahodený, môžeme mu zadať novú úlohu. V praxi sa tento prístup nazýva *thread-pooling*. Profilovanie by mohlo odhaliť stavy programu, kedy počas dlhej doby existuje veľa súbežne existujúcich vlákien, no vždy iba podmnožina z nich reálne vykonáva nejakú činnosť, a ostatné spia. Odhaliť by bolo možné množstvo krátko žijúcich vlákien, pričom opäť len konečný počet z nich existuje naraz v ľubovoľnom momente. Pravda je však taká, že tieto stavy vyplývajú priamo z dizajnu aplikácie, majú deterministický charakter a výsledok profilovania neprináša nové skutočnosti. Optimalizovať tieto stavy znamená vychádzať z úplne nového dizajnu.

Použitie vlákien však môže viesť k nedeterministickému správaniu, kde sledovanie dynamického správania pomocou profilovania už znamená prínos. Škálovanie práce aplikácie do vlákien[6] nie je priamočiara záležitosť. Prípadová štúdia v kapitole 6.1 ukazuje ako profilovanie odhalilo, prečo aplikácia zodpovedná za spracovanie údajov v reálnom čase nestíhala vykonávať činnosť napriek tomu, že dostupná výpočtová sila ostávala nevyužitá. Vlákna nie vždy pracujú nad svojou vlastnou množinou dát a pre rôzne vlákna môže existovať potreba prístupu k tomu istému objektu. Ak je súčasťou prístupu modifikácia objektu a táto modifikácia nie je atomická operácia, je neprípustné, aby dve rôzne vlákna vykonávali modifikáciu v tom istom čase.

Synchronizácia práce vlákien je priamo podporovaná v jazyku Java. Z tohto hľadiska sú dôležité metódy *Thread.sleep(), Object.wait() a Object.notify()* a kľúčové slovo *synchronized* [14]. Prvá metóda uvedie na stanovený počet milisekúnd vlákno do nečinného stavu. Druhá metóda robí to isté, vlákno po jej zavolaní zaspí na stanovenú dobu, činnosť vlákna však môže byť predčasne obnovené iným vláknom, ktoré zavolá metódu *notify()* na tom istom objekte. Dvojica metód je určená na implementáciu kooperujúcich procesov. *Synchronized* umožňuje na úrovni syntaxe Jazyka definovať bloky príkazov, ktoré môžu byť v jednom momente vykonávané najviac v jednom vlákne.

2.4.1 Prehľad stavu vlákien v čase

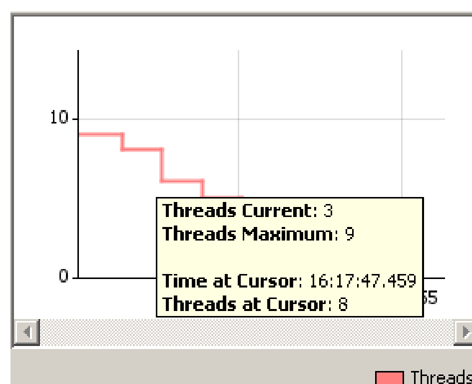
Z uvedeného vyplýva, že vlákna sa v čase nachádzajú v rôznych stavoch. Tieto stavy popisuje

6 Inými slovami paralelné výpočty.

profil reprezentovaný funkciou *ThreadState(thread, time)*. Funkcia vracia nasledovné funkčné hodnoty:

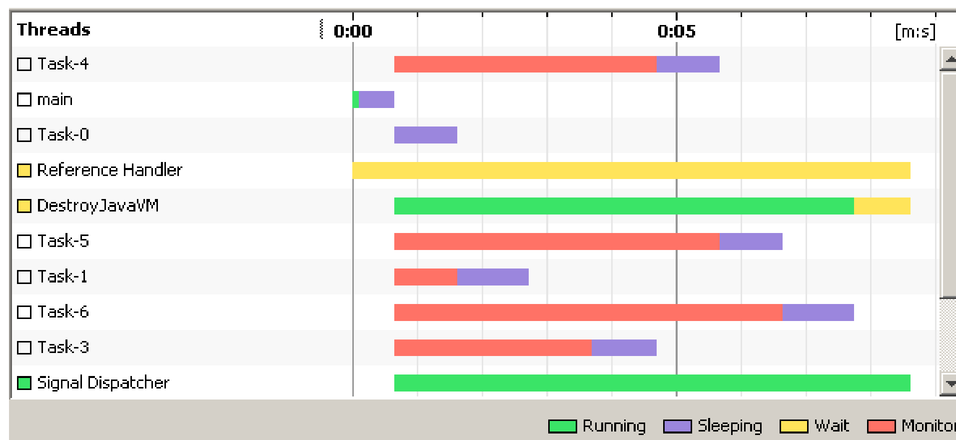
- *not-existing* – vlákno v zadanom čase ešte neexistovalo, alebo už bolo ukončené.
- *running* - vlákno bez prestávky vykonávalo sekvenciu príkazov.
- *sleeping* - vlákno nevykonáva žiadnu činnosť, až kým neuplynie čas zadaný v metóde *Thread.sleep(time).*
- *(waiting, monitor, time)* – vlákno čaká nad objektom *monitor* na zaslanie signálu z iného vlákna. Od zavolania metódy *monitor.wait()* už uplynula doba *time.*
- *(blocked, location, time)* – vlákno čaká na povolenie vstupu do bloku kódu, ktorý je označený ako *synchronized. Location* je lokácia synchronizovaného kódu napríklad v tvare <trieda, metóda, začiatočný riadok>. Vlákno už na prístup čaká po dobu *time*.
- *io* – vlákno sa nachádza v niektorej z metód API[7], ktoré pracujú so vstupno-výstupnými zariadeniami

Dve základné vizualizácie funkcie *ThreadState* sú prehľad počtu (obrázok 9) a prehľad stavu (obrázok 10) aktívnych vlákien počas behu programu. Prídavné informácie o monitoroch využíva napríklad nástroj *JProfiler*, popísaný v kapitole 4.4.



Obrázok 9: Netbeans profiler - prehľad počtu aktívnych vlákien

7 Napríklad natívne metódy tried *FileOutputStream, FileInputStream, SocketOutputStream*, atď ...



Obrázok 10: Netbeans profiler - prehľad stavu aktívnych vlákien

Vlákno, ktoré je v jednom zo stavov *sleeping*, *waiting*, *blocked* alebo *io*, sa nesnaží aktívne uchádzať o procesorový čas. Niektoré profilovacie nástroje zobrazujú modifikovanú verziu *CCT* stromu, v ktorom sa do dĺžky trvania metódy nezaratúvajú časy, ktoré trávila v nečinnom stave. Dĺžky trvania metód potom nie sú jednoduchým rozdielom medzi časom volania a časom návratu. Namiesto toho sa vyratúvajú ako syntetizované atribúty od listov smerom nahor ako súčet dĺžky trvania všetkých podmetód. Volaniam *Object.wait()* a *Thread.sleep()* sa priraďuje nulová dĺžka trvania. Takáto modifikácia má význam v prípadoch, keď sa vo vlákne opakovane vykonáva ten istý algoritmus viackrát, ale medzi jednotlivými spusteniami práce algoritmu sú miesta, kedy vlákno čaká na nové vstupy. Ak by sa do dĺžky trvania práce algoritmu zarátali aj nečinné momenty, skreslilo by to výsledok o skutočnej efektívnosti algoritmu.

Pokročilé profilovacie nástroje idú ešte o jeden krok ďalej. Niektoré operačné systémy poskytujú funkcie na získavanie procesorového času tak, ako je prideľovaný jednotlivým vláknam a nie JVM ako celku. Z nameraných hodnôt sa potom vyratúva približný procesorový čas priradený metódam. Tento čas sa vo všeobecnosti líši od času, získaného z údajov systémových hodín. Vlákno, ktoré je v bežiacom stave na úrovni JVM ešte nemusí byť bežiacim vláknom na úrovni operačného systému.

2.5 Automatická správa pamäte

Špecifikácia JVM definuje tzv. haldu (*heap*), ktorá slúži na uchovávanie objektov vytvorených za behu aplikácie. Na vytvorenie nového objektu slúži kľúčové slovo *new.* Vtedy sa na halde vymedzí nové miesto a zapíšu doň hodnoty dát, ktoré objekt reprezentuje. V syntaxe jazyka Java nie je žiaden mechanizmus na explicitne zrušenie objektu a uvoľnenie miesta na halde. Podľa JVM špecifikácie musia byť objekty uvoľňované automatickým procesom v rámci JVM nazývaným *garbage collection (GC)*. Detaily zvolenej dátovej štruktúry pre haldu ani použitého algoritmu pre

GC proces nie sú špecifikáciou predpísané. Konkrétna implementácia musí spĺňať základnú podmienku správnosti GC procesu a to odstraňovať z haldy jedine tie objekty, ku ktorým už aplikácia nemôže mať od danej chvíle prístup. Možnosť prístupu k objektu neexistuje, ak naň už neexistuje žiadna priama alebo nepriama referencia z tzv. koreňov haldy (*GC roots*).

V súvislosti s vytváraním objektov a ich umiestňovaním na haldu môžu pri behu aplikácie vznikať neželané situácie. Prvou z nich je situácia, kedy na halde vzniká periodicky veľké množstvo krátkodobo žijúcich objektov a aplikácia má na krátke momenty vysoké pamäťové nároky. Druhou situáciou je tzv. *memory leak*, situácia, kedy na halde dlhodobo pribúdajú nové objekty bez toho, aby ich *GC* algoritmus mohol uvoľniť. Obe situácie môžu spôsobiť zníženie výkonu aplikácie a v konečnom dôsledku aj predčasné nekorektné ukončenie aplikácie z dôvodu maximálneho zaplnenia dostupnej pamäte.

2.5.1 Graf haldy

Z abstraktného pohľadu možno haldu v jednom momente behu programu reprezentovať orientovaným grafom. Vrcholy grafu sú objekty, orientovaná hrana vychádza z objektu A do objektu B, ak nejaký člen objektu A (*field*) obsahuje referenciu na objekt B. Objekty majú vstupné aj výstupné hrany. Ďalšími vrcholmi sú už spomínané korene haldy, ktoré majú len výstupné hrany. Patria medzi ne statický členovia tried (*static fields*), lokálne premenné všetkých aktuálne vykonávaných metód vo všetkých vláknach (metódy, ktoré sa nachádzajú v aktuálnych kontextov volania metód) a špeciálne objekty JVM.

Základným profilom pre pamäťové profilovanie je funkcia *Heap(t)*, ktorá vracia graf reprezentujúci stav haldy JVM v zadanom čase *t*. Nie je potrebné aby vrcholy grafu reprezentujúce objekty obsahovali kompletné informácie o dátach, ktoré objekt drží. Vo vrcholoch nemusí byť je prítomná informácia o hodnotách primitívnych dátových typov. V prípade polí primitívnych typov postačuje informácia o ich dĺžke. Graf haldy ale aj napriek tomu ostáva veľmi rozsiahlou dátovou štruktúrou na manuálnu analýzu aj samotnú grafickú prezentáciu. Nástroje HAT, YourKit profiler a JProfiler ponúkajú rôzne pomocné funkcie na analýza grafu a sú predstavené vo štvrtej kapitole.

2.5.2 Tabuľka inštancií tried

Vhodnými algoritmami je možno previesť profil grafu haldy do inej, jednoduchšej lineárnej formy. Objekty v grafe možno rozdeliť podľa príslušnosti k triedam a pre každú triedu zadefinovať funkciu *HeapByClass(class)*. Funkcia vracia pre zadanú triedu dve hodnoty. Prvou z nich je počet

inštancií objektov danej triedy v halde, druhou veľkosť dát, ktoré všetky objekty triedy v halde zaberajú. Táto funkcia však málokedy dáva jednoznačnú odpoveď na otázku, kde jednotlivé inštancie objektov vznikali. Vo väčšine prípadov sú na prvých troch miestach tabuľky funkcie vždy objekty typu *int[]*, *char[]* a *java.lang.String*. Z tabuľky však ťažko vyčítať, odkiaľ sa na tieto objekty odkazuje. Oveľa lepší obraz o situácii sa dosiahne, ak sa veľkosť objektu nebude počítať len na základe veľkostí dátových štruktúr, ktoré drží, ale prirátajú sa k nej aj veľkosti všetkých objektov, na ktoré sledovaný objekt odkazuje. Obrázok 11 znázorňuje objekt *Client*, ktorý si v 48 bajtoch drží referencie na tri objekty, jeden z objektov je však rozsiahla kolekcia objektov. Možno povedať, že existencia objektu *Client* je príčinou zaplnenia haldy JVM v rozsahu vyše jedného megabajtu.

Name	Objects		Shallow Size		Retained Size	
<Objects by classes>	37 667	100 %	2 053 616	100 %	2 053 616	100 %
java	33 819	90 %	1 230 072	60 %	2 044 520	100 %
sk.bgs.aaalogs.protocol.client	8	0 %	200	0 %	1 322 344	64 %
Client	1	0%	48	0%	1 322 192	64%
confirmedMessageIds java.util.Set					1 274 608	
byteBuffer java.nio.ByteBuffer					32 832	
lastConfirmationRequestId java.lang.String					72	
ReturnState	5	0 %	80	0 %	80	0 %
ClientConfiguration	1	0 %	40	0 %	40	0 %
ReturnState[]	1	0 %	32	0 %	32	0 %
char[]	3 614	10 %	686 376	33 %	686 376	33 %

Obrázok 11: Yourkit profiler - zoznam tried a ich veľkostí v pamäti

2.5.3 Vytváranie objektov v metódach

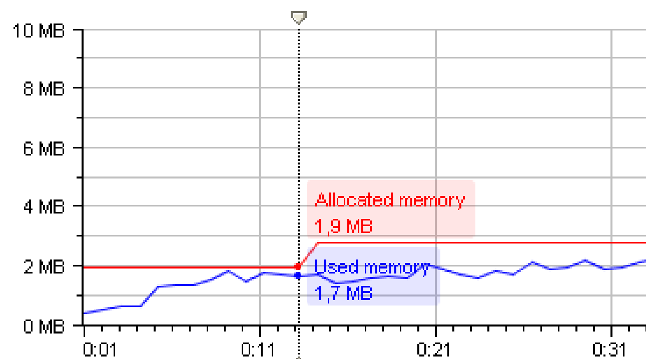
Ak sa už podarí identifikovať na halde množinu objektov, ktoré spôsobujú neželané zaplnenie pamäte, je potrebné identifikovať miesto v kóde, v ktorom objekty vznikali. Ide o prípady, kedy v kóde existuje viacero miest, kde sa množina objektov vytvára. Odpoveď na takéto otázky poskytuje ďalšie rozšírenie grafu haldy. Každý objekt v grafe bude mať jeden atribút naviac. Atribútom je vlákno a aktuálny kontext volaní metód pri vzniku objektu. Z takto rozšíreného grafu haldy možno odvodiť verziu *CCT* stromu, ktorá bude pre každý kontext obsahovať informácie o počte a veľkosti objektov, ktoré v ňom vznikli (obrázok 12).

Name	Objects		Size	
All Threads	10 011	100 %	1 337 000	100 %
Intersection.main(String[])	4 204	42 %	244 168	18 %
Intersection.init()	235	2 %	34 960	3 %
Intersection$DemoControls.<init>(Intersection$Demo)	233	2 %	34 608	3 %
java.awt.Container.add(String, Component)	1	0 %	24	0 %
Intersection.start()	9	0 %	392	0 %
java.awt.Component.setSize(Dimension)	5	0 %	136	0 %
java.awt.Container.add(Component)	1	0 %	16	0 %
java.awt.Frame.<init>(String)	3 413	34 %	116 504	9 %
java.awt.Window.pack()	520	5 %	85 264	6 %
java.awt.Window.show()	14	0 %	1 448	0 %
java.awt.EventDispatchThread.run()	780	8 %	940 664	70 %
java.lang.Thread.<init>(ThreadGroup, String)	3	0 %	72	0 %
java.lang.Thread.run()	5 024	50 %	152 096	11 %
<Objects without allocation information>	10 765		864 264	

Obrázok 12: YourKit profiler – vytváranie objektov podľa kontextov výpočtu

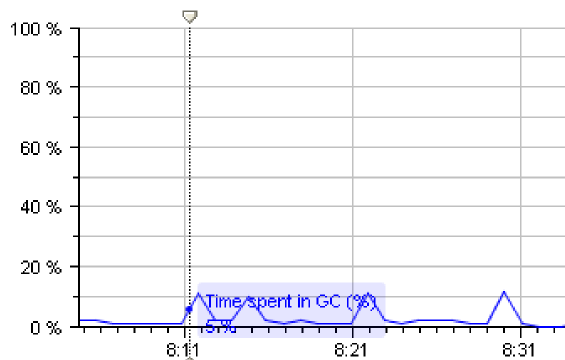
2.5.4 Prehľad stavu haldy a *GC* procesu v čase

O tom, kedy sa halda nachádza v nežiadúcom stave a kedy je najvhodnejší moment na získanie grafu haldy informuje dôležitá dvojica profilov. Prvou z nich je funkcia *HeapSize(t)*, ktorá vracia veľkosť všetkých alokovaných objektov na halde v danom memente (obrázok 13). Profilovacie nástroje môžu brať do úvahy aj detaily konkrétnej implementácie JVM a zobrazovať rozšírené metriky konkrétnej dátovej štruktúry pre haldu [15].



Obrázok 13: YourKit profiler - zaplnenie pamäte objektami

Druhým profilom je funkcia *GCTaskCPU(t)*, ktorá má rovnaký definičný obor ako funkcia *ProcessorTime(t)* z kapitoly 2. Funkčnou hodnotou je podiel času, ktorý JVM strávila vykonávaním GC procesu. Tento profil môže naznačiť ako nastaviť parametre alebo výber konkrétneho GC algoritmu, aby nespôsoboval príliš časté alebo príliš dlhé intervaly, kedy je beh samotnej aplikácie na čas zastavený. Aj v tomto prípade môžu byť funkčné hodnoty rozšírené o atribúty, ktoré charakterizujú konkrétnu implementáciu GC algoritmu.



Obrázok 14: YourKit profiler - podiel GC procesu pri behu aplikácie

2.6 Špecializované profily

Množina profilov, ktorá bola uvedená, je akousi bázou profilov, pomocou ktorých by malo byť možné odhaliť väčšinu výkonnostných problémov rôznych aplikácii. Proces analýzy týchto profilov nemusí byť jednoduchý. Rôzne profilovacie nástroje preto poskytujú modifikácie alebo úplne jedinečné typy profilov za účelom zjednodušenia interpretácie nazbieraných výsledkov. Ich ukážky budú uvedené v kapitole 4.

Java je technológia na všeobecné použitie. Možno pomocou nej implementovať sieťovú aplikáciu (FTP server, WEB server, P2P klient), užívateľské rozhranie v podobe hrubého klienta (*Swing* aplikácia) a iné. V závislosti od domény, do ktorej aplikácia patrí nás môžu zaujímať aj niektoré veľmi špecifické aspekty behu aplikácie. Príkladom môže byť počet súčasne otvorených sieťových spojení, množstvo odoslaných a prijatých dát, rýchlosť vykresľovania grafických prvkov.

Nemôže existovať nástroj na všetko. Dostupné profilovacie programy poskytujú získavanie len štandardnej množiny profilov (volania metód, pamäť, vlákna), ktoré boli v kapitole 2 uvedené, prípadne ich modifikácie. Ich výhodou však je, že na riešenie úlohy profilovania využívajú spravidla pokročilé technologické prístupy, majú nízky dopad na výkon spúšťanej aplikácie, a disponujú širokými možnosťami konfigurácie. Na získavanie neštandardných profilov je potrebné implementovať vlastný profilovací nástroj. V nasledujúcej kapitole budú opísané detaily technológií a postupov používaných pri profilovaní. Bude objasnený spôsob, ako štandartné profilovacie nástroje pracujú a zároveň spôsob, ako by bolo možné vytvoriť vlastné, špecializované profilovacie nástroje.

3 Technológie a postupy na získavanie profilovacích údajov

3.1 Zdroje profilovacích informácií

V predchádzajúcej kapitole bolo predstavených niekoľko typov profilov, ktorými sa dá popísať beh aplikácií v JVM a činnosť samotnej JVM. Možno pozorovať, že vytváranie uvedených profilov závisí na dvoch predpokladoch. Prvým predpokladom je možnosť určitej introspekcie do vonkajšieho alebo vnútorného stavu JVM vo zvolenom čase. Predpokladá sa existencia mechanizmov, ktorý čakajú na vonkajšiu požiadavku a vracajú informáciu v čase jej prijatia. Budú označené ako pasívne mechanizmy (pasívne zdroje), keďže sú z JVM, prípadne inej lokácie „vyťahované“ len na požiadanie. Druhým predpokladom je existencia mechanizmov, ktoré po svojej aktivácii generujú informácie o rôznych udalostiach v čase ich vzniku. Tieto budú označené ako aktívne mechanizmy (aktívne zdroje), pretože generujú informácie bez vonkajších podnetov a zasielajú ich k zaregistrovaným príjemcom.

Na vytváranie niektorých typov profilov sa viac hodia pasívne, na iné zasa aktívne zdroje. O voľbe rozhoduje náročnosť implementácie zdroja, existencia jasne definovaných udalostí a jeho nízky dopad na výkon systému, ktorý väčšinou priamo závisí od množstva informácií, ktoré zdroj generuje. Pasívne mechanizmy poskytujú presne taký počet informácií, ktoré sú od neho vyžiadané, a preto možno jeho činnosť presne regulovať. Aktívne mechanizmy môžu zase na druhej strane začať produkovať nepredpokladane veľké množstvo dát, zahltiť systémové prostriedky a negatívne ovplyvniť beh pozorovanej aplikácie.

Profily JVM ako systémového procesu uvedené v kapitole 2.2 sú zástupcom profilov, kde sa výlučne používajú pasívne zdroje. Príkladom je profil zaťaženia procesorového času JVM procesom. Rozdeľovanie procesorového času medzi procesy má typicky na starosti operačný systém (*multitasking*). Nie je jednoduché definovať udalosti pre túto činnosť. Ak by ňou boli dve udalosti priradenia a odobratia procesora procesu, bolo by síce z nazbieraných údajov možné zrekonštruovať hodnoty funkcie *ProcessorTime(t),* spracovať kvantum generovaných dát by však bola výpočtovo náročná operácia. Lepšou stratégiou je preto nechať zaznamenávanie procesorových časov procesov v réžii operačného systému. Aktuálny stav funkcie bude stále dostupný cez definované rozhranie a profilovací nástroj môže vo zvolených intervaloch hodnoty funkcie čítať.

Rovnaké dôvody vedú k používaniu pasívnych zdrojov pri profilovaní pamäťovej haldy. Aj v tomto prípade by bolo možné sledovať zmeny na halde pomocou udalostí, kedy dochádza k vytvoreniu objektu a odstráneniu objektu z haldy a zrekonštruovať tak napríklad tabuľku inštancií tried (kapitola 2.5.2). Ak by boli navyše sledované všetky priradenia referencií objektov do premenných, vytvoriť by bolo možné aj samotný graf haldy [16]. Išlo by však o spracovanie veľmi veľkého množstva informácií a robenie duplicitnej práce, ktorú už JVM vo svojom vnútri robí pri udržiavaní dátovej štruktúry pamäťovej haldy. Výsledné profily by boli navyše veľmi detailné, zachytávali by stav haldy v ľubovoľnom momente výpočtu. Táto miera detailov nemá až taký praktický význam. Stav haldy je zaujímavý len v niektorých momentoch, najmä v momentoch jej vysokého zaplnenia a jej graf stačí pri profilovaní vytvárať len obmedzený počet krát. Vtedy sa ako výhodnejšie ukazuje opakovaný jednorázový prieskum grafu haldy na požiadanie. Výnimkou sú profily vytvárania objektov v metódach (kapitola 2.5.3), kde je nutné odchytávanie udalosti vytvárania objektov, aby bolo možné danú udalosť priradiť ku kontextu volania metód, v ktorom objekty vznikali. Vo všeobecnosti sú výnimkami profily, ktoré k živým objektom priraďujú rozšírené atribúty, ktoré si virtuálny stroj v pamäťových štruktúrach neukladá (okrem kontextu volaní ním môže byť čas vzniku objektu a iné)

Profily behu podľa volania metód a sledovanie stavu vlákien patria medzi tie, ktoré možno vytvárať z obidvoch typov zdrojov. Pasívne mechanizmy sa použijú, ak je postačujúci približný obraz behu programu a vytvárajú sa aproximačné verzie profilov (kapitola 2.3.5). Generovanie udalostí (vstup a výstup z metódy) v aktívnych zdrojoch sa aktivuje v prípade, kedy sa vyžaduje presný obraz v podobne jedného z profilov z kapitol 2.3.1 až 2.3.4.

V ďalších častiach budú predstavené mechanizmy na získavanie profilovacích údajov, ktoré poskytuje posledná verzia špecifikácie Java technológie (Java SE 6 Mustang), v nasledovnom poradí:

- Profilovanie bez modifikácie kódu sledovanej aplikácie. Tento typ profilovania poskytujú interné funkcie a rozšírenia virtuálneho stroja.

- Profilovanie s modifikáciou kódu, proces známy pod pojmom inštrumentácia kódu.

- Profilovanie implementované výlučne pomocou Java API.

- Aspektovo-orientované programovanie.

3.2 Rozšírenia virtuálneho stroja

Ako už bolo uvedené v popise špecifikácie technológie Java, implementácie Java virtuálneho sú prevažne softvérové. Softvérová implementácia výpočtového prostredia má oproti hardvérovej vo všeobecnom prípade nevýhodu v rýchlosti vykonávania kódu. Na druhej strane však softvérový virtuálny stroj umožňuje oveľa väčšiu flexibilitu a rozšíriteľnosť. Špecifikácia JVM s týmto faktom ráta a využíva ho. Okrem primárnej funkcie vykonávania kódu definuje špecifikácia aj sekundárne funkcie.

Bežne sa takéto funkcie využívajú napríklad pri ladení kódu. Ladenie (*debugging*) je príkladom, kedy sa vo vnútri virtuálny stroj prepne do stavu, v ktorom je počas behu programu navyše schopný prijímať príkazy, ktorými je možné beh zastaviť a znovu obnoviť. Na požiadanie vie JVM poskytnúť informácie o aktuálnom stave programu. Typicky sú to metódy, v ktorých sa vlákna práve nachádzajú, príkaz a riadok v zdrojovom kóde, ktorý je práve v metóde vykonávaný a hodnoty lokálnych premenných. Ďalšie vykonávanie programu môže prebiehať po jednotlivých riadkoch, ďalší príkaz sa vykoná len tedy, ak JVM dostane na to príkaz. Iným typom príkazu je tzv. *breakpoint*, vtedy JVM dáva pozor, kedy sa beh programu dostane na definované miesto a v príslušných momentoch na to upozorní. Vidno, že inak priamočiary proces behu programu, bežiaci ako keby v čiernej skrinke, môže byť procesom manažovaným. Práve manažment a možnosť introspekcie vnútra JVM sú sekundárne funkcie, ktoré špecifikácia JVM definuje. Konkrétne je proces ladenia je podporovaný rozhraním JVMDI (*Java Virtual Machine Debug Interface*).

Profilovanie je proces, ktorý rovnako ako ladenie závisí sekundárnych funkciách. Profilovanie je proces pozorovania, preto sú preň na prvý pohľad dôležitejšie viac funkcie introspekcie ako manažmentu. Vo verzii J2SE 1.2 sa prvý krát objavila špecifikácia rozhrania JVMPI (*Java Virtual Machine Profiler Interface*) ([17], [18]). Rozhranie je definované v hlavičkovom súbore *jvmpi.h*, obsahuje zoznam poskytovaných funkcií. Na prístup k týmto funkciám je potrebné implementovať takzvaného agenta v ľubovoľnom programovacom jazyku, ktorý umožňuje volanie funkcií podľa konvencií jazyka C alebo C++. Samozrejme musí použitý jazyk zohľadňovať platformu, pre ktorú je implementácia virtuálneho stroja určená. Tento agent funguje potom ako rozširujúca dynamická knižnica, beží ako súčasť procesu JVM, ak sa JVM spustí s prepínačom -java *-XrunprofilerLibrary :options*, a počas behu má prístup k všetkých zdrojom dát a ovládacím funkciám JVM, ktoré rozhranie JVMPI definuje. Ide o nasledovné dôležité funkcie a notifikácie:

- notifikácie o každom vstupe do metódy a výstupe z metódy
- notifikácie o vytvorení objektu, presune objektu na halde a jeho zániku

- notifikácie o vytváraní regiónov (*arena*) v štruktúre pamäťovej haldy
- notifikácie o začiatku o konci práce GC procesu
- notifikácie spustení a ukončení vlákien, notifikácie o prístupoch vlákien k monitorom
- notifikácie o nahratí novej triedy (*ClassLoadHook*) a uvoľnení triedy s možnosťou modifikovať bajtkód nových tried
- funkcia na získanie všetkých aktuálnych kontextov volaní vo vláknach a stavov vlákien
- funkcie na získanie využitia procesorového času vláknami
- funkcie na získanie zoznamu všetkých objektov na halde a kompletného grafu haldy

Vidno, že rozhranie JVMPI poskytuje v podstate všetky typy aktívnych aj pasívnych zdrojov informácii, ktoré sú potrebné na tvorbu profilov definovaných v kapitole 2. Ako však vývoj pokračoval, prinášala jeho implementácia čoraz viac problémov. Problematické je získavanie grafu haldy či zoznamov objektov. JVMPI vracajú informácie tohto typu ako návratovú hodnotu jedného volania funkcie. Návratovou hodnotou je pole bajtov, ktoré reprezentuje haldu. Pri zavolaní funkcie preto vzniká v JVM procese štruktúra rovnakých rozmerov, ako samotná halda objektov a a agent ju musí spracovať ako celok. JVMPI tiež obsahuje nedostatky dizajnového charakteru. Notifikácie možno zapínať a vypínať len na globálnej úrovni. Nemožno nastaviť žiadnym spôsobom podmnožinu metód a objektov, o ktorých majú byť notifikácie generované. Zapnutie notifikácii pritom výrazne spomaľuje beh programu.

Prvé implementácie virtuálnych strojov pracovali výlučne na princípe interpretácie bajtkódu. Implementácia rozhrania JVMPI bola v prostredí interpretovaných programov naozaj prirodzená. Interpretér je softvérový a teda ľahko rozšíriteľný komponent [19]. Interpreter najprv parsuje bajtkód, výsledkom je postupnosť inštrukcií a ich argumentov. Vyhodnotenie jednej inštrukcie bajtkódu je viditeľne oddeliteľná funkcia v implementácii interpretera. Ak sa rozhraním JVMPI aktivovalo generovanie notifikácií o vstupoch a výstupoch z metód a vytváraní objektov, zdrojom notifikácii bola vyhodnocovacia funkcia, ktorá zaslala notifikáciu spustenému agentovi. Rovnako boli prvé GC algoritmy pomerne jednoduché a vedeli ľahko generovať notifikácie o pohyboch a rušení objektov. S príchodom moderných virtuálnych strojov sa začali veci komplikovať. Virtuálne stroje začali bajtkód kompilovať do natívneho kódu pre platformu, pre ktorú boli určené. Táto technika je známa pod pojmom *Just-In-Time compiling (JIT)*, pretože kompiluje metódy tried až

pri ich prvom zavolaní. Firma SUN je známa svojou HotSpot JIT technológiou [20]. Okrem iných vlastností, ktorými HotSpot technológia disponuje, je vysoká optimalizácia kódu, špeciálne určená pre kompilovanie objektovo-orientovaných jazykov. V týchto virtuálnych strojoch je veľmi ťažké implementovať funkcionalitu JVMPI, najmä generovanie rôznych notifikácií. Stroj sa pravidelne nachádza v momentoch, kedy v ňom výlučné beží beží kus skompilovaného kódu. Tento kód často zodpovedá mnohým volaniam metód a vytváraním množstva objektov, keďže jedna z najefektívnejších metód optimalizácie je vkladanie kódu metód navzájom do seba (*method inlining*). Aktivácia generovania notifikácií zabraňuje mnohým optimalizáciám. Rovnako priniesli moderné virtuálne stroje nové GC algoritmy. Aj ich činnosť sa nezlučuje s typom notifikácii, ktoré rozhranie JVMPI vyžaduje. Preto bolo rozhranie JVMPI počas celej doby svojej existencie označované ako experimentálne a určené na nahradenie novým rozhraním v budúcnosti.

Ako reakcia na vzniknuté problémy vznikla požiadavka s označením JSR-163 na špecifikáciu novej profilovacej architektúry, ktorá by okrem iného nahradila rozhranie JVMPI a rešpektovala nové trendy vo vývoji JVM technológií [21]. JSR-163 okrem iného špecifikuje nové univerzálne rozhranie JVMTI určené pre tvorbu profilovacích a ladiacich agentov [22]. JVMTI poskytuje väčšie množstvo funkcií, s lepšie navrhnutým dizajnom. Rozhranie rešpektuje dodávateľov virtuálnych strojov, ktorý nechcú implementovať plnú množinu jeho funkcií a obsahuje možnosť nastaviť zoznam dostupných funkcií. Profilovací nástroj si vie preto zistiť, či mu konkrétna implementácia JVM poskytuje potrebné funkcie. Špecifikácia JSR-163 bola zaradená do J2SE od verzie 5.0. Rozhrania JVMDI a JVMPI prestali byť súčasťou špecifikácie od verzie 6.

```
jvmtiFrameInfo frames[5];
jint count;
jvmtiError err;

err = (*jvmti)->GetStackTrace(jvmti, aThread, 0, 5,
                               &frames, &count);
if (err == JVMTI_ERROR_NONE && count >= 1) {
   char *methodName;
   err = (*jvmti)->GetMethodName(jvmti, frames[0].method,
                       &methodName, NULL);
   if (err == JVMTI_ERROR_NONE) {
      printf("Executing method: %s", methodName);
   }
}
```

Obrázok 15: JVMTI - Funkcia vracajúca aktuálne vykonávanú metódu vlákna

Rozhranie JVMTI umožňuje napríklad získavanie grafu haldy alebo množiny objektov na halde traverzovaním. Dá sa tak preskúmať len zvolená množina objektov dosiahnuteľných zo zadaných koreňov, alebo len inštancie objektov podľa zadanej triedy. Najdôležitejšou zmenou je, že neposkytuje notifikácie o vstupoch a výstupoch z metód a o vytváraní a presune objektov tak, ako

ich poskytuje JVMPI. Namiesto toho je rozšírená možnosť výmeny bajtkódu počas behu programu, ktorú už v obmedzenej miere poskytovalo JVMDI rozhranie (*class redefinition*) a JVMPI rozhranie (*ClassLoadHook*). JVMTI umožňuje v ľubovoľnom momente behu aplikácie vymeniť bajtkód množiny tried za nový. Nový bajtkód sa použije len pri nových volaniach metód triedy. Ak práve beží kód metódy nejakej z vymieňaných tried, beží až do konca na starej verzii. Nový bajtkód musí byť kompatibilný so starou verziou, nesmú sa meniť názvy ani počty metód a členov triedy. Túto funkcionalitu musí samozrejme podporovať aj JIT kompilátor virtuálneho stroja a prekompilovať nanovo zmenené časti bajtkódu. Je dôležité uvedomiť si, že inštrukcie, ktoré mohli byť do bajtkódu pridané budú pre JIT kompilátor nerozlíšiteľné od ostatných inštrukcií a aplikujú sa na ne rovnaké optimalizácie. Výsledný efekt je ten, že vykonávanie pridaných bajtkódových inštrukcií spôsobuje menšie spomalenie ako prerušované vykonávanie bajtkódu generovaním notifikácií. Ako súvisí výmena bajtkódu s profilovaním, uvádza nasledujúca časť popisujúca techniku inštrumentácie kódu.

3.3 Inštrumentácia kódu

Inštrumentáciu kódu možno v skratke definovať ako vkladanie pomocných inštrukcií do kódu programu, ktorých úlohou je generovať informácie súvisiace s behom programu. Ručné vkladanie takýchto dočasných príkazov používa každý programátor počas vývoja na to, aby si potvrdil či vyvrátil svoje predpoklady o stave programu počas jeho behu.

```
public int getDogCount(List<Animals> animals) {
    int dogCount = 0;
    for (Animal animal : animals) {
        if (animal instanceof Dog) {
            dogCount++;
        }
    }
    return dogCount;
```

pôvodný kód

```
public int getDogCount(List<Animals> animals) {
    int dogCount = 0;
    for (Animal animal : animals) {
        System.out.println("animal = " + animal.getClass().getName());
        if (animal instanceof Dog) {
            dogCount++;
        }
    }
    return dogCount;
}
```

rozšírenie kódu o dočasný pomocný príkaz

Typicky sa k tomuto kroku programátor podujme, ak je v programe chyba a statickou analýzou kódu sa ju nepodarilo odhaliť. Opísaná činnosť sa volá ladenie (*debugging)*.

Vidno, že ladenie a profilovanie majú jednú spoločnú vlastnosť. Obidva procesy skúmajú bežiaci program, obidva procesy generujú informácie. Druh informácií je však odlišný. Dáta získané procesom ladenia majú kvalitatívny charakter. Hodnoty a typy dát priamo odrážajú logiku, ktorú program implementuje a vstupno-výstupné podmienky, ktoré má program spĺňať. Miesta, kam sa pomocné príkazy pri ladení vkladajú sú veľmi špecifické a úzko súvisia s aplikačnou logikou a hľadanou chybou. Ladiace inštrukcie skoro vždy pristupujú k premenným objektov alebo volajú metódy tried z vyvíjanej aplikácie. Po nájdení o odstránení chyby sa pomocný príkaz odstráni alebo zakomentuje a v prípade odstránenia je málo pravdepodobné, že sa presne rovnaký príkaz niekedy vloží na presne rovnaké miesto. Rovnaká chyba by sa už nemala znovu objaviť a ak aj áno, kód mohol byť medzitým modifikovaný tak, že je nutné opätovné prehodnotenie situácie a vloženie pomocného príkazu programátorom. Ladenie sa preto radí medzi manuálne činnosti.

Zoberme si teraz príklad sledovania počtu inštancií konkrétnej triedy *Trieda*. Riešenie úlohy inštrumentáciou kódu znamená vložiť do kódu všetkých konštruktorov triedy *Trieda* ako posledný príkaz volanie statickej metódy *Profiler.incrementInstanceCount()*, a prekryť implementáciu metódy *Object.finalize()*. Metóda *finalize()* je volaná v momente[8], kedy *garbage collector* uvoľňuje objekt z pamäte. Prekrytá implementácia bude obsahovať volanie príkazu *Profiler.decrementInstanceCount()*. Metódy triedy *Profiler* implementujeme podľa vlastných potrieb. Môže to byť jednoduché počítadlo, ktoré na požiadanie vypíše aktuálny počet existujúcich inštancií, prípadne si naviac môže pamätať zmeny počítadla v čase a na konci vygenerovať graf. Rovnakým spôsobom by sa dal modifikovať kód viacerých tried naraz a profilovacie metódy volať s parametrom *Trieda.class* na rozlíšenie tried. Generovanie notifikácii o vstupoch a výstupoch z metód sa realizuje napríklad zahrnutím pôvodných príkazov metódy do *try {...} finally {...}* bloku, pred prvý pôvodný príkaz sa vloží volanie *Profiler.methodEnter(class, method)* a do *finally {...}* bloku sa vloží volanie *Profiler.methodExit(class, method)*. Takto sa zabezpečí odchytenie ukončenia metódy pre všetky *return* inštrukcie metódy.

Vidno, že profilovanie je proces zberu kvantitatívnych údajov, spravidla nezávislý od logiky sledovaného programu. Profilovacie inštrukcie ignorujú stav tzv. „business objektov“. Miesta určené na inštrumentáciu pri zbere profilovacích dát sú podľa nejakého presného predpisu roztrúsené po celom kóde a zohľadňujú tak niektorý všeobecný aspekt programovania a nie konkrétnej vyvíjanej aplikácie. Pod všeobecným aspektom programovania sa myslí to, čo majú programe vyvíjané v tomu istom jazyku spoločné, napríklad volania metód, vyhadzovanie výnimiek, vytváranie objektov, zmena hodnôt členov tried a iné. Pod konkrétnym aspektom

8 Kedy tento moment nastane, však nie je dopredu známe. Každá implementácia JVM môže používať rôzne algoritmy na správu pamäte. Programátor sa preto nemôže spoliehať, že metóda *finalize()* bude pri dvoch behoch programu vždy spúšťaná v tých istých časoch alebo v tom istom poradí pre rôzne objekty.

vyvíjanej aplikácie sa myslí napríklad úspešné pripojenie k databáze, prijatie správy o finančnej transakcii a podobne.

Bolo by neúnosné manuálne modifikovať kód všetkých tried, ktoré nás zaujímajú. Na rozdiel od ladenia nie je profilovanie jednorázová záležitosť. Potreba profilovania sa môže opakovať, množina tried, na ktoré profilovanie zameriava pozornosť sa môže meniť, mení sa aj potreba pre rôzne typy profilov. Kompilátor jazyka Java neumožňuje žiadne predspracovanie zdrojového kódu pomocou direktív *define, ifdef* a iných v štýle jazyka C, ktoré by jednoduchou konfiguráciou zvolených konštánt určili, ktoré z profilovacích príkazov majú byť prítomné v skompilovanom programe. Existencia presného predpisu naznačuje, že inštrumentácia môže byť automatickým procesom. Nasledujúce dve podkapitoly popisujú automatický proces inštrumentácie na úrovni zdrojového kódu a na úrovni bajtkódu.

3.3.1 Inštrumentácia zdrojového kódu

Ak chceme meniť štruktúru programu na úrovni zdrojového kódu, musíme ju najprv zo zdrojových súborov získať, upraviť a posunúť ďalej vlastnému procesu kompilácie. V dnešnej dobe je dostupných viacero parserov, ktoré vedia parsovať zdrojové kódy písané v Jave. Dva najpopulárnejšie nástroje sú *Antlr* [23] a *JavaCC* [24]. Obidva nástroje generujú parsery z LL(k) gramatík a ponúkajú funkcie na vytváranie syntaktických stromov a manipuláciu s nimi. Samozrejmosťou sú definičné súbory s gramatikami jazyka Java.

Hlavnou výhodou tohto prístupu je, že si nemusíme osvojovať žiadnu novú technológiu, na ktorú sme doteraz neboli zvyknutí. Výnimkou je iba API parsovacích nástrojov, konkrétne podmnožina API, určená na prácu so syntaktickým stromom. Akonáhle je syntaktická štruktúra k dispozícii, je vkladajú do nej profilovacie príkazy zapísané vo textovom formáte príkazov jazyku Java, tak ako by sa to robilo pre manuálnej inštrumentácii. Úloha teda pozostáva z čítania zdrojových súborov, ich prevodom na syntaktické stromy, úpravy štruktúr podľa zadaných parametrov a celý proces sa spravidla skončí vytvorením modifikovanej kópie zdrojových súborov. Vidno, že inštrumentačný nástroj je špecializovaný kompilátor. Nástroj *Instr* [25] je akousi nadstavbou parsera, ktorý navyše ponúka funkcie, ktoré boli práve spomenuté. Funkcie na čítanie z veľkého množstva súborov, preddefinované konštanty na identifikáciu typov uzlov v syntaktickom strome, traverzovacie funkcie. V dodávanom balíku nástroja *Instr* sú aj ukážkové programy, jeden z nich napríklad demonštruje, ako modifikovať zdrojové súbory tak, že po ich spustení program na výstup vypisuje riadok zo zdrojového kódu, ktorý sa práve vykonáva (obrázky 16 a 17).

```
public class Loop {
                public static void main(String args[])
                {
                        int n = 1;
                        for (int i = 1; i <= 10; i++)
                                n = n * i;
                        System.out.println(n);
                }
        }
```

Obrázok 16: Instr - Program určený na inštrumentáciu

```
        [loop.java 2]   public static void main(String args[])
        [loop.java 4]           int n = 1;
        [loop.java 5]           for (int i = 1; i <= 10; i++)
        [loop.java 6]                   n = n * i;
        [loop.java 6]                   n = n * i;
        [loop.java 6]                   n = n * i;
        [loop.java 6]                   n = n * i;
        [loop.java 6]                   n = n * i;
        [loop.java 6]                   n = n * i;
        [loop.java 6]                   n = n * i;
        [loop.java 6]                   n = n * i;
        [loop.java 6]                   n = n * i;
        [loop.java 6]                   n = n * i;
        [loop.java 7]           System.out.println(n);
        3628800
```

Obrázok 17: Instr - Spustenie inštrumentovaného programu

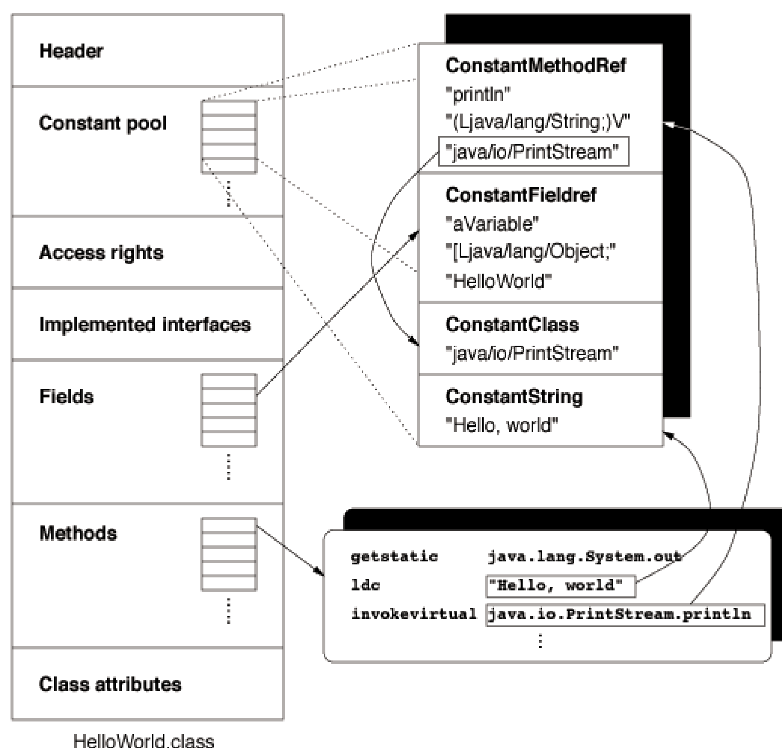
Faktom je, že žiaden profilovací nástroj pre Javu v súčastnosti nevyužíva tento prístup. Dôvodov je niekoľko. Hlavnou nevýhodou je, že zdrojový kód nemožno inštrumentovať, pokiaľ nie je k dispozícii. Takáto situácia je bežnejšia, ako by sa mohlo zdať. Množstvo programových knižníc je dodávaných len v skompilovanej forme v *class* súboroch. Samotné štandartné Java API je archív *class* súborov v súbore *rt.jar.* Vo verzii 1.5 (*Tiger*) od firmy SUN obsahuje 12856 tried. V mnohých prípadoch nemusia byť knižnice, ktoré sa využívajú v programoch implementované efektívne. Neplatí to dokonca ani pre implementácie štandardného Java API. Neefektívnosť knižníc môže byť spôsobená aj nevhodným použitím. Vzniká tu potreba profilovania kódu, ktorý sme doteraz brali ako čiernu skrinku a nezamýšľali sa nad jeho vnútornými vlastnosťami. V kapitole 6.2 je sa uvádza príklad, kedy profilovací nástroj odhalil pamäťový problém pri používaní triedy štandardného API. Zdrojový kód môže chýbať aj v prípade vlastných produktov, ide skôr však o výnimočné situácie.

Ďalšou komplikáciou je fakt, že potom, čo je zdrojový kód inštrumentovaný, výsledné súbory musia prejsť celým procesom kompilácie. V prípade veľkých projektov môže trvať celý *build* rádovo desiatky sekúnd až minúty. Projekt sa síce dá rozdeliť na moduly, ktorých kompilácia by mohla bežať nezávisle od seba, a potom by sa kompilovali len tie moduly, v ktorých boli inštrumentáciou pozmenili zdrojové kódy. Treba opäť spomenúť, že profilovanie je proces, ktorý je spúšťaný niekoľko krát, aby sa postupne zúžila oblasť, kde sa hľadaný problém nachádza. Všetky opísané postupy predstavujú relatívne pomalý proces, ktorý vývojový tím môže vnímať ako

spomalenie a extra prácu pri manažovaní zdrojových kódov a nevyužiť tak naplno výhody profilovania.

3.3.2 Inštrumentácia bajtkódu

Keďže sa inštrumentácia zdrojových súborov ukazuje ako nevhodný prístup, ďalšou možnosťou je modifikovať program potom, čo bol skompilovaný do *class* súborov. Každý *class* súbor obsahuje definíciu práve jednej triedy alebo rozhrania (*interface*). *Class* súbor sú binárne dáta. Prvé štyri bajty hlavičky sú vždy rovnaké (*0xCAFEBABE*), hlavička ďalej špecifikuje verziu formátu. Po hlavičke nasleduje zoznam konštánt (*constant pool*), textová časť, v ktorej sú uložené názvy všetkých tried, metód, premenných a reťazové konštanty, ktoré sa v triede používajú. Každá z nich má pridelený svoj číselný identifikátor. Nasledujú prístupové práva (*access rights*), ktoré sú reprezentované bitovou maskou, ďalej zoznam implementovaných rozhraní, zoznam členov triedy samotné inštrukcie všetkých metód a doplňujúce atribúty ako meno pôvodného zdrojového súboru. Bajtkód dostal svoje meno podľa toho, že každá inštrukcia v metódach je reprezentovaná jedným bajtom.



Obrázok 18: Štruktúra class súboru

Na prácu s bajtkódom existuje niekoľko voľne dostupných knižníc, väčšina z nich je implementovaná v jazyku Java. Nástroje ako BCEL, ASM, JMangler a jclasslib poskytujú funkcie

na transformáciu konštrukciu tried na úrovni bajtkódu. *Class* súbor reprezentujú objektovou štruktúrou, objekty predstavujú reprezentácie jednotlivých stavebných prvkov triedy. Nad štruktúrou poskytujú funkcie na jej modifikáciu. Iný prístup modifikácie bajtkódu poskytuje nástroj Javassist. Nevyžaduje znalosť bajtkódu, vkladané inštrukcie sa zadávajú ako príkazy s Java syntaxou. Javassist obsahuje jednoduchý interný kompilátor Java príkazov do bajtkódu .

Pomocou opísaných nástrojov by bolo možné použiť postup podobný tomu, ako bol opísaný pre inštrumentáciu zdrojového kódu. Opäť sa postupne načítali všetky *class* súbory, ktoré aplikácia používa, vrátane archívov (súbory s príponou *jar*), ktoré treba dekomprimovať. Bajtová informácia každej cieľovej triedy sa zašle ako vstup do niektorého zo spomenutých nástrojov a výsledkom by bola modifikovaná kópia všetkých pôvodných tried, ktoré sa podieľajú na behu aplikácie. Tento postup sa nazýva statická inštrumentácia bajtkódu.

Tento postup má svoje výhody a nevýhody.. Výhodou je, že je možné inštrumentovať triedy, ku ktorým nie je dostupný zdrojový kód. Celý proces inštrumentácie by mal byť teoreticky rýchlejší, keďže relatívne náročný proces parsovania textu je nahradený čítaním presne definovaných štruktúr bajtkódu. Na druhej strane tu je fakt, že je potrebné sa naučiť novú programovaciu techniku, ktorú predstavuje bajtkód. Spomenuté nástroje na modifikáciu bajtkódu však prácu v mnohom uľahčujú a v prípade nástroja Javassist túto nevýhodu dokonca úplne eliminujú. Ostáva teda posledný problém, a to manažovanie množstva *class* súborov a ich inštrumentovaných kópií, podobne ako vznikol pri inštrumentáciu zdrojových súborov.

Práve pre tento účel sa hlavnou filozofiou profilovacej architektúry JSR-163 stala podpora inštrumentácie bajtkódu za behu. Všetky modifikácie bajtkódu sa realizujú v pamäti a odpadajú tak problémy s manažovaním kópií súborov. Inštrumentácia bajtkódu za behu sa nazýva aj dynamická inštrumentácia. Zároveň rieši dynamická inštrumentácia výkonnostné a algoritmické problémy súvisiace s problémom vkladania rôznych profilovacích volaní. V prípade, že profilovací agent dostane počas behu programu príkaz začať sledovať volania metód, musí sa upraviť kód všetkých tried, ktoré budú od danej chvíle zavolané. Naivné riešenie by spustilo inštrumentáciu metód všetkých nahraných tried, čo predstavuje výpočtovo náročný proces a viditeľné zastavenie behu programu. Optimalizovať počet inštrumentovaných metód a tried by sa dal realizovať statickou analýzou kódu. Optimálnymi algoritmami statickej inštrumentácie sa zaoberal Larus [26], v prítomnosti polymorfizmu však nemusí optimálna statická inštrumentácia znížiť počet metód v dostatočnej miere. Java navyše umožňuje volanie metód mechanizmom jazykovej reflexie (*java.lang.reflect*). V prítomnosti jazykovej reflexie je optimalizácia inštrumentácie statickou analýzou neriešiteľným problémom. Dmitriev [27] popisuje ako využiť dynamickú inštrumentáciu

na optimalizáciu počtu modifikovaných metód (*dynamic call graph revelation*). Podstata spočíva v tom, že z kódu jednej metódy sa statickou analýzou zistia všetky možné nasledovné priame volania (rešpektujúc polymorfizmus) a inštrumentujú sa metódy tried len na jeden krok dopredu. Volania pomocou reflexie sa odhaľujú tiež dynamicky inštrumentáciou metódy *java.lang.reflect.Method.invoke()* a odchytením hodnoty jej parametrov. Samotný priebeh programu teda určuje, ktoré metódy sa budú modifikovať a ktoré nie. Profilovací agent si drží zoznam modifikovaných tried a ich pôvodných verzií a po prijatí príkazu na zastavenie profilovania vykoná hromadnú inštrumentáciu tried do pôvodného stavu. JIT kompilátor po chvíli zmeny v kóde a prekompiluje a aplikácia môže ďalej bežať plnou rýchlosťou.

3.4 Podpora profilovania na úrovni Java API

3.4.1 Základná výpočtová reflexia

Definícia výpočtovej reflexie je schopnosť programu získať informáciu o svojej vlastnej štruktúre a stave výpočtu (*introspection*) a schopnosť meniť svoju štruktúru a správanie (*intercession*) [28]. Java patrí k jazykom, ktoré možnosť výpočtovej reflexie v istom rozsahu umožňujú. Práve reflexia je hlavným mechanizmom pre získavanie profilovacích údajov na úrovni API programovacieho jazyka. Samotný stav a štruktúru programu zachytávajú, vytvárajú a modifikujú obyčajné triedy a metódy štandardného API, ku ktorým má každý program prístup. V mnohých prípadoch sú tieto metódy funkčne ekvivalentné s funkciami, ktoré v súčastnosti poskytuje rozhranie JVMTI.

Na získanie stavu výpočtu slúžia metódy triedy *java.lang.Thread*. Statická metóda *getAllStackTraces*() poskytuje zoznam inštancií všetkých objektov triedy *Thread* a ku každému z nich aktuálny kontext volania. Objekty reprezentujú aktívne vlákna v momente zavolania metódy *getAllStackTraces*(). Objekt triedy *Thread* má metódu *getState()*, ktorá vracia jeden zo stavov, v ktorom sa môže vlákno nachádzať. Uvedené API možno použiť na implementáciu veľmi jednoduchého profilovacieho nástroja, ktorý by osobitnom vlákne v pravidelných intervaloch zbieral údaje o vláknach a volaných metódach. Obmedzením API je fakt, že dátová štruktúra reprezentujúca kontext volania obsahuje len jednoduché názvy metód a nerozlišuje medzi dvoma metódami ktoré majú s rovnaký názov ale rôzne parametre.

Reflexívne vlastnosti poskytuje ďalej dvojica tried *java.lang.Runtime* a *java.lang.System*. Obsahujú metódy na získanie veľkosti aktuálne dostupnej a zaplnenej pamäte, metódy na získanie počtu dostupných procesorov a možnosť explicitného spustenia GC procesu. Pre profilovanie je zaujímavá dvojica metód *traceInstructions()* a *traceMethodCalls()*. Obidve by mali aktivovať

generovanie informácii o volaní metód a vykonávaní inštrukcií. Špecifikácia funkcií však vôbec nepopisuje formát emitovaných informácií a dokonca dovoľuje implementácii JVM funkcionalitu vôbec neposkytovať. Nerobia to napríklad virtuálne stroje firmy SUN od verzie Java SE 1.4 vyššie.

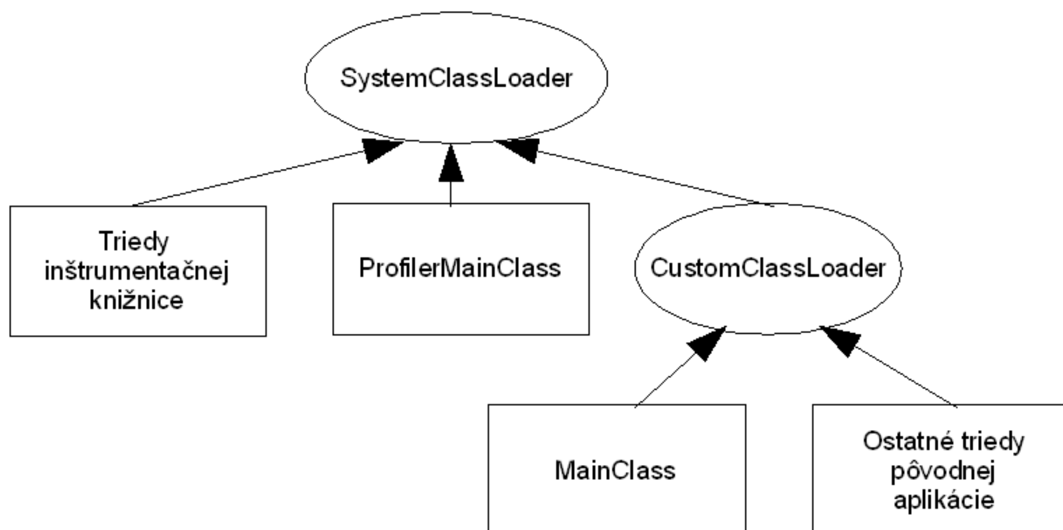
Ďalšou formou výpočtovej reflexie je spôsob, akým JVM nahráva do prostredia spúšťaného programu definície tried. Zopakujeme, že program pre JVM nie je jeden monolitický spustiteľný súbor ale namiesto toho sa skladá z množstva individuálnych tried. Základná predstava o jednej triede je jeden *class* súbor. Nie je to však pravidlo, definícia triedy môže byť uložená na ľubovoľnom inom médiu, odkiaľ sa dá ako postupnosť bajtov, napríklad zo vzdialeného servera. Spôsob, akým JVM tieto bajtové informácie načítava, špecifikuje niekoľko vlastnosti [29]. Dve z nich predstavujú mechanizmus výpočtovej reflexie:

1. Neskoré načítavanie (*lazy loading*) – definícia triedy je načítaná do pamäte a nová trieda je skonštruovaná až v momente, kedy sa o ňu prvýkrát požiada. Implementácia JVM síce môže vopred načítať aj triedy, o ktoré počas behu programu ešte nebolo požiadané, všetky chybové hlásenia súvisiace napríklad s bezpečnosťou alebo typové nezrovnalosti musí pozdržať až do momentu prvého použitia triedy.

2. Rozšíriteľný načítavaní mechanizmus – za definíciu nových tried je zodpovedný takzvaný zavádzač tried (*ClassLoader*,) trieda zo štandardného Java API. Kým vo väčšine prípadov všetky triedy do aplikácie načítava jeden špeciálny, preddefinovaný systémový zavádzač tried, je možné odvodiť si vlastný a požiadavky na definovanie tried smerovať na neho.

Implementáciou vlastného mechanizmu načítavania tried možno realizovať čiastočne dynamickú inštrumentáciu kódu. Ak sa aplikáciu spustí príkazom *java MainClass*, trieda *MainClass* bude načítaná systémovým zavádzačom, a všetky ostatné triedy, o ktoré bude požiadané v rámci triedy *MainClass* budú smerované na zavádzač triedy *MainClass*, teda opäť systémový. Aby sa dosiahlo načítavanie iným zavádzačom, je potrebné spustenie hlavnej metódy triedy *MainClass* obaliť do inej hlavnej triedy, ktorú sa bude nazývať *ProfilerMainClass.* Jej hlavná metóda vytvorí novú inštanciu triedy *CustomClassLoader. CustomClassLoader* je odvodený od objektu *java.lang.Classloader* a možno v ňom prekryť metódu, v ktorej sa načítaný bajtkód zasiela do systémového mechanizmu, v ktorom sa nová trieda registruje. V prekrytej metóde je na úrovni jazyka k dispozícií bajtkód ešte nezadefinovanej triedy a metóda predstavuje miesto, kde je možne vykonať inštrumentáciu. Na tento *CustomClassLoader* smerujeme požiadavku na zavedenie pôvodnej hlavnej triedy aplikácie *MainClass* a pomocou reflexívneho mechanizmu jazyka Java požiadame práve skonštruovanú triedu *MainClass*, aby spustila hlavnú metódu *main*. Tentokrát

budú všetky ostatné triedy použité počas behu aplikácie zavedené zavádzačom *CustomClassLoader.*. Situáciu popisuje obrázok 19.



Obrázok 19: inštrumentácia pomocou upraveného zavádzača tried

Zjednodušene povedané sa dosiahlo to, že inštrumentačný mechanizmus už nemusí chodiť za triedami, ale triedy prídu za ním. Inštrumentácia prebieha v pamäti, takže odpadajú problémy so správou súborov. Navyše sa v súlade s vlastnosťou neskorého načítavania inštrumentujú len tie triedy, ktoré sa pri danom spustení používajú. Nemusí sa robiť žiadna zbytočná práca naviac. Popísaný postup predstavuje síce dynamickú, ale len jednorázovú inštrumentáciu tried. Je možné vykonať ju len pri štarte aplikácie, známa je preto aj pod pojmom *load-time instrumentation*.

3.4.2 Rozšírenia výpočtovej reflexie podľa JSR-163

Špecifikácia profilovacej architektúry JSR-163 okrem natívneho rozhrania JVMTI špecifikuje aj rozšírenie štandardného Java API na podporu profilovania. Novinkou je dvojica balíkov *java.lang.management* a *java.lang.instrument*.

Prvý balík definuje nový štandard na tvorbu mechanizmov určených na monitorovanie a manažovanie vyvíjaných aplikácií. Súčasťou balíka je aj niekoľko hotových tried, ktoré poskytujú funkcie na monitorovanie nahrávania tried (*ClassLoadingMXBean*), kompilácie kódu (*CompilationMXBean*), práce GC procesu (*GarbageCollectorMXBean*), pamäte (*MemoryManagerMXBean*, *MemoryMXBean*, *MemoryPoolMXBean*), platformy (*OperatingSystemMXBean*) a vlastného behu Java aplikácie JVM (*RuntimeMXBean*, *ThreadMXBean*). Funkcie základnej výpočtovej reflexie opísané v predchádzajúcej časti nevznikli

pôvodne za účelom podpory profilovania. JSR-163 tieto funkcie zjednocuje do spoločného balíka a rozširuje ich o ďalšie. Podobne ako pri natívnom rozhraní JVMTI, aj v prípade tohto API sa dodávateľ implementácie virtuálneho stroja môže rozhodnúť tieto funkcie nepodporovať a korektne označiť vybrané funkcie za neimplementované.

Balík *java.lang.instrument* prináša na úroveň jazyka možnosť plnej dynamickej inštrumentácie tried tak, ako ho podporuje JVMTI. Na získanie mechanizmu inštrumentácie je potrebné implementovať agenta (*Java programming language agent*). Agent je program, ktorého vstupným bodom nie je metóda *main(String args[])* alebo metóda *premain(String options, Instrumentation inst)*. Agent sa spúšťa spolu s monitorovanou aplikáciou príkazom *java -javaagent :agentJarPath=options MonitoredApplicationMainClass.* Ak to implementácia virtuálneho stroja podporuje, agent dostane pri štarte JVM v parametri *inst* inštanciu objektu, pomocou ktorého možno inštrumentovať triedy hlavnej aplikácie.

3.5 Aspektové programovanie

Jeden z hlavných spôsobov, ako zvládnuť komplexitu vyvíjaného softvéru je jeho dekompozícia na menšie časti, z ktorých každá má jasne definovanú funkcionalitu a môže byť vyvíjaná nezávisle od ostatných. Syntax objektovo orientovaného jazyka, akým je napríklad Java, podporuje modularizáciu projektu možnosťou tvorby tried a ich zaraďovaním do hierarchie balíkov. Existujú však aj funkcie softvéru, ktoré je ťažko modularizovať klasickým spôsobom, pretože zasahujú do mnohých komponentov. Klasickým uvádzaným príkladom je logovanie. Metódy zápisu do logov sa volajú z rôznych miest programu a spôsobujú, že mnoho rôznych modulov obsahuje logiku, ktorá sa týka tej istej funkcionality v programe. Takéto funkcie sa nazývajú „presahujúce časti“ programu (*cross cutting concerns*) alebo aspekty programu.

Aspektovo-orientované (AO) programovanie [30] vzniklo ako paradigma, ktorá sa snaží riešiť efektívnym spôsobom implementáciu aspektov programu. Zvyčajne je postavené na vlastnom jazyku, ktorý umožňuje definovať množiny bodov v štruktúre alebo výpočte programu. Definovaným bodom priraďuje akcie, ktoré majú byť vykonané pri ich dosiahnutí. Najznámejším AO jazykom určeným pre Javu je AspectJ. Funguje na princípe statickej inštrumentácie kódu v čase kompilácie[9]. Na základe definícií aspektov vyhľadá vo vygenerovanom bajtkóde body aspektov (*pointcut*) a inštrumentuje ich definovanými akciami (*advice*). Na obrázku 20 je definícia aspektu, ktorý počas behu programu odpovedá na otázku „Koľko krát bola vykonaná metóda triedy *Point*, ktorej meno sa začína na „*set*“,má jeden parameter typu *int* a táto metóda bola priamo alebo

9 Určite bude zaujímavé sledovať vývoj AO programovania v spojení s dynamickou inštrumentáciou.

nepriamo zavolaná z metódy *Line.rotate()*? Odpoveď na túto otázku možno považovať za úzko špecializovaný profil programu.

Vo všeobecnosti možno povedať, že funkcie, ktoré predstavujú profilovacie inštrukcie pridávané inštrumentáciou do kódu programu, predstavujú jeden jeho aspekt. Aspektové programovanie možno preto využiť na jednoduchú definíciu tzv. profilovacích aspektov. Syntax jazyka samozrejme nemôže pokryť všetky možné transformácie kódu, tak ako ich možno realizovať inštrumentáciou na najnižšej úrovni bajtkódu. V súčastnosti je však syntax AO jazykov dostatočne bohatá na riešenie implementácie väčšiny profilovacích aspektov.

```
aspect SetsInRotateCounting {
    int setCount = 0;

    before(): call(void Point.set*(int))
              && cflow(call(void Line.rotate(double))) {
        setCount++;
    }
}
```

Obrázok 20: AspectJ - definícia aspektu

Pohľad do histórie ukazuje, že princípy AO programovania pri riešení problému profilovania boli použité pred tým, ako bol pojem vôbec definovaný. Začiatkom 90-tych vznikol ATOM, framework určený na tvorbu profilovacích nástrojov [31]. Nástroj poskytoval funkcie na inštrumentáciu kódu, používateľ len špecifikoval detaily inštrumentácie, a to miesta v štruktúre inštrumentovaných programov a mená funkcií, ktoré mali byť na zadaných miestach volané. Zakladaná filozofia *pointcut-advice* sa zhodovala s dnešnými AO jazykmi.

4 Profilovacie nástroje

4.1 HPROF a JHAT

HPROF [32] je jednoduchý profilovací nástroj dodávaný s implementáciami JVM od firmy SUN. Prvý krát sa objavil vo verzii J2SE 1.2. Bol dodávaný aj so zdrojovým kódom ako ukážka použitia vtedy nového JVMPI rozhrania. V nasledujúcich verziách JVM 1.3 a 1.4 sa objavili už spomínané problémy funkčne implementovať JVMPI rozhranie a HPROF nepracoval vždy spoľahlivo. S príchodom špecifikácie JSR-163 a JVMTI rozhrania vo verzii J2SE 5.0 boli zdrojové kódy kompletne prepísané podľa novej architektúry pre profilovanie a opäť ich možno nájsť ako ukážku v dodávaných implementáciách JVM.

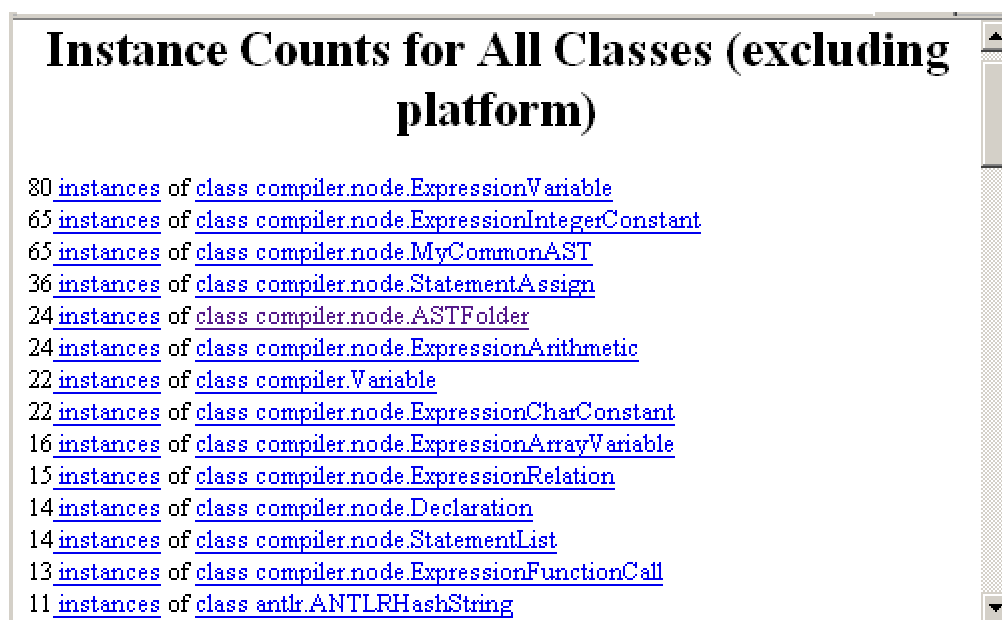
Množina funkcií, ktoré HPROF poskytuje sa od jeho prvého vydania prakticky nezmenila. HPROF je dodávaný ako samostatná dynamická knižnica, spúšťa sa pri štarte JVM pridaním parametra *-agentlib:hprof=options* príkazu *java*. Všetky výstupy sú smerované do zvoleného súboru alebo sieťovú destináciu. Všetky výstupy majú obyčajný textový formát. HPROF vytvára nasledovné výstupy:

- Tabuľky volaní metód získané vzorkovaním alebo s plnými detailami.
- Životné cykly vlákien spolu s informáciou o monitoroch a časoch, ktoré vlákna čakali na prístup k monitorom.
- Graf haldy s voliteľnou možnosťou zaznamenávať kontexty volaní metód, kde boli objekty vytvárané. Graf haldy je jediný profil, ktorý HPROF poskytuje aj v binárnom formáte.

HPROF poskytuje veľmi obmedzenú možnosť konfigurácie. V podstate sa dá jeho činnosť regulovať jedine nastavením intervalu vzorkovania a hĺbky zobrazovaných kontextov volaní. Nie je možné definovať množiny tried, na ktoré má byť obmedzené sledovanie volania metód alebo vytvárania objektov. Implicitne zahŕňa do výsledkov aj profilovanie tried štandardného API a spôsobuje tak veľké spomalenie behu profilovanej aplikácie. Všetky výstupy sú generované z údajov získavaných počas celého behu aplikácie.. Aktualizácia medzivýsledkov je síce možná zaslaním signálu JVM procesu, ide ale o inkrementálny proces. Preto nie je možné zamerať sa len na určený úsek behu aplikácie. Textové výpisy síce poskytujú vysoký detail informácií, je však dosť ťažké sa v nich orientovať. HPROF v podstate plní funkciu nástroja na demonštráciu použitia profilovacích rozhraní definovaných v JVM špecifikácii. Neexistencia používateľského rozhrania,

možnosti konfigurácie a rozšírenej analýzy výsledkov je dôvodom, prečo nemá v praxi veľké uplatnenie.

Možno však povedať, že HPROF zaviedol určitý štandard binárneho formátu grafu haldy. Verzie J2SE 5 a 6 priniesli nové možnosti, ako vie na základe vonkajšej požiadavky v ľubovoľnom momente behu aplikácie alebo pri páde JVM z dôvodu zaplnenia pamäte JVM vygenerovať graf haldy ([33], [34]). Používaným formátom je práve HPROF formát. Spolu s nástrojom HPROF bol od začiatku dodávaný nástroj HAT (JHAT od verzie J2SE 6), ktorý slúžil na analýzu súborov s grafom haldy [35]. HAT funguje na princípe web servera, vie zobrazovať nasledovné reprezentácie haldy:



Obrázok 21: JHAT - počet inštancií podľa tried

- Zoznam tried, ktorých inštancie sa v halde nachádzali, spolu s počtom živých objektov podľa tried (obrázok 21)
- Detaily zvolenej triedy, jej štruktúra a zoznam všetkých jej inštancií.
- Detaily zvolenej inštancie (objektu).
    - Kontext volaní, v ktorom objekt vznikol.
    - Priamo alebo nepriamo dosiahnuteľné objekty z daného objektu (*outgoing references*).
    - Objekty, ktoré priamo odkazujú na zvolený objekt (*incoming references*).

    - Korene haldy, ktoré nepriamo odkazujú na zvolený objekt (*GC root paths*).
- *Object query language* (*OQL*), jazyk s podobnou syntaxou ako jazyk SQL, na výber množiny tried a objektov z haldy pomocou zadaných dotazov.

4.2 YourKit profiler

YourKit profiler [36] je komerčný softvérový produkt vyrábaný rovnomennou ruskou spoločnosťou. V poslednej verzii 5.5. sa nástroj zameriava výlučne na profilovanie CPU a pamäte, okrem základnej telemetrie chýba úplne profilovanie stavu vlákien. Mierou prepracovania existujúcich funkcií sa však radí medzi skutočne profesionálne nástroje určené nie len pre fázu vývoja a testovania, ale aj do produkčnej prevádzky. Architektúra sa klasicky skladá z agenta, ktorý rozširuje JVM proces a grafickej front-end časti, ktorá vzdialenému agentovi zasiela príkazy a od neho prijíma, analyzuje a zobrazuje nazbierané výsledky.

Tvorcovia produktu sa zamerali na stabilitu a použiteľnosť nástroja. Možno konštatovať, že obidva ciele boli splnené na vysokej úrovni. Hlavnou črtou je tzv. *on-demand profiling.* Agent je schopný bežať v rámci JVM ako spiaci proces, na požiadanie začať zberať profilovacie údaje a opäť sa vrátiť do spiaceho stavu ľubovoľný počet krát. Profilovacie údaje, ktoré agent zbiera sú vzorkované aj presné sledovanie metód, sledovanie vytvárania objektov a telemetria JVM. Práca agenta nie je závislá od spustenej front-end časti, agent si všetky nazbierané výsledky ukladá u seba. Po pripojení na agenta je možné od neho stiahnuť výsledky za približne poslednú hodinu (závisí od dostupnej pamäte). Toto platí pre všetky typy profilov, ktoré nástroj poskytuje.

Použiteľnosť je ďalej dosiahnutá možnosťou integrácie do viacerých vývojových prostredí. Integrácia znamená jednoduché spustenie programu v profilovacom móde z vývojového prostredia a spätná navigácia z výsledkov profilovania na miesta v zdrojovom kóde vo vývojovom prostredí. Agent je schopný bežať na virtuálnych strojoch firmy SUN vo verziách 1.3, 1.4, 5.0, 6.0, firmy BEA od verzie 5.0 R26 a firmy IBM vo verzii 5.0 SR1 a vyššie. Súvisia s tým obmedzenia niektorých funkcií, keďže jedine SUN virtuálne stroje úplne spĺňajú predpísanú špecifikáciu. Napriek tomu je táto vlastnosť unikátna medzi profilovacími nástrojmi.

Je možné definovať filtre, ktoré špecifikujú množinu metód a objektov, ktorých volania a vytváranie majú byť sledované. Filtre tiež zužujú množstvo zobrazovaných výsledkov. Filtre pre metódy sa definujú regulárnymi výrazmi, ktoré sa aplikujú na plné mená tried. Filtre pre objekty umožňujú v XML formáte špecifikovať zložitejšiu logiku podobnú tej, akú poskytuje *OQL* jazyk v

nástroji HAT. Zaujímavou vlastnosťou filtrov je, že ak už raz množina výsledkov obsahuje informáciu, ktorá nespĺňa podmienku filtra, no informácia predstavuje drahé volanie alebo vytváranie objektov, vo výsledkoch sa aj tak objaví. Mnoho podobných drobností posúvajú použiteľnosť nástroja na vysokú úroveň.

YourKit profiler poskytuje nasledovné profily:

- Telemetria: zaťaženie procesorového času v čase, zaplnenie pamäťovej haldy, práca GC procesu v rámci JVM, počet nahraných tried, počet aktívnych vlákien,

- CPU profily (vzorkované aj detailné): *CCT* strom, *MBTT* strom, tabuľku metód a špeciálny profil, tzv. *method merged calees view*, kde sa pre zvolenú metódu zobrazia všetky metódy (celé kontexty volaní), ktoré z nej boli volané vo všetkých vláknach. Ide teda o obrátenú verziu *MBTT* stromu.

- Pamäťové profily:

  - Zoznam tried, strom tried podľa príslušnosti k balíkom, pre obidva pohľady počet objektov pre triedy, veľkosť pamäte, ktorú objekty zaberajú, veľkosť pamäte rozšírená o všetky veľkosti objektov, na ktoré sa priamo alebo nepriamo odkazuje

  - Zoznam individuálnych „najväčších“ objektov podľa pamäte, ktorú by bolo možné uvoľniť pri ich odstránení.

  - Zoznam inštancií objektov pre vybranú triedu a pre každý z nich už spomínané *incoming references, outgoing references a GC root paths.* Špecialitou sú objekty *java.lang.String*, pri ktorých sú zobrazované ich hodnoty. Na základe toho je možné prehľadávať haldu aj podľa zadaných reťazcov.

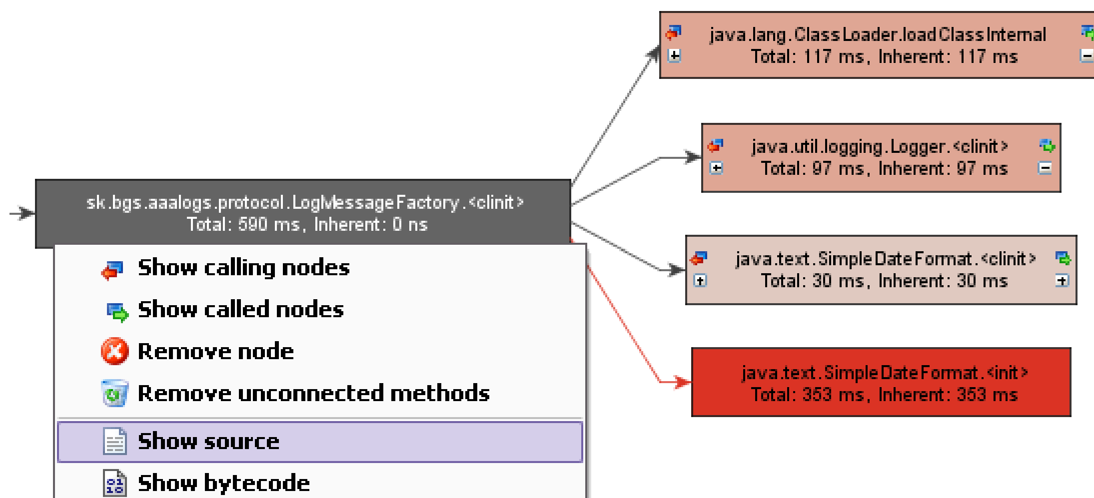
  - *CCT* strom a tabuľka metód podľa počtu a veľkosti vytváraných objektov.

  - Import grafu haldy zo súboru v HPROF formáte.

4.3 JProfiler

JProfiler [37] je druhý z dvojice predstavených komerčných produktov. Množinou funkcií je veľmi podobný nástroju YourKit profiler, uvedené preto budú hlavne črty, v ktorých sa odlišujú. V poslednej verzii 4.2 architektúra opäť rozlišuje agenta a používateľské rozhranie, obidva

komponenty sú však na sebe závislé. Agent vždy pozastaví štart JVM a čaká na pripojenie. Pomocou používateľského rozhrania sa musí zadefinovať typ sledovaných dát a až potom sa pustí samotná aplikácia. Používateľské rozhranie možno od agenta síce odpojiť a znovu pripojiť, typ sledovaných dát sa však už počas behu nedá zmeniť a vtedy je nutný reštart. Preto je nástroj určený skôr na profilovanie vyvíjaných aplikácii a nie aplikácii v prevádzke. Online dokumentácia je jedna z najlepších, popisuje technické princípy profilovania a spôsoby interpretácie výsledkov a je vhodná pre používateľov, ktorí sa ešte neorientujú v problematike profilovania.



Obrázok 22: JProfiler - graf volaní metód

JProfiler disponuje najväčším množstvom typov poskytovaných profilov zo všetkých dostupných nástrojov:

- Telemetria: miera zaplnenia haldy, počet vytvorených objektov, počet objektov odstránených GC procesom, počet nahraných tried, počet vlákien
- CPU profily: *CCT* strom, tabuľka metód a graf volaní metód. Graf volaní metód je grafová reprezentácia *CCT* stromu, v ktorej je možná navigácie medzi metódami podľa toho, ako boli medzi sebou navzájom volané (obrázok 22)

- Pamäťové profily: rozdielom oproti YourKit produktu je chýbajúci zoznam „najväčších objektov“, zato však JProfiler poskytuje jedinečnú funkcionalitu zobrazovať kumulatívne referencie medzi množinami objektov (obrázok 23), zatiaľ čo všetky ostatné nástroje vedia zobrazovať navigáciu a hľadať cesty len pre jednotlivé objekty. Pomocou tohto profilu je veľmi jednoduché identifikovať veľké kolekcie (*java.util.Collection*) objektov rovnakého typu, ktoré sú najčastejším dôvodom *memory-leak*-ov.

Current object set: 2896 instances of java.lang.String
2 selection steps, 67 kB shallow size, calculate deep size

Cumulated incoming references | Show counts and sizes of reference holders | Use ...

Reference type	Object count	Size
field **key** of java.util.HashMap$Entry	1 284	30 816 bytes
class array content	253	7 408 bytes
field **key** of java.util.Hashtable$Entry	168	4 032 bytes
field **value** of java.util.Hashtable$Entry	158	3 792 bytes
field **key** of java.util.LinkedHashMap$Entry (defined by ja	110	3 520 bytes
field **value** of java.util.LinkedHashMap$Entry (defined by	68	2 176 bytes
field **protocol** of java.net.URL	60	3 360 bytes
field **host** of java.net.URL	60	3 360 bytes
field **file** of java.net.URL	60	3 360 bytes
field **authority** of java.net.URL	60	3 360 bytes
field **path** of java.net.URL	60	3 360 bytes
field **val** of java.io.ExpiringCache$Entry	40	960 bytes
field **name** of java.security.Provider$EngineDescription	30	720 bytes

Obrázok 23: JProfiler - kumulatívne referencie medzi objektami

- Profily vlákien:
  - Časový priebeh stavu vlákien a tabuľka súčasného stavu vlákien.
  - Detekcia *deadlock*-ov: grafická reprezentácia čakania vlákien na prístup k monitorom.
  - Zoznam monitorov, ich vlastník a čakatelia (vlákna), pre každého čakateľa čas čakania, aktuálne kontexty volaní metód pre vlastníka a čakateľov. História a štatistika používania monitorov.

4.4 NetBeans profiler

Netbeans profiler je modul s profilovacou funkcionalitou dodávaný výlučne ako súčasť vývojového prostredia Netbeans od firmy SUN. Vedúcou postavou projektu je Misha Dmitriev [27]. Od roku 2000 pracoval na implementácii *HotSwap* funkcionality JVMDI rozhrania. Možnosť dynamickej zmeny kódu tried za behu aplikácie ho priviedli na myšlienku implementovať profilovací nástroj, ktorý by vedel svoju činnosť v ľubovoľných momentoch behu aplikácie zapínať a vypínať. Projekt

bol nazvaný JFluid a zameriaval sa výlučne na profilovanie efektivity kódu. Nástroj fungoval len na špeciálne upravenej verzii JVM. Projekt bol jedným z hlavných impulzov, ktoré viedli ku konečnej špecifikácii profilovacej architektúry JSR-163.

Z projektu JFluid sa stal NetBeans profiler ako neoddeliteľná súčasť vývojového prostredia NetBeans. Neexistuje možnosť integrácie do iných vývojových prostredí. Je voľne distribuovaný, zdrojový kód však nie je k dispozícii. Architektúra opäť rozoznáva agenta, ktorý nie je závislý od front-end časti. Mód profilovania možno za behu ľubovoľným spôsobom meniť, v poslednej verzii 5.0 je však táto funkcionalita stále nespoľahlivá a môže viesť k pádu sledovaného JVM procesu. Aj kvôli veľkej závislosti na vývojovom prostredí je nástroj určený skôr pre profilovanie aplikácií vo vývoji.

Špecialitou nástroja je metrika nazvaná „prežívajúce generácie". Pri každom cykle GC procesu sa všetkým objektom, ktoré upratovanie haldy prežijú, zvýši číslo generácie. Metrika hovorí o počte rôznych hodnôt generácií objektov z jednej triedy. Ak je číslo vysoké, znamená to, že objekty nejakej triedy kontinuálne vznikali počas behu aplikácie a z každej generácie prežila nejaká ich podmnožina. Táto situácia je charakteristická pri *memory-leak*-och s pomalou tendenciou rastu. Treba však poznamenať, že možnosť traverzovať graf haldy, ktorý prinieslo rozhranie JVMTI a úspešné ho využívajú dva spomenuté komerčné produkty, tieto problémy odhaľujú efektívnejšie.

NetBeans profiler poskytuje oproti komerčným produktom menšiu skupinu profilov:

- Telemetria: Zaplnenie pamäťovej haldy, podiel GC procesu na práci JVM, počet aktívnych vlákien, maximálna hodnota prežívajúcich generácií.

- CPU profily: *CCT* strom, *MBTT* strom, tabuľka metód, profilovanie vybraného rozsahu riadkov v zvolenej metóde prezentované len počtom prechodov a celkovým časom, ktoré aplikácia strávila vo vybraných príkazoch. Nie je možné vzorkované profilovanie. Je nutné zadať množinu vstupných metód, od ktorých sa má sledovať kontext volaní. Dôvodom, prečo sa koreňové metódy musia zadávať explicitne je pôvodná myšlienka mať možnosť zúžiť sledovanie volania metód na podmnožinu aplikácie. Nástroj však nezačne sledovať zadanú metódu, pokiaľ sa tá v aktuálnom momente nachádza v nejakom kontexte volaní. Tento fakt spôsobuje veľké obmedzenie pri sledovaní už bežiacich aplikácií. Navyše, pokiaľ je profilovaná aplikácia, ktorá nebola spustená z prostredia NetBeans, je potrebné zadať koreňovú metódu ručne, čo vyžaduje znalosť o vnútornej štruktúre aplikácie.

- Pamäťové profily:

    - tabuľka tried s počtom a celkovou veľkosťou všetkých jej inštancií a hodnota prežívajúcich inštancií.

    - *CCT* strom podľa počtu a veľkosti vytváraných objektov.

    - *MBTT* strom pre vytváranie inštancií objektov zvolenej triedy. Ide v podstate o *MBTT* strom, kde sledovaná metóda predstavuje volania konštruktorov objektov triedy.

- Profily vlákien: sledovanie stavu vlákien v čase, detaily stavov zvoleného vlákna. Informácie o monitoroch nie sú poskytované.

4.5 Eclipse TPTP

TPTP (The Eclipse Test & Performance Tools Platform) je framework pre vývojové prostredie Eclipse určených na tvorbu nástrojov pre podporu monitorovania, testovania a profilovania. Definuje spoločnú infraštruktúru, používateľské rozhrania, dátové modely a podporné knižnice. Časť projektu, ktorá sa zaoberá profilovaním, obsahuje sadu niekoľkých hotových modulov. Profilovacie moduly sú však veľmi silno previazané s vývojovým prostredím a profilovaniu vzdialených aplikácií predchádza pomerne komplikovaná konfigurácia. Zdrojové kódy projektu sú voľne dostupné.

Projekt poskytuje veľmi malé množstvo profilov:

- CPU profily:

    - Tabuľky metód s možnosťou ich grupovania podľa príslušností k triedam a balíkom.

    - UML diagramy, ktoré zachytávajú tok volania metód v programe. Vytvorené diagramy sú však veľmi rozsiahle a je skoro nemožné s v nich orientovať.

    - Profil pokrytia kódu počas behu aplikácie. Profil slúži skôr na testovacie účely, hovorí o kóde metód, ktorý nebol nikdy vykonaný.

- Pamäťové profily: len tabuľka tried s počtom a veľkosťou vytvorených objektov.

4.6 Java Interactive Profiler

Posledným uvádzaným nástrojom je Java Interactive Profiler (JIP) [38]. JIP je profiler implementovaný ako agent, 100% v jazyku Java, bez natívnych komponentov. Zdrojové súbory sú voľne dostupné, preto je JIP výbornou ukážkou funkčného projektu postaveného na novej profilovacej architektúre JSR-163 a dynamickej inštrumentácii bajtkódu. JIP je dodávaný ako súbor *profile.jar*, spúšťa sa parametrom *-javaagent:profile.jar* príkazu *java*.

JIP vytvára profily len v textovom formáte. Sleduje volania metód a vytváranie objektov. Na zvolený port mu možno zasielať dvojicu príkazov *start* a *stop*. Pri štarte sa vynulujú všetky doterajšie výsledky, pri zastavení sa vypíšu do súboru. Je preto možné sledovať zvolené časové úseky behu aplikácie. V konfigurácii sa dajú nastaviť rôzne filtre a parametre, ktoré špecifikujú mieru sledovaných detailov. Sú nimi napríklad maximálna hĺbka sledovaných kontextov volaní, množiny tried.

JIP vytvára nasledovné profily:

- CPU profily: *CCT* strom, tabuľka metód
- Pamäťový profil: zoznam tried podľa počtu vytvorených inštancií

4.7 Porovnanie profilovacích nástrojov

Nasledovne porovnanie je subjektívnym vyhodnotením autora tejto práce, ktoré vychádza zo skúseností za obdobie kedy s nimi pracoval. Nie je možné úplne exaktne porovnať profilovacie nástroje, keďže každý má svoje špecifiká, preto výber vhodného profilovacieho nástroja ovplyvňuje viacero faktorov:

1. Cenová dostupnosť - NetBeans profiler. V prípade, kedy nie je možné zakúpiť komerčný produkt, je najlepšou voľbou. Napriek nedoriešeným technickým nedostatkom disponuje najväčšou množinou poskytovaných profilov z voľne dostupných produktov.

2. CPU profilovanie - YourKit profiler. Vzorkované aj detailne zbieranie dát možno aktivovať a deaktivovať v ľubovoľnom čase, výsledky sú zobrazované vo veľmi prehľadnej forme a viacerých stromových aj tabuľkových reprezentáciách. Aplikácia filtrov nikdy nespôsobuje stratu relevantných informácií.

3. Pamäťové profilovanie - YourKit profiler a JProfiler. Profil „najväčších" objektov (YourKit)

a kumulatívne referencie (JProfiler) predstavujú najlepšie dostupné spôsoby automatickej analýzy grafu haldy.

4. Profilovanie vlákien – JProfiler. Okrem základnej telemetrie ako jediný produkt poskytuje sledovanie používania monitorov vo viacerých grafických zobrazeniach.

5. Podpora vývoja aplikácie – YourKit profiler a JProfiler. Výsledok vyplýva z bodov 2, 3 a 4. Navyše obidva nástroje umožňujú ľahkú integráciu do rôznych vývojových prostredí a rešpektujú tak už zavedenú konfiguráciu vývoja aplikácie.

6. Profilovanie produktov nasadených v prevádzke – YourKit profiler. Schopnosť agenta bežať samostatne v aktívnom aj spiacom móde, zmeny módu profilovania na požiadanie počas behu aplikácie, vysoká stabilita, nízka záťaž sledovaného procesu.

7. Tvorba vlastného profilovacieho nástroja – HPROF a JIP. K obidvom produktom sú voľne dostupné zdrojové kódy, nezahŕňajú používateľské prostredie, majú relatívne malý rozsah preto sú vhodné na štúdium ako referenčná ukážka použitia profilovacej architektúry.

5 Začlenenie profilovania do procesu vývoja softvéru

To, aký typ informácie profily programov predstavujú by malo byť po prečítaní druhej a tretej kapitoly zrejmé. Nezodpovedanou otázkou zostáva, prečo, kedy a ako profilovať. Nasledujúce štyri podkapitoly uvádzajú štyri typické situácie, kedy začlenenie profilovania do procesu vývoja a údržby softvéru môže významnou mierou k vyriešeniu problémov.

5.1 Hľadanie chýb v aplikácii

Vo fáze implementácie a testovania vyvíjanej aplikácie je bežným javom, že aktuálna verzia softvéru nespĺňa niektorú z funkčných požiadaviek. Oveľa nepríjemnejšou situáciou je, keď sa chyba v aplikácii odhalí až po jej nasadení do produkčnej prevádzky. Kód sa vracia do rúk programátorom spolu s výsledkami testu alebo prevádzky. Výsledkami sú spravidla logy aplikácie, popis počiatočného a koncového stavu dát, s ktorými aplikácia pracovala a ostatné vonkajšie prejavy aplikácie (výpisy na obrazovku, stav systému počas behu aplikácie). Programátor sa na základe dodaných dát snaží v kóde nájsť miesto zodpovedné za nefunkčné správanie. Chybové miesto hľadá statickou analýzou kódu, pričom si v mysli prehráva beh programu, tak ako si ho dal do spojitosti s dodanými výsledkami. V mnohých prípadoch je tento spôsob analýzy veľmi náročný. Pomôckou je proces ladenia programu v prípade, že je možné simulovať rovnaké podmienky, v ktorých chyba vznikla a zároveň ak existuje aspoň približný odhad príčiny chyby. Bežné sú situácie, kedy programátor jednoducho netuší, čo chybu spôsobuje a je nútený začať proces ladenia naozaj „zoširoka“ a postupne zužovať oblasť, v ktorej sa môže chyba nachádzať.

Proces ladenia „zoširoka“ môže byť nahradený procesom profilovania. Väčšina profilov obsahuje už len súhrnné informácie o behu programu, z ktorých nemožno spätne odvodiť presný priebeh programu a už vôbec nie stavy (hodnoty premenných a podobne), v ktorých sa aplikačná logika nachádzala. Napriek tomu môže programátor vytvorený profil porovnať s predpokladaným správaním programu, s tým, čo očakáva, že by mal profil ukázať. Profil na rozdiel od plného výpisu logov predstavuje kompaktnú informáciu a analýza takejto informácie sa spravidla končí dvoma extrémnymi spôsobmi. Buď sa veľmi rýchlo zistí, že vytvorený profil nehovorí nič o tom, kde sa chyba nachádza (vývojový tím tak nestratil mnoho času), alebo sa v rovnako krátkom čase potvrdí opak a profil poukáže priamo alebo približne na miesto chyby. Teória sa potom overí zameraním sa špeciálne len na indikovanú oblasť.

5.2 Optimalizácia výkonu aplikácie

Druhým typom chýb v aplikáciach je nesplnenie niektorej z nefunkčných požiadaviek. Jedným z typov nefunkčných požiadaviek je aj výkon aplikácie, definovaný nejakou kvantitatívnou metrikou. To, že aplikácia neplní úlohy s očakávaným výkonom sa navonok prejavu tým, že nestíha plniť množstvo zadaných úloh v predpísanom čase. Dôvodom môže byť nedostatočný hardvérový výkon, alebo neoptimálna implementácia funkcií aplikácie.

Ak existuje podozrenie na neoptimálny kód, opäť vzniká potreba jeho lokalizácie. Častým prístupom k riešeniu výkonnostného problému je jednoduchý odhad problému bez toho, aby bol potvrdený. Situácia sa môže skončiť tak, že programátor venuje množstvo času na optimalizáciu kódu, ktorý je vykonávaný relatívne zriedkavo alebo kódu, ktorý neoptimálny vôbec nebol a odhad vznikol na dohadoch a nedostatočných znalostiach a skúsenostiach. Optimalizácia navyše často spôsobuje odchýlenie sa od štandardného dizajnu kódu, preto nepotrebné optimalizácie alebo optimalizácie s nízkym prínosom môžu spôsobiť viac škody ako úžitku. Nepísané pravidlo hovorí, že prehľadný kód je lepší ako 0,5% zlepšenie výkonu.

V tomto prípade slúži profilovanie nielen na potvrdenie odhadov o príčine výkonnostného problému, ale vo väčšine prípadov tieto príčiny priamo odhaľuje. Dlhodobé skúsenosti s profilovaním aplikácii navyše potvrdzujú tzv. Paretovo alebo 80-20 pravidlo: „80% času zo svojho výpočtu trávi aplikácia v 20% svojho kódu“ [3]. Ak sa toto pravidlo aplikuje na rozhodnutia pri dizajne aplikácie, potom možno dobrý a prehľadný dizajn skoro vždy uprednostňovať pred optimálnym dizajnom. Vo fáze implementácie softvéru, kedy sa objavia prvé neoptimálnosti, profilovanie zvyčajne ukáže, že ich môžno odstrániť zmenami lokálneho charakteru, ktoré dobrý dizajn narušia len vo veľmi malej miere. Ak aj nebude možné realizovať optimalizáciu jednoduchým spôsobom, ale jedine rozsiahlym zásahom do dizajnu, bude zásah opodstatnený výsledkami profilovania.

Pri profilovaní výkonu programu si treba uvedomiť, že prítomnosť profilovacieho mechanizmu nevyhnutne program spomaľuje a výsledky nemusia celkom presne zohľadňovať reálny výkon programu. Ak program počas profilovania beží 10-20 krát pomalšie ako v normálnej situácii, výsledky už môžu byť veľmi nepresné a je potrebné nastavením filtrov alebo iným spôsobom znížiť mieru detailov získavaného profilu. V prostredí *real-time* systémov a distribuovaných systémov môže prítomnosť profilovacieho mechanizmu spôsobiť tzv. Heisenbergov efekt a spôsobiť zlyhanie programu pri plnení funkčných požiadaviek. Aj na tieto skutočnosti treba brať ohľad pri konfigurácii profilovania a interpretácii jeho výsledkov.

5.3 Predikcia budúceho správania aplikácie

Testovanie softvéru sa zvyčajne končí konštatovaním, že softvér splnil všetky funkčné a nefunkčné požiadavky. Softvér počas testovania bežal v konfiguráciách a pri zaťažení, ktoré sa snažia čo najvernejšie simulovať podmienky reálnej prevádzky. Počas testovania však skoro nikdy nemožno pokryť všetky situácie, ktoré môžu v prevádzke nastať a aplikácia prejde fázou testovania, pričom obsahuje chybu, ktorá sa môže prejaviť v tom najnevhodnejšom momente počas prevádzky.

Doplnenie fázy testovania o vytváranie profilov testovaného softvéru umožňuje tieto profily skúmať a overiť si, či sa aplikácia správala aj vo vnútri tak, ako sa predpokladalo. Vytváranie profilov by sa dalo realizovať aj priebežne počas samotnej prevádzky, pričom je ale potrebné brať ohľad na konfiguráciu, ktorá spôsobí len nízke ovplyvnenie behu programu. Profily by mohli odhaliť situácie, kedy sa vnútorný stav aplikácie nebezpečne priblížil k jej potencionálnemu pádu. Profily by tiež mohli odpovedať na otázku, ako sa bude softvér správať pri budúcich zmenách konfigurácie (napr. výmene databázy). Zistené nedostatky možno potom opraviť preventívne.

5.4 Dynamická analýza neznámeho kódu

Netradičným spôsobom využitia profilovania je pomôcka pri analýze a snahe o pochopenie neznámeho kódu. Neznámym kódom sa zaoberá napríklad človek, ktorý prebral projekt po svojom predchodcovi, človek, ktorý je nútený pochopiť implementačné detaily knižnice dodávanej nejakou treťou stranou alebo proces *re-engineering*-u. Klasickým prístupom je čítanie dokumentácie softvéru a manuálna analýza kódu. Dokumentácia však nemusí byť vždy dostatočne kvalitná a manuálna analýza kódu je relatívne náročná úloha, pri ktorej je nutné si v mysli predstavovať, čo kód po spustení robí. Pomôckou pri tejto úlohe je profil, ktorý poskytuje pohľad na kód, pohľad, ktorý je inak náročný na predstavu.

6 Prípadové štúdie

6.1 Prepaid Billing Server

6.1.1 Popis produktu

Prepaid Billing Server (ďalej len server) je softvérový systém pôvodne určený na spoplatňovanie predplatených dátových služieb. Každý používateľ služby má k dispozícii vlastný účet s naplneným kreditom, po prihlásení k službe sa mu v jednominútových intervaloch začína znižovať stav kreditu.

Predplatená služba je fyzicky reprezentovaná hardvérovým prístupovým zariadením. Prístup k službe pre používateľa je realizovaný obsadením jedného portu zariadenia po dobu celého trvania prístupu. Zariadenie zasiela serveru pri vzniku a ukončení prístupu správu. Rovnako zasiela správy v pravidelných intervaloch počas celej doby trvania prístupu, ktoré informujú o tom, že prístup stále trvá.

Server je zodpovedný za príjem zasielaných správ. Z každej správy extrahuje informácie o používateľovi a prístupe. Na základe týchto a ďalších[10] informácii postupne znižuje stav účtu daného používateľa. V prípade vyčerpania kreditu na účte server zašle externému zariadeniu žiadosť o odpojenie prístupu pre daného používateľa.

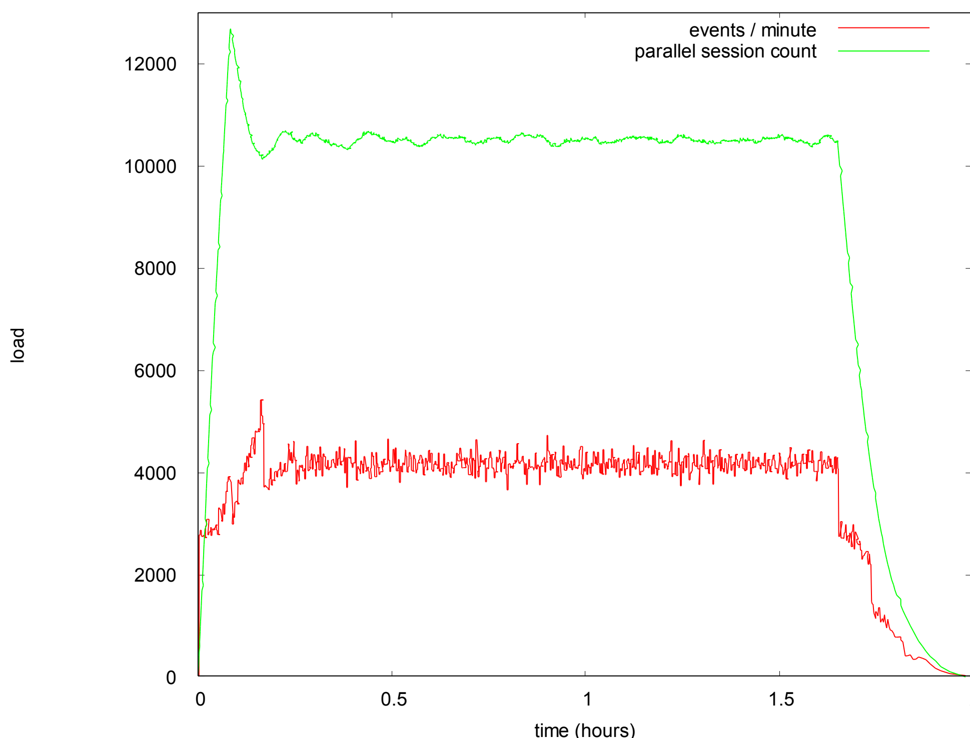
Hlavná dizajnová črta servera je rozdelenie práce do jedného hlavného a viacerých pracovných vlákien. Hlavné vlákno je zodpovedné za príjem správ zo siete od prístupového zariadenia a ich prerozdeľovaniu pracovným vláknam. Pracovné vlákna sú zodpovedné za spracovanie jednotlivých správ. Spracovanie znamená na základe správ modifikovať údaje, ktorými sú reprezentované aktívne prístupy a stavy účtov používateľov. Tieto údaje sa nachádzajú v pamäti aj v databáze a pracovné vlákna k nim musia pristupovať tak, aby bola zachovaná ich integrita.

Produkt pred nasadením úspešne prešiel testovaním pri predpokladanej záťaži. Bezchybná bola aj jeho samotná prevádzka. Po nejakom čase sa objavila otázka, či by mohlo byť riešenie použité aj na spoplatňovanie iných typov služieb, kde sa však predpokladala oveľa vyššia záťaž

10 Napríklad rozlišovanie medzi pracovným dňom, sviatkom alebo dennou a nočnou prevádzkou.

6.1.2 Konfigurácia testu

Pre potreby testovania bol vytvorený samostatný softvérový modul, určený na simulovanie prístupov k predplatenej službe. Simulátor na základe konfigurácie generuje v presných časových intervaloch správy v rovnakom formáte ako prístupové zariadenie.



Obrázok 24: Vygenerovaná záťaž

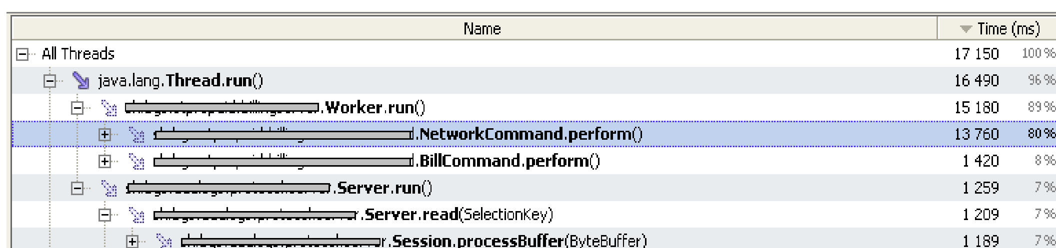
Podľa požiadavky mala byť simulovaná dvojhodinová záťaž s dlhodobým priemerom 10000 paralelných prístupov. Z obrázku 24 vidno, že server mal prijať každú minútu v priemere 4000 správ o stave na prístupovom zariadení. Z toho vyplýva potreba vyrátať každú minútu spoplatnenie pre vyše 10000 prístupov pre približne rovnaké množstvo používateľov. Išlo o rádovo stonásobné vyššiu záťaž ako v doterajšej prevádzke.

Počas prvého spustenia testu bolo monitorované iba celkové zaťaženie procesorov pomocou štandardného nástroja operačného systému. Výsledok ukázal dlhodobé zaťaženie pod hranicou 20%. Napriek tomu ukázala kontrola výsledného stavu dát v databáze, že server nespracoval veľké množstvo prístupov. Problémy počas testu tiež hlásil generátor správ, ktorému server nestíhal odoberať správy. Následná kontrola log súboru ukázala, že server sa pravidelne nachádzal v stave, kedy nebolo k dispozícii žiadne voľné pracovné vlákno a modifikácia stavu účtu musela byť spustená v hlavnom programovom vlákne. Táto situácia je nežiadúca, keďže vedie zablokovaniu

hlavného vlákna, ktoré potom nie je schopné prijímať zo siete všetky prichádzajúce správy. To pravdepodobne viedlo k ich stratám, čo malo za následok stratu informácií o prístupoch.

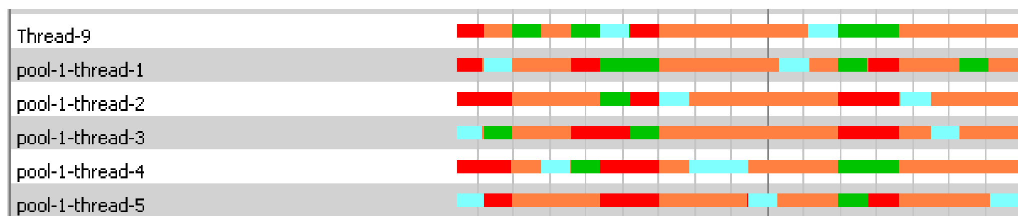
6.1.3 Profilovanie a zistené skutočnosti

Bolo potrebné zistiť dôvod vyťaženosti pracovných vlákien. Prvým krokom bolo profilovanie programového kódu, ktorý v nich beží. Podľa očakávania mal server počas výpočtu tráviť najviac času práve v metódach, ktoré spracúvajú prichádzajúce správy. Použitý bol *YourKit Profiler* v režime vzorkovania metód. Profilovanie bolo spustené na dobu 60 sekúnd, výsledok ukázal, že všetky pracovné vlákna si na spracovanie prichádzajúcich správ vyžiadali spolu len približne 14 sekúnd procesorového času (obrázok 25). Samotné čítanie správ zo siete trvalo len niečo vyše jednej sekundy. Prepočítanie stavu účtov prebehlo raz a jedno pracovné vlákno úlohu splnilo pod



Obrázok 25: Spracovanie prichádzajúcich správ

dve sekundy. Keďže počas jednej minúty nebežali na hardvéri okrem testovanej aplikácie žiadne iné výpočtovo náročné procesy, aplikácia zjavne nevedela efektívne využiť prostriedky systému. Ďalším krokom bolo preto preskúmanie stavu pracovných vlákien. Monitorovanie bolo realizované nástrojom *JProfiler*. Výsledok je znázornený na obrázku 26, ktorý odhalil prekvapujúce zistenie. Väčšinu času trávili pracovné vlákna v nečinnom stave, napriek tomu, že práce bol dostatok. Pôvodný zámer zaviesť do servera *load balancing* nebol vôbec dosiahnutý. Spracovanie správ bolo obmedzované prítomnosťou jedného alebo viacerých úzkych hrdiel.

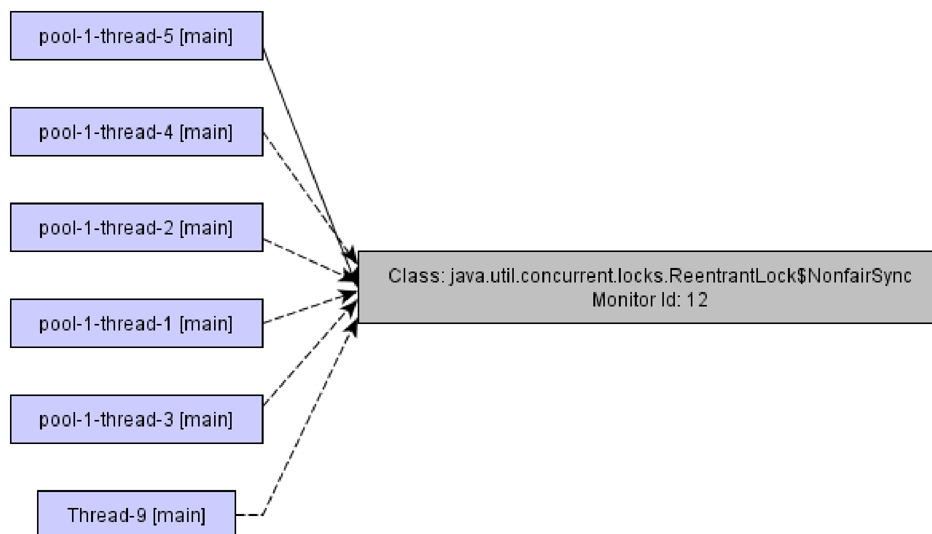


Obrázok 26: Stav pracovných vlákien servera

6.1.4 Riešenie problému

Distribúcia intervalov, v ktorých sa pracovné vlákna nachádzali, vykazovala vysokú pravidelnosť a nemohla byť náhodná. Z toho jasne vyplývalo, že činnosť vlákien musela byť obmedzovaná synchronizačným mechanizmom prítomným priamo v kóde, pri práci so sieťovou komunikáciou

alebo pri prístupe do databázy. V následnej revízii kódu preto musela byť prehodnotená nutnosť synchronizácie v doterajšom rozsahu. Odstránenie nepotrebnej synchronizácie by umožnilo dosiahnuť žiadaný výsledok skutočne paralelnej práce vlákien. V opačnom prípade by bolo efektívnejším riešením jediné pracovné vlákno, čím by sa minimálne ušetrila réžia spojená so synchronizáciou.



Obrázok 27: Čakanie nad monitormi v hlavnom a pracovných vláknach (graf)

Identifikácia synchronizačných mechanizmov v kóde bola realizovaná opäť nástrojom *JProfiler*. Grafické znázornenie čakania nad monitormi ukazovalo viacmenej stále tú istú situáciu, čakanie nad monitorom v triede *ReentrantLock* (obrázky 27 a 28).

Duration	Type	Monitor ID	Monitor class	Waiting thread	Owning thread
90 ms	Blocked	12	java.util.concurrent.locks.Reentrant…	pool-1-thread-2 [main]	pool-1-thread-1 [main]
69 ms	Blocked	12	java.util.concurrent.locks.Reentrant…	pool-1-thread-1 [main]	pool-1-thread-4 [main]
49 ms	Blocked	12	java.util.concurrent.locks.Reentrant…	pool-1-thread-4 [main]	pool-1-thread-3 [main]
29 ms	Blocked	12	java.util.concurrent.locks.Reentrant…	pool-1-thread-3 [main]	pool-1-thread-5 [main]
9252 µs	Blocked	12	java.util.concurrent.locks.Reentrant…	pool-1-thread-5 [main]	pool-1-thread-2 [main]
5305 µs	Waiting	11	java.util.concurrent.locks.AbstractQ…	Kill Service [main]	

Obrázok 28: Čakanie nad monitormi v hlavnom a pracovných vláknach (tabuľka)

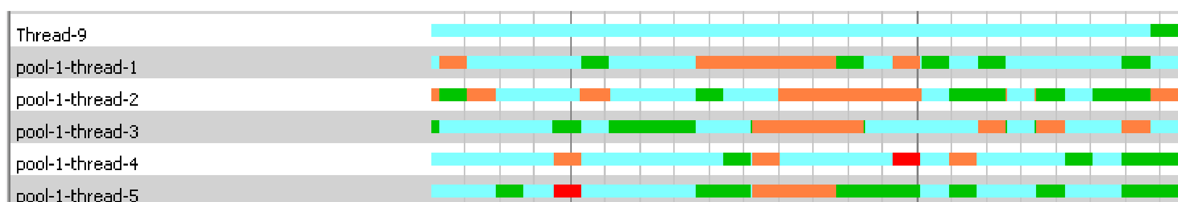
Treba poznamenať, že doteraz prebiehala celá analýza problému bez toho, aby bol skúmaný samotný zdrojový kód. Výsledky profilovania vlákien však vo veľkej miere naznačili, v ktorých častiach kódu sa budú problémy nachádzať. Po analýze samotného kódu bol správne potvrdený predpoklad príliš silnej a konzervatívnej synchronizácie. Každé pracovné vlákno si pomocou mechanizmu triedy *ReentrantLock* pri časti svojej práce uzamykalo prístup ku globálnej kolekcii, ktorá obsahuje reprezentácie užívateľských prístupov k službe. Kód bol následne optimalizovaný tak, že vlákna si uzamykali prístup len k jednotlivým objektom z kolekcie. Kolekcia ako celok sa zamykala len v prípade pridávania alebo odstraňovania jej prvkov (pri vzniku a zániku prístupu k službe). Správnosť implementácie bola potvrdená zopakovaním profilovania stavu vlákien. Objekt

ReentrantLock sa ako monitor objavoval už len vo veľmi zriedkavých prípadoch, kedy dve rôzne vlákna náhodou dostali na spracovanie správy týkajúce sa toho istého prístupu. Úprava kódu však priniesla len nepatrné zvýšenie výkonu. Počas práce vlákien sa navyše objavilo čakanie nad synchronizovaným kódom v triedach *OraclePreparedStatement* a *OracleConnection*, ktoré reprezentujú prístup k údajom v databáze (obrázok 29). Táto skutočnosť odhalila nielen výkonnostný problém, ale aj implementačnú chybu, ktorá sa doteraz neprejavovala a do programu bola zavedená predchádzajúcou úpravou. Vlákna používali pri prístupe k databáze spoločné rozhranie, ktoré nedostatočne zohľadňovalo paralelné prístupy. V konečnom dôsledku umožňovalo viacerým vláknam naraz modifikovať stav niektorých objektov a narúšať tak integritu dát.

Duration	Type	Monitor ID	Monitor class	Waiting thread	Owning thread
9444 µs	Blocked	18	oracle.jdbc.driver.OraclePreparedSt...	pool-1-thread-2 [main]	
402 µs	Blocked	18	oracle.jdbc.driver.OraclePreparedSt...	pool-1-thread-4 [main]	pool-1-thread-2 [main]
9986 µs	Blocked	18	oracle.jdbc.driver.OraclePreparedSt...	pool-1-thread-2 [main]	pool-1-thread-4 [main]
234 µs	Blocked	18	oracle.jdbc.driver.OraclePreparedSt...	pool-1-thread-1 [main]	pool-1-thread-2 [main]
10 ms	Blocked	18	oracle.jdbc.driver.OraclePreparedSt...	pool-1-thread-2 [main]	pool-1-thread-1 [main]
25 ms	Blocked	18	oracle.jdbc.driver.OraclePreparedSt...	pool-1-thread-4 [main]	pool-1-thread-2 [main]
410 µs	Blocked	18	oracle.jdbc.driver.OraclePreparedSt...	pool-1-thread-2 [main]	pool-1-thread-4 [main]
6112 µs	Blocked	18	oracle.jdbc.driver.OraclePreparedSt...	pool-1-thread-1 [main]	pool-1-thread-4 [main]

Obrázok 29: Prístup vlákien k objektom, ktoré reprezentujú prístup k databáze

Problém sa dal vyriešiť zavedením výlučného prístupu k databázovému rozhraniu, alebo vytvorením rozhrania pre každé vlákno samostatne. Celé riešenie predpokladalo výkonnú databázu schopnú pracovať na viacerými paralelnými spojeniami s možnosťou škálovať výkon podľa hardvérového výkonu. Nemalo preto zmysel „šetriť" databázu a implementovalo sa druhé riešenie. Po tejto poslednej úprave server prvýkrát úspešne a bez akýchkoľvek varovných hlásení zvládol celý dvojhodinový test (obrázok 30). Vlákna si už navzájom neblokovali prácu nepotrebným uzamykaním objektov. V nečinnom stave sa nachádzali jedine v momentoch, kedy do systému neprišli žiadne nové správy na spracovanie (oranžová farba) a pri čakaní na prístup k sieťovým zdrojom (modrá farba).



Obrázok 30: Stav pracovných vlákien po optimalizácii servera

6.1.5 Záver

Uvedený produkt obsahoval viacero implementačných nedostatkov. Tieto však nikdy neboli odhalené, keďže produkt úspešne zvládol fázu testovania. V čase dodávky produktu nemal

vývojový tím k dispozícii žiaden profilovací nástroj, ktorý by umožňoval analýzu behu programu v takom rozsahu, ako tu bola uvedená. Profilovanie nebolo súčasťou cyklu vývoja produktu. Je pravdepodobné, že v opačnom prípade by boli nedostatky odhalené skôr, ako by sa vôbec prejavil ich negatívny dopad na výkon programu.

6.2 Identifikácia pamäťových problémov

V praxi bolo profilovanie využité v dvoch reálnych situáciach, kedy bola činnosť aplikácie predčasne ukončená z dôvodu *OutOfMemoryException* výnimky. Zhodou okolností implementovali obidve aplikácie komunikáciu dvoch vzdialených procesov.

Prvý prípad bol ukážkou toho, ako môže byť ťažké aj v kóde veľmi malého rozsahu identifikovať príčinu neúmyselného narastajúceho zaplnenia pamäte. Dvojica procesov si medzi sebou navzájom posielala správy mechanizmom *java.io.ObjectOutputStream (OOS).* Po prenesení niekoľkých desiatok megabajtov dát však jedna alebo druhá aplikácia po niekoľkých minútach padla s *OOME* výnimkou. Skupina programátorov po dlhej chvíli statickou analýzou kódu eliminovala všetky možnosti, kde môžu vytvárané objekty správ „visieť" a jediným neznámym prvok v kóde ostala trieda *OOS*. Dokumentácia triedy explicitne neupozorňuje na žiadne obmedzenia jej používania.

Jednou z možností bolo štúdium zdrojového kódu triedy, ktoré by v konečnom dôsledku odhalilo dôvod, prečo trieda drží referencie na vytvárané objekty. Analýzu urýchlilo profilovanie grafu haldy počas opakovaného testu. YourKit profiler v profile „najväčších" objektov podľa očakávaní určil triedu *OOS* zodpovednú za zaplnenie prakticky celej haldy. Skoro rovnakú veľkosť zaberalo neznáme pole objektov, ktoré sa objavilo na treťom mieste. Vyhľadanie cesty od problémového poľa ku koreňom haldy presne určilo jeho umiestenie v triede *OOS*. Analýzu zdrojového kódu bolo teda možné zúžiť na operácie nad identifikovaným poľom objektov a určiť príčinu ukladania objektov do neho. Trieda totiž umožňuje transport objektov ich serializáciou, pričom medzi nimi zachováva existujúce referencie. Táto funkcionalita vyžaduje, aby si transportný mechanizmus pamätal objekty, ktoré už boli prenesené. Je na zodpovednosti používateľa, aby toto pole objektov explicitne vyprázdnil zavolaním metódy *reset(),* ak ten vie, že ďalšia sada prenášaných objektov neobsahuje referencie na doteraz prenesené objekty. Možno povedať, že až profilovanie triedy nakoniec viedlo k správnemu pochopeniu jej funkcionality.

Druhou aplikáciou bol projekt už pomerne veľkého rozsahu. Jeho moduly implementujú infraštruktúru na routovanie špeciálneho typu správ v sieti. Počas fázy testovania nastala situácia, kedy sa dlhodobé testovanie neúspešne skončilo po približne troch hodinách chybou z dôvodu

zaplnenia pamäte jedného z modulov. Hľadanie chyby tradičnými postupmi by bolo časovo veľmi náročné, keďže replikovať situáciu si vyžadovalo vždy relatívne dlhé čakanie. Dizajn modulu je pomerne zložitý, routované správy sa prechádzajú množstvom kolekcií a niekedy sa aj perzistentne ukladajú na disk a znovu čítajú. Statickou analýzou bolo veľmi ťažké identifikovať problém. Počas druhého spustenia bol nástrojom YourKit získaný graf haldy. YourKit profiler jediným kliknutím a za minimálny čas identifikoval veľkú kolekciu, ktorá nebola nikdy vyprázdňovaná a ukladali sa v nej všetky správy, ktoré kedy prešli modulom. Je zaujímavé, že kolekcia musela pri páde modulu obsahovať niekoľko stoviek tisíc správ a pomerne často v nej boli vyhľadávané správy. Použitá implementácia hash-mapy však bola natoľko efektívna, že modul neprejavoval viditeľné zvýšenie nárokov na procesorový čas ani v neskorých štádiách behu.

6.3 Identifikácia výkonnostných problémov

Profilovanie bolo vykonané na niekoľkých starších aj aktuálne vyvíjaných projektoch. Cieľom bolo nájsť v projektoch neefektívne implementácie lokálneho charakteru a vytvoriť ich zoznam. Získaný zoznam mal byť vo vývoji nových projektov reprezentovať novú množinu znalostí. Okrem iného by zohrával rolu pri dizajnových rozhodnutiach.

Profilovanie objasnilo niektoré dohady, ktoré sa týkali ceny operácie vytvárania nových objektov. O výkone Java virtuálnych strojov existuje mnoho dohadov. Java ako objektovo-orientovaný jazyk sa vyznačuje tým, že prirodzene podporuje dizajn programov, v ktorých je vytvárané veľké množstvo krátkožijúcich objektov. Operácii vytvorenia nového objektu na halde sa prisudzuje vysoká cena operácie vzhľadom na procesorový čas. Technické dokumentácie moderných virtuálnych strojov však vytváranie objektov zaraďujú medzi vysoko optimalizovaný proces. Profilovanie dalo za pravdu druhej strane. Volania konštruktorov väčšiny objektov sa v tabuľkách metód nachádzali na spodných priečkach a predstavovali len zlomok práce programov. Na druhej strane sa zase ukázalo, kedy tento predpoklad použiť nemožno. Volania konštruktorov niektorých objektov sa ukázali ako drahé volania. Profilovanie ich jasne odlíšilo od „lacných“ objektov a na základe toho bolo možné odchýliť sa od jednoduchého dizajnu programu za účelom optimalizácie len tam, kde to bolo potrebné. Príkladom je drahé vytváranie objektov niekoľkých nasledovných tried:

- *java.util.Calendar* – Pri vytvorení si kalendár inicializuje netriviálne vnútorné štruktúry. Oplatí sa používať vždy len jednu inštanciu kalendára a metódou *setTime()* mu meniť dátum, ktorý reprezentuje .

- *java.util.regex.Matcher* – Pri vytvorení sa zadaný regulárny výraz prevádza na objektovú reprezentáciu konečného automatu. Ak sa bude rovnaký regulárny výraz vyhľadávať viackrát, oplatí sa používať tú istú inštanciu objektu. Treba dať pozor na to, aby dve rôzne vlákna nepoužívali ten istý objekt v rovnakom čase. Možným riešením je synchronizácia alebo *ThreadLocal* mechanizmus.

- *java.text.SimpleDateFormat* – Podobne ako predchádzajúci prípad.

- *javax.swing.JFileChooser* – Prvé vytvorenie objektu bez toho, aby bolo dialógové okno zobrazené trvá rádovo stovky milisekúnd, je spôsobené volaním systémovej funkcie a spôsobuje viditeľné prerušenie. Vytvorenie objektu treba pozdržať až do momentu, kedy sa má prvý krát zobraziť.

Druhým opakujúcim sa prípadom bolo nie vždy vhodné použitie štandardnej logovacej architektúry *java.util.logging*. Problematickým bolo volanie metódy *Logger.log(Level level, Object message)*. Volanie bolo vložené aj na miesta programu, ktoré sa vykonávali veľmi často, a malo poskytovať veľmi detailné výpisy. Parameter *level* bol nastavený na najnižšie úrovne a v bežnej prevádzke sa preto správa zadaná v druhom parametri nemala zobrazovať vo výpisoch. Programátorom však unikol fakt, že ako parameter správy zadávali často objekty, ktorých prevod na reťazcovú reprezentáciu metódou *Object.toString()* nie je lacná operácia. A keďže nastavenie úrovne logovania sa kontroluje až vo vnútri metódy log(), operácia prevodu objektu na reťazec sa vykonávala vždy. Profilovanie takéto miesta priamo odhalilo a umožnilo obaliť príkaz logovania do podmienky, ktorá najprv explicitne skontroluje nastavenú úroveň.

7 Záver

Cieľom tejto diplomovej práce bolo predstaviť proces profilovania aplikácií spúšťaných v Java virtuálnych strojoch. Dostupná literatúra a dokumentácia profilovacích nástrojov sa problematikou zaoberajú vždy len čiastkovo. V práci bol prezentovaný pohľad na problematiku z rôznych pohľadov a boli objasnené súvislosti medzi rôznymi technológiami Java platformy, ktoré s profilovaním priamo súvisia.

Postupne boli opísané najdôležitejšie časti špecifikácie jazyka Java a zadefinované štruktúry, ktoré formálne popisujú beh programov implementovaných v jazyku Java. Medzi zadefinovanými štruktúrami a profilmi, ktoré poskytujú v súčastnosti dodávané profilovacie nástroje boli popísané vzájomné vzťahy. Práca tak odpovedá na otázku vzniku ustálených typov profilov, ktoré poskytuje väčšina profilovacích nástrojov. Porovnáva komerčne dodávané aj voľne dostupné hotové profilovacie nástroje.

Práca odhalila technické pozadie procesu profilovania tak, aby bolo možné pochopiť, aké vplyv na beh aplikácií prináša aktivácia profilovacích mechanizmov a ako sa profilovacie techniky postupom času vyvíjali spolu s vývojom Java virtuálnych strojov. Popisuje profilovaciu architektúru v Java špecifikácii, ktorá umožňuje implementovať vlastné sofistikované profilovacie nástroje. Uvedená je súvislosť s aspektovo-orientovaným programovaním, ktoré na druhej strane efektívnym a ľahkým spôsobom umožňuje implementovať jednoduché profilovacie rozšírenia vyvíjanej aplikácie.

Nakoniec sa práca venuje začlenením profilovania do procesa vývoja a údržby softvéru, odpovedá na otázky kedy a ako použiť proces profilovania. Zahŕňa postrehy autora a iných, ktorí použili proces profilovania pri reálnom riešení implementačných a výkonnostných problémov, ktoré sprevádza vývoj počítačového softvéru. Po prečítaní práce by mal získať čitateľ dostatočný prehľad o problematike a mala by mu byť uľahčená adopcia princípov procesu profilovania.

8 Zoznam použitej literatúry

[1] Elektronický lexikón slovenského jazyka
http://www.slex.sk

[2] Merriam-Webster Online Dictionary
http://www.m-w.com/dictionary/profile

[3] Robert Bernecky
Profiling, performance, and perfection (tutorial session)
1989

[4] The Java Virtual Machine Specification
http://java.sun.com/docs/books/vmspec/

[5] Sun Microelectronics - picoJava Microprocessor Core Overview
http://www.sun.com/microelectronics/picoJava/overview.html

[6] The Java Language Specification
http://java.sun.com/docs/books/jls/

[7] Java Compilers
http://www.cs.cmu.edu/~jch/java/compilers.html

[8] Con Tran, Pierre N. Robillard
Teaching structured assembler programming
1985

[9] Glenn Ammons, Thomas Ball, James R. Larus
Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling
1997

[10] Software Development in the UNIX Environment - Profiling Your Code
http://www.dartmouth.edu/~rc/classes/soft_dev/profiling.html

[11] Susan L. Graham, Peter B. Kessler, Marshall K. McKusick
Gprof: a Call Execution Profiler
2003

[12] The Java Tutorial - Thread Scheduling
http://java.sun.com/docs/books/tutorial/essential/threads/priority.html

[13] Performance Documentation for the Java HotSpot VM – Threading
http://java.sun.com/docs/hotspot/threads/threads.html

[14] The Java Tutorial - Synchronizing Threads
http://java.sun.com/docs/books/tutorial/essential/threads/multithreaded.html

[15] Visualgc - Visual Garbage Collection Monitoring Tool
http://java.sun.com/performance/jvmstat/visualgc.html

[16] Brian A. Malloy, James F. Power
Using a Molecular Metaphor to Facilitate Comprehension of 3D Object Diagrams
2004

[17] Java(TM) Virtual Machine Profiler Interface (JVMPI)
http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html

[18] Sheng Liang, Deepa Viswanathan
Comprehensive Profiling Support in the Java Virtual Machine
1998

[19] Alan Durham, Edson Sussumu, Arlindo Flávio da Conceição
A framework for building language interpreters
2003

[20] The Java HotSpot Technology
http://java.sun.com/products/hotspot/

[21] JSR 163 - Java Platform Profiling Architecture
http://www.jcp.org/en/jsr/detail?id=163

[22] JVM Tool Interface
http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html

[23] ANTLR - Another Tool for Language Recognition
http://www.antlr.org/

[24] Java Compiler Compiler - The Java Parser Generator
https://javacc.dev.java.net/

[25] Java(tm) Source Code Instrumentation
http://www.glenmccl.com/instr/instr.htm

[26] Thomas Ball, James R.Larus
Optimally profiling and tracing programs
1992

[27] Misha Dmitriev - Better Profiling through Code Hotswapping:
http://java.sun.com/developer/technicalArticles/Interviews/Dmitriev_qa.html?feed=JSC

[28] Massimo Ancona, Walter Cazzola
Implementing the essence of reflection: a reflective run-time environment
2004

[29] Sheng Liang, Gilad Bracha
Dynamic Class Loading in the Java Virtual Machine,
1998

[30] Aspect-Oriented Software Development Community & Conference
http://aosd.net/

[31] Amitabh Srivastava, Alan Eustace
ATOM - A System for Building Customized Program Analysis Tools
1994

[32] HPROF: A Heap/CPU Profiling Tool in J2SE 5.0
http://java.sun.com/developer/technicalArticles/Programming/HPROF.html

[33] Using JConsole to Monitor Applications
http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html

[34] jmap - Memory Map
http://java.sun.com/j2se/1.5.0/docs/tooldocs/share/jmap.html

[35] HAT - The Java Heap Analysis Tool
https://hat.dev.java.net/

[36] YourKit profiler
http://www.yourkit.com/

[37] ej-technologies Jprofiler
http://www.ej-technologies.com/products/jprofiler/overview.html

[38] JIP - The Java Interactive Profiler
http://jiprof.sourceforge.net/