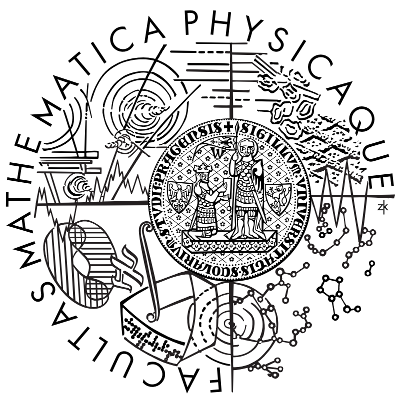
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Peter Morong
Locating Performance Regressions In Code
Lokalizovanie zmien výkonnosti v kóde
Katedra softwarového inženýrství
Vedoucí: Doc. Ing. Petr Tůma, Dr.
Studijní program: Informatika

Ďakujem svojmu vedúcemu diplomovej práce Doc. Ing. Petrovi Tůmovi, Dr. za kopec podnetných nápadov pri štúdiu vlastností a fungovania merania výkonnosti a za veľa cenných rad pri písaní tejto práce.

Čestne prehlasujem, že som svoju diplomovú prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím s požičiavaním práce.

V Prahe dňa 10.12.2007 Peter Morong

Abstrakt

Název práce: Lokalizovanie zmien výkonnosti v kóde
Autor: Peter Morong
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: Doc. Ing. Petr Tůma, Dr.
e-mail vedoucího: Petr.Tuma@mff.cuni.cz
Abstrakt: Cieľom tejto práce je vylepšiť hľadanie zmien v zdrojovom kóde, ktoré môžu spôsobovať zmenu výkonnosti a uľahčiť tak testovanie software. Pre tento účel je vytvorený framework, ktorý obsahuje popis prípadne vytvorenie nástrojov a definíciu postupov ako s nimi pracovať. Práca začína s popisom profilovania a kým spôsobom môže tento proces ovplyvniť výsledky merania výkonnosti. Sú definované požiadavky na profiler potrebný na túto prácu a na ich základe bol zvolený OProfile ako vzorový profiler. Ďalšia časť obsahuje analýzu behu programu popis jeho dvoch častí Execution a Waiting. Nasleduje rozdelenie zmien v zdrojovóm kóde do jednotlivých kategórií podľa toho, akým spôsobom je možné ich detekovať. Tretia časť definuje dve metódy na lokalizovanie zmien vo výkonnosti: filtrovanie zmien v kóde a porovnávanie výsledkov profileru. Filtrovanie zmien v kóde je definované ako prienik zoznamu zmien v zdrojovom kóde a zoznamu riadkov kódu, ktoré boli pri meraní spustené. Druhá metóda je len porovnávanie výstupov z profileru s použitím vizualizačného nástroja. Záverečná časť je ukážka použitia týchto metód v realnych projektoch.
Klíčová slova: benchmarkovanie, výkonnosť, profilovanie, regresia

Title: Locating Performance Regressions In Code
Author: Peter Morong
Department: Department of Software Engineering
Supervisor: Doc. Ing. Petr Tůma, Dr.
Supervisor's e-mail address: Petr.Tuma@mff.cuni.cz
Abstract: The objective of this work is to improve look up for changes in source code performance and help to remove burden at software testing with it. It tries to design some framework for this purpose which includes creating or desribing tools and defining methods how to work with them. The work starts with desciption of profiling and how this process can influence the performance measurement results. The profiler requirements are defined for purposes of this work and the OProfile is selected as the representative profiler. The next part contains analysis of program run and description of Execution and Waiting part of the run. Following is the categorization of various source code changes due to the posibility of their detection. The third part defines two methods for locating performance regression: advanced difference filtering and comparing the profiler results. Advanced difference filtering is an intersection between a list of changed code parts and a list of code parts executed at measurement. The second method is just comparing the profiler output using the visualization tool. The final part is demonstration of these two methods on real projects.
Keywords: benchmarking, performance, profiling, regression

Obsah

1 Úvod **4**

1.1 Benchmarking počas vývoja systému 4

1.2 Štatistický model nie je postačujúci 5

1.3 Cieľ práce 6

2 Profilovanie **7**

3 Hľadanie zmien výkonu v zdrojovom kóde **9**

3.1 Beh programu 9

3.2 Analýza zmien v zdrojovom kóde 12

3.3 Metódy na lokalizáciu zmien výkonnosti 13

3.3.1 Filtrovanie rozdielov kódu 14

3.3.2 Porovnávanie výsledkov profileru 15

3.3.3 Návrh vizualizačného nástroja 16

4 Data Visualization Tool **18**

4.1 Architektúra nástroja 18

4.2 Prototypná implementácia 20

4.3 Špecifikácia vstupných a výstupných dát 22

4.4 Zdrojové kódy 24

5 Aplikovanie metód na reálne projekty **25**

5.1 MONO 25

5.2 omniORB 26

5.2.1 Hľadanie vhodného profileru – Oprofile 26

5.2.2 Filtrovanie rozdielov kódu v omniORBe 28

5.2.3 Porovnávanie výsledkov profileru omniORBe 30

5.3 TAO 31

5.3.1 Filtrovanie rozdielov kódu v TAO CORBE 31

5.3.2 Porovnávanie výsledkov profileru v TAO CORBE 34

5.4 Java projekty - JORAM, ProActive 35

5.4.1 Hľadanie vhodného profileru 36

5.4.2 Filtrovanie rozdielov kódu – JORAM 37

6 Záver **39**

7 Zdroje **41**

7.1 Ostatné zdroje 41

8 Appendix A. **43**

8.1 Porovnanie verzií 20050522-123900 a 20050624-154300 43

8.2 Porovanie verzií 20050713-171045 a 20050718-080035 43

8.3 Porovanie verzií 20051013-175104 a 20051014-080512 49

1 Úvod

Výraz benchmarking popisujúci proces merania a následného porovnania výsledkov bol pôvodne používaný zememeračmi, ktorý si takto označili pevný bod z ktorého boli vykonané všetky ostatné merania. V 70tych rokoch nadobudol širší význam. V súvislosti napríklad s organizáciami, benchmarking je proces, ktorý identifikuje najlepšie postupy a snaží sa aplikovať ich aj v iných organizáciach. V oblasti vývoja softwaru má tento proces podobný význam. Jeho hlavným cieľom je zvýšiť výkon systému a zefektívniť beh programu.

1.1 Benchmarking počas vývoja systému

Testovanie systému je veľmi dôležitou súčasťou vývojového cyklu. V dnešnej dobe v podstate nie je možné vyvinúť software bez toho aby neprešiel nejakým procesom testovania. Rýchlo rastúce systémy tvorené mnohými nezávislými modulmi, rozsiahle tímy ľudí, ktorí sa podieľajú na vývoji, rovnako aj samotný vývoj, ktorého hlavnou prioritou býva často ušetriť peniaze a čas – to sú hlavné faktory, ktoré zvyšujú pravdepodobnosť vzniku chýb. Preto vo vývoji hraje testovanie systému takú dôležitú úlohu.

Existuje viac spôsobov ako je možné testovať systém. Netestuje sa pritom len správna funkcionalita, ale aj výkon. Dokonca testovaniu výkonu môže mať vyššiu prioritu, pretože čas, ktorý potrebuje systém na vygenerovanie odpovede bude kľúčový a pokiaľ systém nestihne odpovedať v tomto časovom úseku, tak potom sa odpoveď stáva zbytočnou. Testovanie výkonu je vlastne meranie rýchlosti akou je systém schopný vykonať konkrétnu úlohu. Cieľom týchto meraní nie je len zistiť čas kedy dostaneme odpoveď, ale hlavne vytvoriť možnosť na porovnanie výkonu dvoch rôznych systémov s rovnakou funkcionalitou. Rovnako je možné porovnávať dve rôzne verzie toho istého systému. Opakovaný benchmarking, ktorý spočíva v pravidelnom meraní rôznych verzií jedného systému je hlavnou témou tejto práce.

Konkrétne sa táto práca zameriava na to, akým spôsobom môžu zmeny v zdrojovom kóde ovplyvniť výkon. Na to je práve potrebné meranie jednotlivých verzií systému počas jeho vývoja. Takýto proces vyžaduje prostredie, ktoré bude dostatočne robustné aby zvládlo stiahnutie, skompilovanie systému a spustenie jednotlivých meraní. Skompilovanie a hlavne správny beh môže byť problematický pri systéme, ktorý je vo vývoji. Preto som sa rozhodol v rámci spolupráce s DSRG (Distributed Systems Research Group) využiť ich prostredie. Druhou časťou procesu je vyhodnotenie výsledkov testu, ktoré je výrazne komplikovanejšie ako vyhodnotenie funkčných testov. Výsledok totiž vyzerá ako množina nejakých náhodných čísel (časov) a je ťažké na prvý pohľad určiť ako veľmi sú presné a ako veľmi sú ovplyvnené napríklad operačným systémom. Všeobecne sa vie, že pri testovaní výkonu vzniká určitá warm-up fáza, kedy na začiatku testu sú výsledky výrazne ovplyvnené prípravou prostredia, v ktorom bežia. Preto výsledky meraní z

tejto fázy nie sú zahrnuté do celkových výsledkov. Zatiaľ neexistuje postup, podľa ktorého by bolo možné jednoznačne spočítať dĺžku warm-up fázy, takže býva nastavená manuálne na základe predošlých výsledkov.

Porovnávanie výsledkov, ktoré vyzerajú ako náhodné čísla vyžaduje určitú znalosť štatistiky. Najdôležitejšie je pri rozdielnych výsledkoch zistiť, či sa skutočne jedná o zmenu výkonu, alebo rozdielnosť je spôsobená vonkajším vplyvom. Jedno meranie pozostáva z viacerých behov benchmariek, kde jeden beh benchmarku tvorí opakované volanie jednej operácie systému a meranie času pre každé takéto meranie.

Pri vytváraní modelu som vychádzal z rovnakej myšlienky ako DSRG. Táto myšlienka je založená na predpoklade, že merania sú nezávislé a rovnomerne rozdelené v jednom behu a jednotlivé behy sú tiež nezávislé a rovnomerne rozdelené náhodné veličiny. Výsledok je potom priemer priemerov meraní zo všetkých behov. Popis ako je možné spočítať odhadovaný výsledok, je zverejnený v článku [1].

1.2 Štatistický model nie je postačujúci

Štatistický model vyzerá dobre pre vyčíslenie výsledku benchmarkingu a môže ušetriť čas potrebný na vykonanie všetkých meraní. Ale tento model nie je postačujúci pre vývojárov. Jeho nedostatkom je všeobecná neplatnosť predpokladu, na ktorom je postavený. Tento tvrdí, že merania sú nezávislé a rovnomerne rozdelené. Pri skúmaní výsledkov jednotlivých meraní je vidieť, že vznikajú vzorky dát, ktoré nie sú náhodné. Rovnako neplatí rovnomerne rodelenie, pretože vo výsledkoch je vidieť dáta, ktoré ležia výrazne mimo ostatné výsledky. Takéto odlišnosti vznikajú hlavne vonkajšími vplyvmi na meraný systém. Preto pri spracovávaní dát je potrebné ich analyzovať a podobné anomálie nebrať do úvahy.

Aj keby sa nepozeralo na tieto próblemy a štatistický model by fungoval bez akýchkoľvek výhrad, vývojárom by mohol slúžiť iba čiastočne. Je možné zistiť, že pri vývoji systému došlo k zmene jeho výkonu. Pokiaľ je tá zmena negatívna, je potrebné zistiť, čo ju spôsobilo. V tomto bode už štatistický model nijak neposlúži a je potrebné nájsť metódy, ktoré zjednodušia postup hľadania zmeny výkonu.

Teda máme 2 verzie systému, ktoré sú do určitej miery rozdielne čo sa zdrojových kódov týka a na základe meraní bolo zistené, že jedna verzia je rýchlejšia ako druhá. Vzniká otázka, čo spôsobuje rozdielnu dobu behu programov. Je zrejmé že takéto správanie bude spôsobené práve rozdielnosťou oboch verzií a teda konkrétnu príčinu je potrebné hľadať v častiach systému, ktoré nie sú rovnaké. Čím výraznejšie by boli rozdiely medzi oboma verziami, tým ťažšie sa bude hľadať príčina rozdielnej rýchlosti behu. Preto sa používa pravidelný test výkonu, aby sa častým testovaním minimalizoval rozsah zmien medzi porovnávanými verziami. Napriek tomu sa môže jednať o obsiahlejšiu časť kódu a preto je potrebné tento proces zefektívniť.

1.3 Cieľ práce

Vývojári, ktorých môžeme považovať za užívateľov benchmarkingu a prostredia, v ktorom sa merania automaticky vykonávajú, potrebujú mať k dispozícii efektivný nástroj na testovanie výkonu. Hlavným cieľom týtchto testov je zistiť, či zmeny vykonané v meranom systéme majú vplyv na jeho celkový výkon. Pokiaľ áno, je potrebné analyzovať zmeny zdrojového kódu, ktoré spôsobili zmenu výkonnosti a prípadne opraviť nevhodne zmenený kód. Ide o bežné zmeny, ktoré je možné vykonávať na zdrojovom kóde ako pridávanie, mazanie a menenie riadkov. Problém vzniká na projektoch, na ktorých pracuje veľké možstvo ľudí. Zmeny medzi dvoma verziami obsahujú príliš veľa zdrojového kódu a manuálne hľadanie príčiny zmeny výkonu by zabralo príliš veľa času.

Cieľom tejto práce je zjednodušiť tento proces. Navrhnúť nejaký framework, ktorý by obsahoval nástroje potrebné na vykonanie meraní a postupy, ktoré by sa aplikovali pri analýze zmeny výkonu a ukázali vývojarom kritické miesta v kóde.

Najprv začneme skúmaním behu programu a jeho vlastností. Na základe týchto vlastností budú odvodené určité metódy a postupy, ktoré sa dajú použiť za daných okolností. V druhej časti bude overovanie do akej miery je tento teoretický model použiteľný na reálnych projektoch.

2 Profilovanie

„*Profilovanie v počítačovej terminológií je analýza bežiaceho počítačového programu za účelom získania jeho skutočného a nie predpokladaného správania*“ [2]. Hlavným cieľom profilovania je zbieranie informácií o bežiacom programe a určovanie charakteristiky behu. Jedná sa o zaznamenávanie udalostí počas behu programu. Tieto udalosti delíme na hardwarový typ (hw prerušenia, systémové hodiny) alebo softwarový typ (sledovanie spúšťania funkcií, alokácie pamäte).

Treba počítať s tým, že profilovací mechanizmus má veľký vplyv na beh samotnej aplikácie. Jedná sa predovšetkým o spomalenie behu, ktoré je závislé od granularity zbieraných údajov.

Spomalenie v behu programu spôsobujú profilovacie funkcie, čo sú vlastne inštrukcie vrátane systemových volaní spúšťané nad rámec behu samotného programu. Pozastavujú jeho beh a zaznamenávajú potrebné údaje. Spôsobujú tým nielen spomalenie v behu ale aj tzv. cache pollution (zaznamenávanie informácií do pamäte, ktoré by sa tam pri bežnom spustení nevyskytovali), ktorá môže ovplyvniť správanie sa cache a memory.

V projektoch, ktoré obsahuje táto práca je možné rozlíšit 2 základné typy profilovania podľa algoritmu akým zbierajú data:

- Event Based Profilers – založené na zaznamenávaní udalostí programu
- Statistical Profilers – založené na pravidelnom vzorkovaní programu

Event Based Profiler zaznamenáva rôzne udalosti, ktoré vznikajú pri behu programu ako napr. function enter/leave, class load/unload, thread create/destroy, atď. Jeho fungovanie je založené na fakte, že má pod kontrolou celý beh programu a je presne informovaný o daných udalostiach. Jedná sa typicky o prípad, že program je interpretovaný, teda beží nad nejakou virtual machine a profiler pri zbieraní údajov spolupracuje práve s touto virtual machine. Príklady konkrétnych projektov sú MONO a projekty v Jave. Pozitívum tejto metódy je, že spomínané udalosti dokáže pomerne presne zaznamenať. Na druhej strane spôsobuje výraznejšie spomalenie, ktoré je navyše nerovnomerne rozložené počas behu programu.

Statistical Profiler je založený na pravidelnom vzorkovaní bežiaceho programu. Ako vonkajšia entita v pravidelných intervaloch zastavuje beh programu a zaznamenáva, kde sa momentálne program nachádza vo svojom behu. Nedokáže zaznamenať presný počet volaní danej funkcie, ale dokáže štatisticky určiť, koľko času z celého behu strávil program v danej funkcii. Výhodou tohto profilovania je jeho rovnomerne rozložená a zároveň nízka réžia.

V niektorých prípadoch je dokonca možné, že profiler bude mať vplyv na výslednú funkčnosť profilovanej aplikácie. Z pohľadu benchmarkingu preto nie je možné priamo porovnávať časy behov aplikácie získané s použitím a bez použitia profileru. Ďalšie riziko použitia profileru je iná lokalizácia zmeny výkonu. To znamená, že niektoré časti kódu môžu bežať relatívne rýchlejšie, prípadne pomalšie vzhľadom k ostatným častiam kódu ako v prípade bez použitia profileru. Takéto

anomálie spôsobuju, že aplikácia sa javí neefektívna v iných častiach zdrojového kódu ako v skutočnosti je a zároveň je potrebné si uvedomiť riziko, ktoré vzniká pri použití metód na hľadanie zmien výkonu prezentovaných v tejto práci.

Takéto výrazne zmenenie výsledkov profilingu oproti realite je možné ukázať na nasledujúcom príklade: Je program, ktorý má dva procesy. V jednom procese beží kód, ktorý v nejakom čase odomkne zámok. Druhý proces zatiaľ čaká na ten zámok spôsobom, že 100 krát sa opýta, či je odomknutý a keď nie, tak sa uspí na 10 sekúnd. Pri behu bez profilovania prvý proces odomkne zámok v momente, keď sa druhý proces dotazuje 95. krát a teda druhý proces beží ďalej bez uspatia. So zapnutým profilingom nastane situácia, že profiler pozastaví beh prvého procesu, aby zaznamenal potrebné údaje a ten nestihne odomknúť daný zámok pred 100. dotazom druhého procesu a teda druhý proces je uspaný, čo spôsobí výrazné spomalenie. Tento príklad je umelo vytvorený a predpokladá sa, že podobná situácia nastane len výnimočne. Napriek tomu je potrebné myslieť na dané riziká pri používaní profileru.

To sú v podstate všeobecné poznatky o profileroch a ich vplyve na meranie výkonnosti. Pre potreby tejto práce bolo nutné nájsť profiler, ktorý by spĺňal nasledujúce požiadavky:

i) Dostatočne všeobecný, aby bol použiteľný na viac projektoch
ii) Bol schopný vraciať informácie o behu na úrovni riadkov zdrojového kódu
iii) Na základe jeho výstupu a príslušných debugovacích informácií bolo možné určiť umiestnenie konkrétneho riadku v konkrétnom súbore vo filesystéme
iv) Profiloval nielen hlavný program ale aj potrebné zlinkované knižnice
v) Nespomaľoval výrazne beh systému a pokiaľ možno, aby bola jeho réžia rovnomerne rozložená v celom behu

Hneď sa ukázalo že bod i) nie je riešiteľný úplne všeobecne a preto bolo nutné projekty kategorizovať podľa toho v akom prostredí sú implementované a snažiť sa nájsť vhodný profiler pre každú množinu. Nakoniec aj tak existujú proprietárne projekty, ktoré musia mať svoj vlastný profiler, prípadne aspoň implementovaný určitý interface pre použitie všeobecnejšieho profileru (Mono). Z tohto hľadiska sa projekty, ktoré boli použité rozdelili do 3 skupín:

- C++ projekty
- Java projekty
- Mono

Konkrétne profilery, ktoré boli otestované sú popísané pri testovaní reálnych projektov v druhej časti tejto práce. Zatiaľ budeme predpokladať, že máme ideálny profiler, ktorý spĺňa vyššie definované predpoklady.

3 Hľadanie zmien výkonu v zdrojovom kóde

3.1 Beh programu

Program môžeme definovat ako jednoduchý zoznam inštrukcií. Konkrétny beh programu je potom postupnosť týchto inštrukcií ako boli vykonávané v čase. Z hľadiska času, ktorý je pre nás zaujímavý je to postupnosť časových úsekov, kde každý časový úsek reprezentuje vykonávanie určitej inštrukcie. Prípadne tento úsek reprezentuje čas, kedy sa vo vykonávaní kódu čaká na výsledok nejakej operácie potrebný pre pokračovanie v behu.

Za čakanie považujeme pozastavenie behu programu, pokiaľ nebudú pripravené systémové zdroje potrebné pre jeho ďalší beh, prípadne kým nedostane signál z druhého programu bežiaceho v inom procese alebo vlákne, že môže pokračovať vo svojom behu.

Vyššie programovacie jazyky členia program na konkrétne menšie časti kódu, ktoré sú reprezentované procedúrami. Jednotlivé príkazy sú rozdelené do procedúr, ktoré sa môžu navzájom volať. Objektové orientované jazyky ako napr. Java sú špeciálnym prípadom procedurálneho prístupu. Každé volanie procedúry je spojené s alokáciou sýstémových zdrojov – pamäte. Preto tento typ čakania sa vyskytuje s určitou pravidelnosťou vo všekých programoch.

Výslednou výkonnosťou sa nazve suma časových úsekov, ktoré musia byť spustené, aby bola vykonaná daná funkcia. Pri skúmaní výkonnosti systému sa vytvorí sada programov nazývaných benchmarky, ktoré otestujú rýchlosť výkonávania konkrétnych funkcií v danom systéme. Zmena výkonnosti je spôsobená zmenou dĺžky niektorých časových úsekov alebo pridaním (odobraním) nových časových úsekov.

Jednotlivé časové úseky je možné kategorizovať do dvoch skupín ako už bolo spomínané vyššie:

- Execution – vykonávanie postupnosti inštrukcií, tak ako nasledujú za sebou v behu programu s tým, že každá inštrukcia tvorí jeden časový úsek. Jedná sa iba o inštrukcie, ktoré sú priamo súčasťou programu. Je možné ich sledovať s použitím profileru.
- Waiting – predstavuje časové úseky, kedy program čaká. Tieto časové úseky vznikajú procesmi v prostredí, v ktorom program beží, ale nie sú súčasťou meraného systému. Muselo by byť profilované celé prostredie, kde systém beží, aby bolo možné vysvetliť, čo konkrétne vplýva na beh meraného systému.

Pre nás je zaujímavé akým spôsobom môže byť zmenená dĺžka behu programu, teda čo spôsobí, že sa vyskytne viac časových intervalov pri behu programu alebo predĺženie už existujúcich intervalov. Je viac možností ako takéto niečo môže nastať. Predpokladajme, že meraný systém nie je operačný systém, takže na svoj beh potrebuje nejaký operačný systém. Ďalší predpoklad je, že hardware na ktorom je systém meraný sa bude správať konštantne. Teda pri každom spustení bude

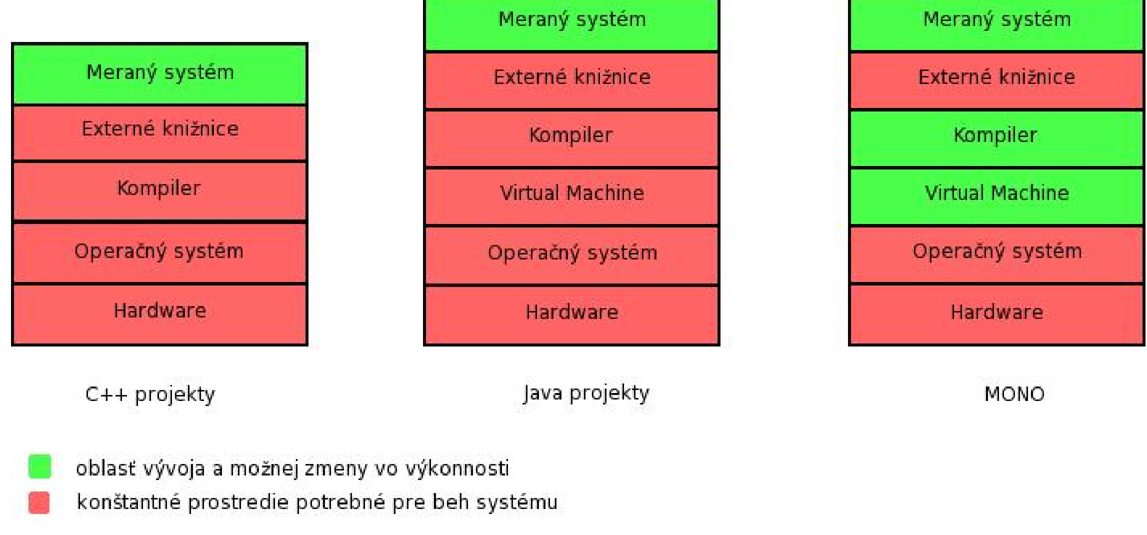
fungovať rovnako rýchlo. Za týchto podmienok je možné rozložiť dôvody zmeny výkonnosti medzi 4 entity na software úrovni:

- operačný systém – je potrebné, aby merania, ktoré sú navzájom porovnávané bežali na rovnakej verzii operačného systému. Inak by veľká časť zmeny v rýchlosti mohla byť pripísaná práve operačnému systému. Aj keď sa operačný systém nijak nemení, ovplyvňuje výrazne beh aplikácie a pri každom spustení vznikajú kvôli nemu odlišné výsledky. Na tento problém zatiaľ nie je žiadne riešenie, takže sa merania spustia viackrát a vytvorí sa priemerný výsledok
- virtual machine – pokiaľ je potrebná k behu programu, tak existujú dve možnosti ako môže byť vo vzťahu s meraným systémom:
  i) nezávislá na vývoji – nie je vyvíjaná spolu so systémom, je iba potrebná na jeho beh. V tomto prípade je jej vplyv podobný ako vplyv operačného systému a požaduje sa, aby merania prebehli na rovnakej verzii virtual machine.
  ii) závislá na vývoji – je potrebné vnímať ju ako časť systému a čas strávený na vykonávaní jej inštrukcií sa berie ako čas strávaný pri behu samotného systému
- compiler – môže ovplyvňovať beh systému dvoma spôsobmi podobne ako virtual machine, teda je to nezávislá entita, alebo je to časť systému vyvíjaná spolu s ním. Keď je to nezávislá entita, tak je postačujúce, že kompilácia prebehne za rovnakých podmienok a rovnakou verziou compileru. Pokiaľ je compiler súčasťou vyvíjaného systému, potom je to trochu problém, pretože môže výrazne zmeniť rýchlosť aplikácie, ale pritom sám nie je spúšťaný, takže do behu programu nie je zahrnutý. Tento problém bude rozobraný podrobnejšie v kapitole Analýza zmien zdrojových kódov.
- external libraries – nie sú súčasťou vývoju systému, ale môžu byť výraznou súčasťou behu programu a profiler zaznamená čas strávený volaním inštrukcií práve z externých libraries
- meraný systém – samotný zdrojový kód, ktorého výkonnosť chceme zistiť. Cieľom je porovnať dve rozdielne verzie, takže okrem vyššie spomínaných vplyvov by mala byť rozdielnosť vo výkonnosti spôsobená práve rozdielnosťou verzií.

Toto rozdelenie definuje, ktorými časťami sa budeme zaoberať a ktoré zanedbáme aj keď vieme, že majú vplyv na výsledok. Pri skúmaní behu programu sa budeme pozerať na celok tvorený všetkými jeho časťami, ktoré sa priamo podielajú na jeho behu, teda meraným systémom, prípadne virtual machine, ostatné entity budú považované za koštantné.

Týmto je jasne definované, čo bude predmetom merania a čo bude považované za prostredie, v ktorom systém beží. Budeme predpokladať, že prostredie pôsobí na meraný systém rovnako a nebude merané, ani sa ďalej nebudú brať do úvahy jeho vlastnosti.

Zmeny v zdrojovom kóde, ktoré majú vplyv na dĺžku behu môžu byť rozdelené podľa toho aký typ časových úsekov ovplyvňujú. Boli definované dva typy časových úsekov Execution a Waiting. Nakoľko väčšina popísaných zmien môže mať vplyv



[Obr. 1] Merané a konštanté časti prostredia

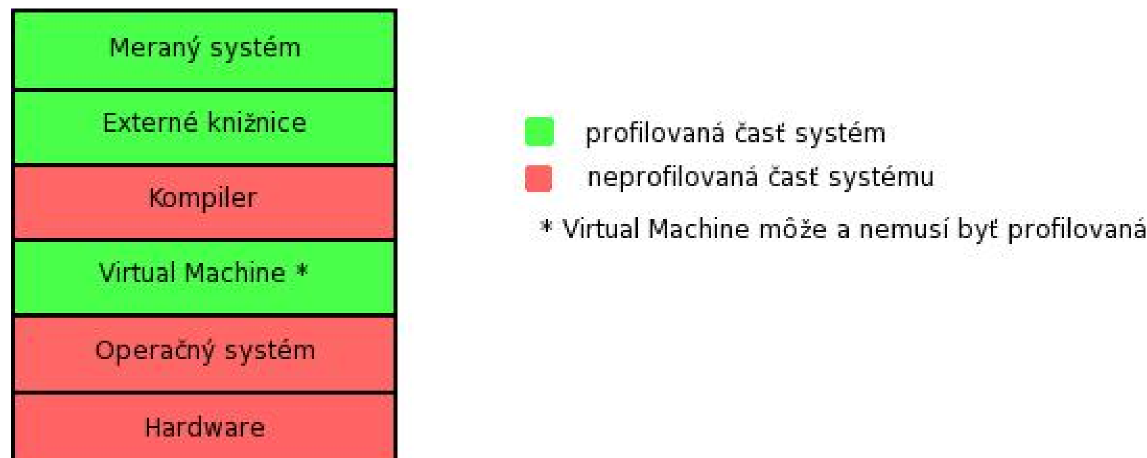
na oba typy, tak nasledujúce rozdelenie popisuje kam pravdepodobnejšie daná zmena zapadne:

i) Execution
   - Viac zdrojového kódu, viac iterácií – pridávanie inštrukcií znamená pridávanie časových úsekov potrebných na ich vykonanie k celkovému behu programu. Zrejme najčastejšia zmena, pokiaľ prebieha aktívný vývoj systému a pridáva sa nova funkcionalita. Rovnako opakovanie inštrukcie alebo celej množiny inštrukcií predĺži beh programu. Môže sa jednať o cykly prípadne opakované volanie funkcií.
   - Optimalizácia kódu, refactoring – zrýchlenie systému zrušením behu niektorých inštrukcií, ktoré nie sú nezbytne potrebné na dosiahnutie daného výsledku, funkcionalita rozdelená do viac funkcií s menším počtom riadkov, ako keď je koncentrovaná v jednej veľkej spôsobuje iný prístup k zdrojom.
   - Komplexnejšie datové štruktúry – menenie datových typov premenných a vytváranie zložitejších datových štruktúr ovplyvňuje prístup k systemovým zdrojom, hlavne k pamäti.

ii) Waiting
   - Náročnejšie systémové operácie – rôzne dôvody môžu spôsobiť, že prostredie, v ktorom systém beží zmení rýchlosť jeho behu (napr. práca s diskom, práca s procesmi...).

Táto kategorizácia nevymenováva všetky možné zmeny v zdrojovom kóde, ale skôr tie hlavné, ktoré sa vyskytujú najčastejšie. Podobne ako pri predchádzajúcom vymenovaní sw entít, ktoré ovplyvňujú beh systému aj teraz je snaha rozdeliť zmeny na tie, ktoré budú sledované a tie, ktoré síce ovplyvnia beh, ale budú zanedbané a prostredie považované za konštantné, pretože nie je cieľom merať celé prostredie.

Keď sa pozrieme na toto rozdelenie z pohľadu profilovania, ktoré nás v tomto momente zaujíma najviac, tak sa jedná o dve katogórie, ktoré je možné priamo zachytiť profilerom a ktoré nie je.



[Obr. 2] Kategorizácia častí systému podľa toho, či sú profilované

Výkonnosť je výrazným spósobom ovplyvnená aj efektom alokácie zdrojov pri spúšťaní programu. Tento efekt je spôsobený tým, že aktuálny stav zdrojov, ktorými disponuje operačný systém je pri každom spustení iný. Dôsledkom toho pri každom spustení vznikajú iné výsledky, preto je potrebné spustenia opakovať a vytvoriť priemer zo všetkých výsledkov. Až na tento efekt sú všetky zmeny výkonnosti spôsobené zmenami v zdrojom kóde alebo nastaveniami pri kompilácií a spustení.

3.2 Analýza zmien v zdrojovom kóde

Vysvetlili sme si beh programu a čo všetko môže vplývať na výsledný čas. Definovali sme, čo je meraný systém a čo všetko môže byť jeho súčasťou. Snažíme sa vysvetliť, ktoré zo zmien v meranom systéme majú za následok zmenu v jeho výkonnosti. Už bolo povedané, že na výslednú výkonnosť vplýva veľa vonkajších okolností, ale tieto budú zanedbané a budeme predpokladať, že prostredie v ktorom beží meraný systém sa správa rovnako pri každom spustení.

Budeme sa zaoberať len zmenami v meranej aplikácii. Rôzne zmeny v zdrojovom kóde, ktoré majú vplyv na výkonnosť je možné rozdeliť do kategôrií podľa možnosti ich lokalizovania. Celkovo sú tri hlavné skupiny, do ktorých je možne začleniť všekty zmeny v kóde: Zmeny, ktoré spôsobia, že zmenený kód beží dlhšie – priamy vplyv, zmeny, ktoré spôsobia, že iný kód (nezmenený) beží dlhšie – nepriamy vplyv a zmeny v moduloch aplikácie, ktoré spôsobia dlhší beh kódu ale samy pritom vôbec nie sú spúšťané počas meraného behu – vonkajší vplyv. Pre každú skupinu skúsim vymenovať reálne príklady pre lepšiu ilustráciu:

i) Priamy vplyv – jedná sa o zmeny v zdrojovom kóde, kedy pri profilovaní je vyhodnotená zmena vo výkonnosti na rovnakom mieste ako zmena v kóde.
   - Pridávanie nových riadkov kódu – volanie nových inštrukcií spomalí beh na rovnakom mieste ako boli pridané, pretože sa tam vloží časový úsek potrebný na jej vykonanie
   - Vytváranie opakovaní volaní riadkov, zvyšovanie počtu opakovaní – podobne ako pri pridávaní inštrukcií sa vkladajú v mieste zmeny časové úseky, ktoré predstavujú čas potrebný na vykonanie opakovania inštrukcie alebo na vykonanie sady inštrukcií
   - Zložitejšie výpočtové operácie a práca s operátormi – optimalizácia výpočtu zmení dĺžku časového úseku potrebného na jeho vykonanie

ii) Nepriamy vplyv – profiler zaznamená zmenu vo výkonnosti na mieste, kde nedošlo k zmene kódu.
   - Zmena konštanty ovplyvňujúca napr. počet opakovaní – zmena konštanty, ktorá sa môže nachádzať v inom súbore so zdrojovým kódom ovplyvní všetky miesta v behu programu, kde sa vyskytuje. Najvýraznejšia je zmena na mieste, kde zmenená konštanta predstavuje počet opakovaní nejakého cyklu.
   - Zmena datových štruktúr – komplikovanejšie datové štruktúry ovplyvňujú beh programu na všetkých miestach, kde sa s nimi pracuje.
   - Zmena názvov, typov premenných, zmena návratových hodnôt funkcií – nie príliš výrazne zmeny vo výkonnosti, ale je dobré brať ich do úvahy

iii) Vonkajší vplyv – jedná sa o vplyv aplikácie, ktorá je súčasťou vyvíjaného systému, ale žiadna z jej častí nie je spustená počas merania. Typickým príkladom takejto aplikácie je kompiler. Zisťovať aký vplyv majú zmeny v kompilery na výkonnosť systému, ktorý je ním kompilovaný sa dá rôznymi spôsobmi. Napr:
   - mať sadu pomocných aplikácií a benchmariek, ktoré budu testovať konkrétnu vlastnosť kompileru. Pri snahe vylepšiť túto vlastnosť sa najprv skompilujú a zmerajú pomocné aplikácie a pokiaľ sa prejaví zmena vo výkonnosti podobného charakteru aj v celom meranom systéme, tak je možné ju takto zdôvodniť
   - iná možnosť je snažiť sa merať systém s konštantnou verziou kompileru. Takže pokiaľ sa zmení kompiler, tak novou verziou kompileru bude potrebné skompilovať staršiu aj novú verziu meraného systému a až potom porovnať ich výkonnosť.

3.3 Metódy na lokalizáciu zmien výkonnosti

Nasledujúca kapitola popisuje myšlienky a metódy, ktoré by mali uľahčiť hľadanie príčin zmeny výkonnosti systému. Do akej miery je možné tieto nové metódy použiť v reálnych projektov a s akou úspešnosťou je popísané v druhej časti tejto práce, zaoberajúcej sa použitím na konkrétnych projektoch.

Ako už bolo spomínané základom pre hľadanie zmien vo výkonnosti je množina všekých zmien v zdrojovom kóde, ktorú je možné jednoducho získať použitím nejakého programu, napr. utility diff v Linuxe. Samotné použitie diffu nemusí byť postačujúce, pretože môže reprezentovať celkom veľké množstvo zdrojového kódu zvlášť pokiaľ sa jedna o projekt, na ktorom sa podiela veľký počet ľudí.

3.3.1 Filtrovanie rozdielov kódu

V prvej fáze tejto práce bola snaha zlepšiť filtrovanie zmien. Vychádzal som z toho, že jeden konkrétny benchmark pri svojom behu netestuje celý systém, ale zameriava sa iba na časť systému potrebnú na vykonanie konkrétnej meranej funkcie. Takže vplyv na výsledok tohto merania budú mať iba také zmeny v zdrojovom kóde, ktoré sa dotýkajú tejto funkcionality.

Bolo potrebné zistiť, ktoré časti programu sa pri behu daného benchmarku spúšťajú. Na tento účel bol použitý mechanizmus profilovania, ktorý zobrazil funkcie a riadky zdrojového kódu, ktoré boli pri meraní spustené. Zoznam týchto spustených častí bol potom porovnaný so zmenami v kóde (výstup z diffu) pre dané časti programu. Takýmto postupom je možné presne lokalizovať zmeny z kategórie *Priamy vplyv*.

Nakoľko v tomto prípade použitie profileru nemá za úlohu merať časy v behu jednotlivých častí, je možné miesto profileru použiť iné nástroje, napr. tzv. Code Coverage nástroje. Tieto programy vo všeobecnosti slúžia na zistenie, ktoré riadky zdrojového kódu boli vykonané pri spustení skúmaného programu. A táto informácie je jediná potrebná pri použití tejto metódy.

Túto metódu som nazval Filtrovanie rozdielov a vo všeobecnosti sa jedná o vytvorenie prieniku medzi zoznamom zmenených častí kódu a zoznamom častí kódu spustených pri meraní. Je možné pomocou nej zachytiť zmeny spadajúce do kategórie Execution. Ale s istotou budú zachytené iba tie, ktoré môžeme zaradiť do skupiny *Priamy vplyv*.

Zmeny typu *Nepriamy vplyv* nemusia byť v určitých prípadoch zachytené. Jedná sa totiž o zmeny, ktoré sa vyskytujú na určitom mieste v zdrojovom kóde, ale spôsobujú zmenu vo výkonosti na inom mieste. Pokiaľ sa zobere v úvahu beh jedného benchmarku, potom vznikajú 3 možnosti zaujímavé z podľadu skúmania zmien typu *Nepriamy vplyv*:

- Zmena zdrojového kódu aj zmena výkonnosti leží v rámci skúmaného benchmarku – v tomto prípade sa zachytí zdroj zmien výkonnosti, pretože zmena zdrojového kódu leží v profilovanej časti, teda patrí do výsledného prieniku
- Zmena zdrojového kódu leží v rámci skúmaného benchmarku ale zmena výkonnosti leží mimo neho – v tomto prípade sa bude jednať o falošný poplach, pretože zmena kódu bude v profilovanej časti, ale dôsledok tejto zmeny z pohľadu výkonnosti nemá žiadny vplyv na výsledky merania
- Zmena výkonnosti leží v rámci skúmaného benchmarku ale zmena zdrojového kódu leží mimo neho – tu vzniká prípad, kedy nebude zachytený dôvod vi-

diteľných zmien výkonnosti, pretože tento dôvod leží mimo skúmanej časti kódu.

kde rámec skúmaného benchmarku je časť kódu, ktorá sa vykonáva pri behu benchmarku.

Z týchto troch spomínaných možností, ktoré môžu nastať sú hneď na prvý pohľad viditeľné nedostatky tejto metódy. Aj keď totiž označí nejaký blok kódu, že by mal mať za následok zmenu výkonnosti, nemusí to byť pravda. Takže sa jedná skôr o vyznačenie podozrivých miest v kóde.

Na druhej strane je možné jednoduchým pokusom overiť, či podozrivý kód skutočne spôsobuje zmenu vo výkonnosti. Z dvoch porovnávaných verzií systému sa vytvorí určitá medziverzia, teda niečo medzi nimi a to takým spôsobom, že sa zoberie kód novej verzie systému a časti podozrivé na zmenu výkonnosti sú nahradené z pôvodnej verzie. Spustí sa meranie na medziverzii a výskedky sú porovnané vzhľadom k obom porovnávaným verziám. Pokiaľ sú v priemere bližšie k starej verzii, potom môžeme považovať zmenu výkonnosti za lokalizovanú a pokiaľ podozrivá časť kódu nie je príliš rozsiahla, je možné ju dohľadať ručne.

Na záver je možné zhodnotiť hlavné plusy a mínusy tejto metódy.

Plusy:

i) Pomerne dobrá detekcia zmien typu *Priamy vplyv*
ii) Možnosť vytvoriť medziverziu pre overenie nájdených regresií

Mínusy:

i) Neschopnosť detekovať niektoré konkrétne typy zmien
ii) Obmedzenie na relatívne malú časť kódu z celého rozsahu merania (prienik diffu a profileru)

3.3.2 Porovnávanie výsledkov profileru

Prvá metóda na hľadanie zmien *Filtrovanie rozdielov* má limitované možnosti dané tým, že sa zameriava iba na časť kódu, ktorý sa spúšťa a nie na systém ako celok. Tým pádom má problém s lokalizovaním zmien typu *Nepriamy vplyv* a bolo potrebné zvoliť trochu iný pohľad a vymyslieť ďalšiu metódu, ktorá by pracovala s celým zdrojovým kódom naraz.

Druhá metóda je teda založená na inej myšlienke: Pri porovnávaní dvoch verzií systému nie je až tak dôležitý výsledný čas behu ako skôr rozdielnosť medzi nimi. A pokiaľ by bol čas ovplyvnený konštantou a rovnako pri oboch meraniach, tak by to vôbec nevadilo. Tento predpoklad som použil pri porovnávaní výsledkov ovplyvnených behom profileru. Dáva totiž možnosť porovnať beh zdrojových kódov riadok po riadku. Konkrétne je možné porovnať časy strávené na jednotlivých riadkoch kódu.

Aby sa mohli objektívne porovnávať výsledky z profileru, bolo potrebné použiť profiler ktorý bude mať nízku réžiu potrebnú na svoj beh a navyše táto bude

rovnomerne rozložená počas celého behu systému. Presne ako to je popísané v poslednom bode požiadaviek na profiler, definovaných v kapitole o profileri.

Metódu porovnávania výsledkov profileru je možné použiť na detekciu oboch typov zmien v zdrojovom kóde: *priamy vplyv* aj *nepriamy vplyv*, rovnako detekuje aj jednu kategóriu zmien z pohľadu behu programu: Execution. Mohlo by sa zdať, že táto metóda je ideálna, keď dokáže detekovať všetky typy zmien a navyše zahŕňa celý zdrojový kód. Mimo nej ležia len zmeny v skriptoch pre kompiláciu.

Nevýhody metódy sa dajú definovať v dvoch hlavných bodoch:

i) Prvá nevýhoda vychádza čiastočne z popisu fungovania profileru. A teda je tu určité riziko, že profiler vyhodnotí zmenu výkonnosti na nesprávnom mieste v zdrojovom kóde.
ii) Druhá nevýhoda spočíva v tom, že nie je možné overiť správnosť nájdených zmien pomocou vytvorenia medziverzie. Pretože profiler ukazuje často kľúčové zmeny na miestach, kde nedošlo k zmene v zdrojovóm kóde, takže medziverziu nie je z čoho vytvoriť.

Výhody metódy:

i) Zachytáva všetky druhy zmien.
ii) Pozerá sa na celý profilovaný kód, teda na celok, ktorý bol pri meraní spustený

Nakoľko pri použití tejto metódy nebolo možné vytvárať medziverzie na overovanie funkčnosti, je to metóda vhodná skôr pre programátorov, prípadne architektov daného systému, ktorí ho dobre poznajú a dokážu na základe výsledkov tejto metódy a znalosti grafu volania funkcií zistiť príčinu zmien vo výkonnosti.

Porovnanie metód

Typy zmien:	Filtrovanie rozdielov kódu	Porovnávanie výsledkov profileru
Priamy vplyv	Áno	Áno
Nepriamy vplyv	Nie	Áno
Execution	Áno	Áno
Waiting	Nie	Nie
Použiteľnosť:		
Overenie medziverziou*	Skôr áno	Skôr nie

* záleží na konkrétnych prípadoch

[Obr. 3] Porovnanie vlastností oboch metód

3.3.3 Návrh vizualizačného nástroja

Pre uľahčenie práce ľuďom, ktorí sa snažia nájsť zmenu výkonnosti na základe výsledkov metódy Porovnávanie výsledkov profileru bol navrhnutý Data Visuali-

zation Tool (DVT). Jeho úlohou je zprocesovať výstupy z profilerov z oboch meraní, vzájomne ich medzi sebou porovnať a poskytnúť použiteľný výstup z tohto porovnania. Cieľom by mal byť výstup, ktorý bude ľudsky čitateľný, prípadne ľahko použiteľný ako vstup do iného systému.

Snahou je vytvoriť všeobecne použiteľný model DVT, teda aby bolo možné čítať výstup z ľuboľného profileru a zároveň výstup z DVT by bol vo formáte ľahko použiteľnom ako vstup pre iný systém.

Hlavným požiadavkom na DVT je schopnosť vygenerovať a zobraziť výsledky na rôznych úrovniach porovnania

i) Adresárov (jednotlivé adresáre sú entity tejto úrovne)
ii) Súborov (súbory jedného adresára (rodič) sú entity tejto úrovne)
iii) Riadkov zdrojového kódu (riadky kódu v jednom súbore (rodič) sú entity tejto úrovne)

teda po porovnaní by sa mala vytvoriť stromová štruktúra, ktorá by zhruba zodpovedala fyzickému uloženiu zdrojových kódov vo filesysteme. Na každej úrovni by mal byť uvedený počet vzoriek pri každej entite, ktorý by predstavoval sumu vzoriek všetkých potomkov. Takto by už na najvyššej úrovni bolo vidieť, v ktorých častiach má zdrojový kód najrozdielnejší výkon a presun po stromovej štruktúre až k poslednému synovi by bolo lokalizovaním kritického miesta v kóde. Na každej úrovni by bolo potrebné ešte spočítať *rozdielnosť* potomkov pre každú entitu. Pokiaľ sú totiž rodičia v dvoch rôznych verziách systému rovnakí, nemusia byť rovnakí aj ich potomkovia.

Pre vysvetlenie tohto požiadavku úvadzam príklad, kde sa porovnávajú 2 adresáre medzi sebou, kde na každom z nich zaznamenal profiler 100 vzoriek a oba adresáre obsahujú po dva súbory. Lenže rozdedelenie vzoriek medzi súbory je v prvom prípade 60 a 40 a v druhom prípade 90 a 10. Aj keď oba adresáre sa javia ako rovnaké z pohľadu celkovej výkonnosti, predsa je medzi nimi rozdiel a je potrebné na to užívateľa upozorniť.

Pre jednoduchší pohyb aj v rozsiahlejších systémoch je potrebné definovať 2 typy porovnania:

- Absolutné porovnanie
- Relatívne porovnanie

tieto dve porovnania sa použijú, pokiaľ výsledok bude zobrazovaný v grafickej podobe užívateľovi. Sprehľadnia grafy vyskytujúce sa pri každej entite, ktoré zobrazujú počty vzoriek pre obe verzie.

Absolutné porovnanie bude priame porovnanie dvoch počtov vzoriek z dvoch verzií systému iba pre jednu entitu, nezávislé od ostatných entít. Toto porovnanie zobrazí aký výrazný je rozdiel medzi verziami v tejto jednej entite.

Relatívne porovnanie porovnáva každý počet vzoriek každej entity vzhľadom k nejakém číslu (maximálnemu počtu vzoriek na jednej entite). Teda vyjadruje aký veľký vplyv majú entity na celkový systém.

4 Data Visualization Tool

Výsledky meraní nám vracajú rôzne čísla, ktoré by mali prezentovať výkonnosť meraného systému. Tieto čísla môžu byť nepoužiteľné, pokiaľ nie je žiadna možnosť ako by sa dali porovnať s výsledkami merania inej verzie systému, prípadne iného systému s rovnakou funkcionalitou. Preto proces merania a zbierania výsledkov musí byť nasledovaný procesom porovnávania. Väčšinou sa jedná práve o porovnanie celkovej výkonnosti určitej funkcie systému, na ktorú boli benchmarky zamerané, ale nie je možné z nich dostať podrobnejšie informácie. Snahou je samozrejme zbaviť výsledky, čo možno najviac vonkajších vplyvov. V tomto smere už existujú nástroje, ktoré používa DSRG, takže nebolo cieľom vyvíjať ďalšie.

Zmenil som trochu pohľad a priority pri meraní a pokúsil sa porovnať data, ktoré boli ovplyvnené profilovaním. Existuje riziko nesprávnych výsledkov, ktoré je popísané v kapitole o profilovaní, ale na druhej strane vzniká sada výsledkov, ktorá umožňuje lepšiu granularitu pri porovnávaní kódu. Nie je už potrebné pozerať sa na meraný kód ako na celok, ktorý potrebuje zmeraný čas na svoje vykonanie, ale je možné skúmať a porovnávať menšie časti systému ako napríklad súbory, či riadky kódu, medzi sebou. Tento pohľad môže uľahčiť hľadanie zmien výkonnosti v kóde, ako to popisuje metóda o porovnávaní výsledkov profilovania.

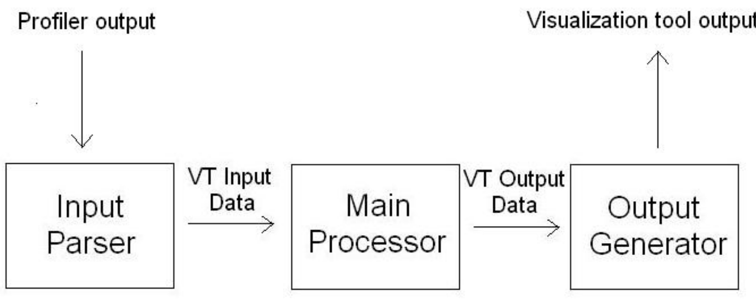
4.1 Architektúra nástroja

Snahou bolo navrhnúť všeobecný model, ktorý by bolo možné použiť na spracovanie výstupu z rôznych profilerov. Rovnaké nároky boli aj na výstup. Cieľom bolo, aby výstup nebol v proprietárnom formáte, ktorý sa síce bude dať jednoducho zobraziť, ale pre prípadne ďalšie výpočty alebo zakomponovanie do iného systému by bol nepoužiteľný.

Hlavným cieľom návrhu fungovania DVT bola schopnosť porovnať výstupy z profileru pri meraní výkonnosti dvoch rôznych verzií rovnakého systému. Detailnejšie požiadavky na tool plynú z popisu metódy porovnávanie výsledkov profileru.

Model DVT bol vytvorený spojením troch komponent: Input parser, Main processor, Output generator.

Každá komponenta funguje samostatne a je možné ju vymeniť za inú v prípade potreby spracovania iného formátu dát. Na prenos dát medzi komponentami je definovany presny formát dát postavený na XML. Pretože sa jedná o sekvenčné volanie komponent na určité data, tak formát dát je jediná požadovaná vlastnosť jednotlivých komponent. Ich vnútorné fungovanie rovnako aj jazyk, v ktorom sú implementované zostávajú na voľbe autora. Pri reálnom použití v tejto práci išlo predovšetkým o čítanie a spracovanie textových dát, preto bol pri vytváraní zvolený jazyk Perl.



[Obr. 4] Prepojenie komponent

Trochu si priblížime fungovanie a podstatu jednotlivých komponent systému v poradí ako sa volajú na konkrétne data.

Vstupom pre celý systém je výstup z profileru, konkrétne dva súbory, kde každý z nich obsahuje zdrojový kód jednej z profilovaných verzií. Prvá komponenta s názvom Input parser načíta tento vstup a upraví do XML formátu nazvaného VT Input Data, ktorý je podrobne definovaný v kapitole Špecifikácia vstupných a výstupných dát. Podstatou tejto komponenty je zjednotiť rôznorodosť výstupov ľubovoľných profilerov do jedného štandardu.

Hlavná komponenta nazvaná Main processor načíta VT Input Data, vykoná všetky výpočty a vygeneruje výstup v XML formáte – VT Output Data. Má dve hlavné časti preprocessing a processing.

Preprocessing spracuje dva vstupné XML súbory, kde každý obsahuje zdrojový kód jednej z verzií a vytvorí pre každú verziu jeden adresár. V každom adresári vytvorí rovnakú adresárovú štruktúru akú má zdrojový kód daného systému, ale všetky sôbory sú v textovom formáte rovnako ako zdrojový kód. Navyše vzniknú pomocné súbory ako napr. directorylist, ktoré obsahujú zoznamy profilovaných súborov a adresárov a sumy vzoriek pre jednotlivé entity tejto adresárovej štruktúry. Na každom riadku je počet vzoriek pre každú entitu a potom názov danej entity, teda súboru alebo adresáru. Tento názov je tvorený relatívnou cestou a pokiaľ ide o súbor, tak aj názvom daného súboru.

Komponenta Main processor pokračuje ďalej v spracovaní takto rozčlenených zdrojových kódov a to zlučovaním na úrovni súborov. Zoberie sa jeden súbor z každej z oboch verzií, použije sa na nich diff, výsledok sa rozsparsuje a takto spojený súbor sa uloží do XML formátu. Nový súbor obsahuje na každom riadku zdrojový kód plus informáciu o tom koľko vzoriek bolo na danom riadku zaznamenaných profilerom pri meraní oboch verzií. Pokiaľ prišlo na danom riadku k zmene, prípadne bol riadok pridaný, alebo odobraný, teda je rozdielny medzi oboma verziamu, potom je táto informácia uvedená v atribute type. Ďalším dôležitým atributom je difference, ktorý sa spočíta:

$\sum |v_n - v_o|$

kde vn a vo sú počty vzoriek profileru na jednotlivých riadkoch pre novú a starú verziu. Atribut difference má určiť do akej miery sú rozdielne výsledky z profilovania dvoch súborov aj keď sumy vzoriek sú veľmi podobné. Rovnako sa hodnoty z tohto atribútu sčítavajú a zobrazujú aj o úroveň vyššie pre adresáre. Táto akcia sa zopakuje postupne na všetky súbory a výsledok je definovaný ako VT Output data.

VT Output formát je navrhnutý spôsobom, aby bol jednoducho importovateľný do ľubovoľného systému. Poslednou komponentou pri spracovaní dát je Output Generator. Jeho úlohou je prispôsobiť dáta vo formáte VT Output do podoby požadovanej koncovým užívateľom. Je na ňom, či budú vykonané nejaké dodatočné výpočty, alebo sa bude jednať len o transformáciu dát.

4.2 Prototypná implementácia

Pre demonštráciu funkcionality DVT bola vytvorená posledná komponenta, ktorá vygeneruje HTML stránky, ktoré je možné priamo zobraziť v prehliadači bez ďalších potrebných úprav. Výsledná grafická podoba obsahuje zoznam entít – súbory a adresáre. Za ním sú dva stĺpce, ktoré uvádzajú počty vzoriek pri spustení pôvodnej aj novej verzie. Ďalší stĺpec zobrazuje percentuálne vyjadrenie rozdielnosti starej a novej verzie, pretože aj pri rovnakom počte vzoriek môžu byť tieto vzorky rozdistribuované rozdielne. Nakoniec zostal úplne prvý stĺpec, ktorý graficky zobrazuje ako veľmi sa líšia obe verzie medzi sebou. Do demonštrácie output generatora boli implementované aj dodatočné výpočty, aby bolo možné prepínať dve rôzne grafické zobrazenia rozdielnosti.

- absolútne porovnanie - porovnajú sa konkrétne počty vzoriek pre jednotlivé riadky a súbory
- relatívne porovnanie - porovnajú sa percentuálne vyjadrenia počtu vzoriek z celkového počtu pre danú entitu

Absolutné porovnanie je použité v prípade, že obe merania prebehli za totožných podmienok, teda rovnaký počet spustení rovnakých benchmariek, rovnaký počet opakovaní merania... Predpokladá sa, že pokiaľ nedošlo k zmene vo výkonnosti, tak aj počty vzoriek budú rovnaké. Grafické zobrazenie v tomto prípade porovnáva počty vzoriek na jednom riadku, teda pre jednu konkrétnu entitu. Je možné na prvý pohľad vidieť aký je medzi verziami rozdiel v tejto entite v počte vzoriek. Takéto zobrazenie ale nezachytáva vzťah k rodičovskej entite a teda ani veľkosť vplyvu na celkový výsledok.

Relatívne porovnanie sa použije v prípade že merania bežali za rôznych podmienok. Napríklad merania boli spustené dokedy nie je dostatočné množstvo dát a potom sú stopnuté - výsledkom sú rôzne počty vzoriek. Preto sa predpokladá, že užívateľ bude presne vedieť akým spôbom bol spustený benchmarking, s ktorého výsledkami pracuje. Grafické zobrazenie v tomto prípade porovnáva počty vzoriek

ako percentuálne vyjadrenia z počtu vzoriek rodičovskej entity. Teda je na prvý pohľad vidieť aký veľký vplyv má daná entita na celkový výsledok a rovnako aj rozdielnosť medzi oboma verziami. Pri výrazne nižších počtoch vzoriek je vplyv zanedbateľný a aj grafy zobrazujúce rozdielnosť sú v tomto prípade zanedbateľné.

Pri častom používaní bolo zistené, že pri vizualizácii dát je ideálne z praktického hľadiska kombinovať obe porovnania. Konkrétne použiť na každej úrovni iný typ porovnania. Na úrovni zobrazenia adresárov je lepšie použiť relatívne porovnanie, aby si užívateľ uvedomil akú váhu majú jednotlivé adresáre navzájom medzi sebou.

Relative (Absolute)	Directory	old version time		new version time		difference	
	src/lib/omniORB/orbcore	17609835	74.67%	4454185	75.60%	16259744	73.69%
	../omniORB4	3028590	12.84%	0	0.00%	1271565	41.99%
	./tcp	709310	3.01%	381490	6.48%	610915	56.01%
	../../../../include/omniORB4	1734666	7.36%	230765	3.92%	1603189	81.57%
	../../../../include/omniORB4/internal	258071	1.09%	14885	0.25%	243344	89.15%
	src/lib/omnithread	154645	0.66%	567176	9.63%	588313	81.50%
	../../../../include	82657	0.35%	79642	1.35%	13101	8.07%
	_root	3561	0.02%	205	0.00%	0	0.00%
	src/lib/omniORB/dynamic	1476	0.01%	1312	0.02%	578	20.73%
	./unix	28	0.00%	25	0.00%	16	30.19%
	../../../include	0	0.00%	162040	2.75%	81874	50.53%

[Obr. 5] Relatívne porovnanie adresárov

Nakoľko relatívne porovnávanie na úrovni zoznamu súborov spôsobovalo efekt zobrazenia grafov pri súboroch s dominantejším počtom vzoriek a grafy pri ostatných súboroch boli zanedbateľne malé, navyše pri väčšom počte súborov bolo výrazne viac na prvý pohľad nepoužiteľných grafov ako tých použiteľných, preto bolo v tomto prípade vhodnejšie použiť absolútne porovnanie. Takže porovnajú sa počty vzoriek medzi dvoma súbormi a nastaví sa mierka grafu vzhľadom k celkovému počtu vzoriek v oboch súboroch dohromady.

Absolute (Relative)	Directory	old version time		new version time		difference	
	src/lib/omniORB/orbcore	17609835	74.67%	4454185	75.60%	16259744	73.69%
	../omniORB4	3028590	12.84%	0	0.00%	1271565	41.99%
	./tcp	709310	3.01%	381490	6.48%	610915	56.01%
	../../../../include/omniORB4	1734666	7.36%	230765	3.92%	1603189	81.57%
	../../../../include/omniORB4/internal	258071	1.09%	14885	0.25%	243344	89.15%
	src/lib/omnithread	154645	0.66%	567176	9.63%	588313	81.50%
	../../../../include	82657	0.35%	79642	1.35%	13101	8.07%
	_root	3561	0.02%	205	0.00%	0	0.00%
	src/lib/omniORB/dynamic	1476	0.01%	1312	0.02%	578	20.73%
	./unix	28	0.00%	25	0.00%	16	30.19%
	../../../include	0	0.00%	162040	2.75%	81874	50.53%

[Obr. 6] Absolútne porovnanie adresárov

Takže na prvý pohľad je zrejmé, ktorý z dvoch porovnávaných súborov je rýchlejší, teda profiler v ňom zaznamenal menej vzoriek. Aká veľká je váha daného súboru k celému systému je určené počtom vzoriek uvedených za názvom súboru. Súbory sú usporiadané podľa tejto váhy.

Absolute (Relative)	Filename	old version time		new version time		difference	
	src/lib/omniORB/orbcore/giopStream.cc	3119710	17.72%	806920	18.12%	2360135	60.11%
	src/lib/omniORB/orbcore/giopImpl12.cc	2910089	16.53%	767063	17.22%	2159004	58.71%
	src/lib/omniORB/orbcore/giopRope.cc	2483281	14.10%	237320	5.33%	2412900	88.69%
	src/lib/omniORB/orbcore/giopStrand.cc	1922345	10.92%	366270	8.22%	1725513	75.40%
	src/lib/omniORB/orbcore/giopImpl11.cc	1668837	9.48%	201886	4.53%	1740088	93.02%
	src/lib/omniORB/orbcore/cdrStream.cc	794555	4.51%	123794	2.78%	670752	73.04%
	src/lib/omniORB/orbcore/giopImpl10.cc	508009	2.88%	75585	1.70%	553020	94.76%
	src/lib/omniORB/orbcore/uri.cc	446290	2.53%	56154	1.26%	472075	93.96%
	src/lib/omniORB/orbcore/giopServer.cc	422349	2.40%	64589	1.45%	386586	79.39%
	src/lib/omniORB/orbcore/GIOP_C.cc	411415	2.34%	73667	1.65%	374067	77.11%
	src/lib/omniORB/orbcore/GIOP_S.cc	344132	1.95%	87187	1.96%	256957	59.57%
	src/lib/omniORB/orbcore/corbaBoa.cc	318409	1.81%	23923	0.54%	150038	43.83%
	src/lib/omniORB/orbcore/giopWorker.cc	275532	1.56%	23001	0.52%	261851	87.71%
	src/lib/omniORB/orbcore/cdrMemoryStream.cc	237473	1.35%	61708	1.39%	175845	58.78%
	src/lib/omniORB/orbcore/giopStreamImpl.cc	215433	1.22%	47785	1.07%	167685	63.71%
	src/lib/omniORB/orbcore/omniInternal.cc	210245	1.19%	34952	0.78%	175371	71.52%
	src/lib/omniORB/orbcore/callDescriptor.cc	181081	1.03%	34768	0.78%	148320	68.71%
	src/lib/omniORB/orbcore/cdrValueChunkStream.cc	164443	0.93%	15214	0.34%	167479	93.22%
	src/lib/omniORB/orbcore/portableserver.cc	140487	0.80%	20886	0.47%	119609	74.12%
	src/lib/omniORB/orbcore/poa.cc	127602	0.72%	19867	0.45%	107794	73.10%
	src/lib/omniORB/orbcore/omniObjRef.cc	115463	0.66%	32469	0.73%	83059	56.15%
	src/lib/omniORB/orbcore/SocketCollection.cc	98377	0.56%	1144528	25.70%	1184952	95.34%

[Obr. 7] Absolútne porovnanie súborov

Samozrejme na oboch úrovniach bola ponechaná možnosť prepínať medzi oboma zobrazeniami, takže si užívateľ môže zvoliť preferovaný formát.

Na úrovni riadkov zdrojového kódu bola bola zrušená možnosť dvoch zobrazení, pretože na tejto úrovni nevidí pozorovať súbor ako celok, ale vždy iba časť, preto bolo vynechané aj grafické zobrazenie rozdielu. Celkové grafické zobrazenie na tejto úrovni je koncipované trochu inak, pretože riadky zdrojového kódu môžu mať výrazne odlišnú dĺžku. Preto boli dva stĺpce obsahujúce počty vzoriek pre jednotlivé verzie posunuté pred stĺpec, ktorý obsahuje konkrétne data, teda riadok zdrojového kódu. Prvý stĺpec obsahujúci grafické porovanie bol odstranený. Navyše bolo pridané farebné odlíšenie riadkov podľa toho, či bol do projektu pridaný (zelená), odobraný (červená), alebo bol zmenený (tyrkysová).

Pokiaľ sa na niektorej z úrovní (vyššej ako úroveň riadkov zdrojového kódu) objaví v zozname entita ktorá nemá potomkov, nevedie v HTML zobrazení žiadny link z nej ďalej. Takýto prípad sa vyskytne, pokiaľ daná entita leží mimo meraný systém a teda je nezaujímavá pre užívateľa, alebo pokiaľ nebola skompilovaná s príslušnými debugovacími informáciami a profiler nenašiel daný súbor so zdrojovým kódom.

4.3 Špecifikácia vstupných a výstupných dát

Koncept DVT bol navrhnutý tak, aby bol všeobecne použiteľný a nejednalo sa iba o jednorázovú záležitosť, ktorá bude generovať graficky výstup len pre potreby tejto práce. Tento konkrétny HTML výstup bol implementovaný pre jednoduchú

ukážku funkcionality. Komponentová architektúra ale dáva dobré možnosti modifikovať aplikáciu pre vlastné potreby. Je potrebné dodržať formát dát ktoré používajú komponenty na komunikáciu medzi sebou a potom je možné zakomponovať svoje komponenty do aplikácie.

Stredom aplikácie je komponenta Main processor, ktorá má definovaný vstup a výstup v podobe súborov v XML formáte.

Vstup je tvorený dvoma xml súbormi, kde každý z nich obsahuje informácie z profilovania jednej z verzií. Stromová štruktúra XML má nasledujúci formát (popis obsahuje názov elementu <element> a zoznam atribútov [atr1, atr2,...]) :

- <profile> – koreňový element
  - <file> [ticks, name] – popisuje súbor, jeho názov vrátane cesty a celkový počet vzoriek, ktoré zazamenal profiler
    - <line> [ticks] – popisuje jeden riadok kódu, počet vzoriek na ňom a data
  - <totalticks> – celkový počet vzoriek z profilovania celého systému

Výstup je štruktúra XML súborov, ktoré sú uložene v adresári output. Hlavným súborom je directorylist.xml, ktorý obsahuje zoznam profilovaných adresárov a rovnako aj odkazy na ďalšie XML súbory, ktoré popisujú jednotlivé adresáre zo zonamu. Jeho formát vyzerá takto:

- <directorylist> – koreňový element
  - <dir> [ticks old, ticks new, difference, source] – popisuje adresár, jeho názov vrátane cesty a celkový počet vzoriek, ktoré zazamenal profiler
  - <sum> [ticks old, ticks new] – celkový počet vzoriek z profilovania celého systému

Ďalej nasledujú XML súbory, ktoré popisujú jednotlivé adresáre, teda obsahujú zoznam súborov, ktoré sa nachádzajú v daných adresároch a spolu s nimi aj odkazy na XML súbory popisujúce ďalšiu úroveň adresárovej štruktúry. XML popisuje následujúci strom:

- <filelist> – koreňový element
  - <file> [ticks old, ticks new, difference, source] – popisuje jednotlivé súbory, počty vzoriek pre obe verzie systému a rozdiel medzi nimi

Tieto dve úrovne boli takmer identické, ale posledná úroveň a to konkrétne XML súbory, ktoré obsahujú už samotný zdrojový kód obsahuje iné vlastnosti pre jednotlivé riadky. Teda hlavne atribut, ktorý určuje, či daný riadok bol pridaný, zmazaný, prípadne modifkovaný s pohľadu porovania oboch verzií. Túto úroveň výstupov definuje nasledujúci strom:

- <filename> – koreňový element
  - <line> [diff, type, ticks old, ticks new] – popisuje riadky kódu,

4.4 Zdrojové kódy

Súčasťou tejto práce je CD, ktoré obsahuje kompletné zdrojové kódy prototypu vizualizačného nástroja. V súbore readme je popis, ako je možné nástroj spustiť, kde sa nachádzajú testovacie data a akým spôsobom je možné z nich vygenerovať HTML výstup. V prípade nedostupného Perlu, ktorý je potrebný na používanie DVT, CD obsahuje aj vygenerované výsledky, z ktorých úkažky je možné vidieť v tejto práci.

5 Aplikovanie metód na reálne projekty

V tejto kapitole sa budem venovať popisu testovania vyššie spomínaných metod na reálnych projektoch. Vychádzal som opäť z práce DSRG a použil ich systém benchmarkov. Aj svoje metody som sa snažil testovať na rovnakých projektoch. Snahou bolo analyzovať uz nájdené zmeny vo výkonnosti a nájsť k nim príslušnú časť kódu, ktorá danú zmenu spôsobila.

Správnosť hľadania bola overovaná vytvorením určitej medziverzie. Kde sa zobrala nová verzia systému a časť obsahujúca podozrivý kód bola nahradená zo starej verzii systému. Následne sa na takto vytvorený systém znovu pustili benchmarky. V niektorých prípadoch sa takáto verzia systému nepodarila úspešne spustiť, prípadne ak boli zmeny príliš previazané, tak ani skompilovať. Potom museli nasledovať ešte určité úpravy a kompromis, aby bolo možné výsledok aspoň čiastočne overiť. Pri použití metódy porovnania výsledkov profileru dochádzalo často k situácií, že zmena výkonnosti bola zaznamenaná na mieste, kde nedošlo k zmene kódu a teda vytvorenie medziverzie nebolo možné uskutočniť.

Všeobecne je možné rozdelit projekty do 2 skupín podľa toho či bežia priamo nad operačným systémom, alebo je na ich beh potrebné spustiť nejakú virtual machine a až nad ňou beží potom samotný systém. Túto druhú skupinu je možné rozdeliť znovu do 2 kategórií podľa toho, či virtual machine (VM) je vyvíjaná spolu so samotným systémom a teda zmeny v jej zdrojovom kóde môžu vplývať aj na výslednú výkonnosť celého systému, alebo nie.

5.1 MONO

Projekt MONO je koncipovaný ako open source implementácia Microsoft .NET development platform pre UNIX sponzorovaný spoločnosťou Novell. V súčasnosti funguje už na viacerých operačných systémoch. Jedná sa o typ projektu, kde určitý systém beží nad svojou VM a používa svoj kompiler. Náročnosť v hľadaní zmien v tomto projekte je zvýšená aj faktom, že spolu so systémom je neustále vyvíjaný aj jeho kompiler a VM. Takže zmenu vo výkonnosti môžu spôsobovať napríklad aj nejaké optimalizácie použité v kompileri prípadne VM. Na zachytenie takýchto zmien je použiteľná iba prvá metóda. Teda prienik zmien a profilovaných častí kódu s tým, že je potrebné profilovať aj samotnú kompiláciu.

Beh systému nad vlastnou VM prináša aj ďalšie úskalia a to aj v prípade samotného profilovania. Je potrebné aby profiler nejakým spôsobom spolupracoval s danou VM, prípadne sa musí použiť vlastný profiler, ktorý poskytuje daná implementácia systému. V prípade Mona sa mi nepodarilo nájsť všeobecný profiler pre .NET, ktorý by bolo možné použiť. Hlavným dôvodom bolo aj to, že celý benchmarking framework bol pripravený pre Unixovský operačný systémm, preto

som ani profilery z iných OS nebral v úvahu. Bol som teda nútený použiť interný profiler, ktorý poskytoval samotný projekt Mono.

Tento profiler poskytoval štandardné výsledky, teda informácie o jednotlivých funkciách a štatistiku z ich počtu volaní a dĺžky behu. Takže bolo možné presne určiť, ktoré funkcie boli volané. Ale podobný výstup fungoval už aj predtým. S pomocou použitie tracovania programu bol vypísaný zoznam volaných funkcií a počty ich volaní. Snažil som sa použitie profileru vylepšiť o informáciu, kde sa fyzicky (v jakých súboroch) nachádza implementácia daných funkcií, aby bolo možné spraviť prienik s výstupom z diffu. Takáto funkcionalita ale nebola daným profilerom podporovaná a nepodarilo sa mi ani následným študovaním zdrojového kódu Mona prísť na nejaké rozumne náročné a použiteľné riešenie.

Ďalšou teoretickou možnosťou bolo použitie programu monocov – Mono Code Coverage, ktorý mal zobraziť časti kódu potrebné na spustenie určitej úlohy. Po diskusii s autormi Mona som sa dozvedel, že tento modul je mŕtvy projekt a nedoporučujú počítať s tým, že by mohol byť funkčný a použiteľny pre vyššie verzie Mona

Finálne sa teda neporadilo aplikovať a následne overiť správnosť vyššie uvedených metód na tomto projekte, ale táto možnosť zostáva otvorená do budúcnosti, kedy by sa mohol objaviť potrebný profiler.

5.2 omniORB

OmniORB je CORBA ORB pre C++ a Python. Je to voľne dostupná implementácia CORBY, ktorá by mala spĺňať špecifikáciu CORBY verzie 2.6. Viac o tomto projekte je k dispozícii na stránkach [7].

S touto implementáciou CORBY som prešiel na typ projektov, ktoré bežia priamo nad operačným systémom a nie je potrebná žiadna virtual machine ani interpreter kódu.

5.2.1 Hľadanie vhodného profileru – Oprofile

Nakoľko C++ projekty boli testované pod operačným systémom Linux a na ich kompiláciu boli použité GNU nástroje, tak prvá možnosť na profilovanie bolo použitie priamo jedného z týchto nástrojov a to gprof. Gprof ale samotný nespĺňal podmienku iv) všeobecných požiadaviek na profiler definovaných v kapitole o profileroch, preto musel byť použitý ešte program sprof, ktorý síce danú funkcionalitu poskytoval, ale ani kombinácia oboch programov nebola dostatočná na získanie potrebných informácií. Nespĺňali totiž všeobecne požiadavku ii).

Pri ďalšom hľadaní vhodného profileru bol objavený program Oprofile, ktorý spĺňal všetky potrebné podmienky.

Fungovanie Oprofilu je založené na pravidelnom vzorkovaní bežiaceho programu. To znamená, že výstupný profil z behu programu na úrovni jednotlivých riadkov zdrojového kódu nemusí presne zachytiť všetky riadky, ktoré boli interpretované. Pre naše účely to nie je závažný problém, pretože zo štatistického hľadiska je pravdepodobné, že budú označené „správne“ riadky. Teda tie, kde systém trávi veľa času, alebo tie, ktoré sa často opakujú.

Správnosť výstupu bola teda akceptovaná a ďalšou pozorovanou vlastnosťou bola stabilita výstupu. Konkrétne ide o vplyv behu profileru na výsledný čas. Predpoklad bol taký, že pokiaľ benchmarking bez profileru dáva približne rovnaké časy, tak so zapnutým profilerom oprofile by mohli byť tie časy tiež rovnaké, iba o konštantu väčšie ako pôvodne výsledky.

Konkrétne merania ukázali, že tento predpoklad je použiteľný. Nemusí platiť všeobecne a ideálne by bolo túto skutočnosť overiť vždy meraním na konkrétnom systéme. My budeme predpokladať, že to platí.

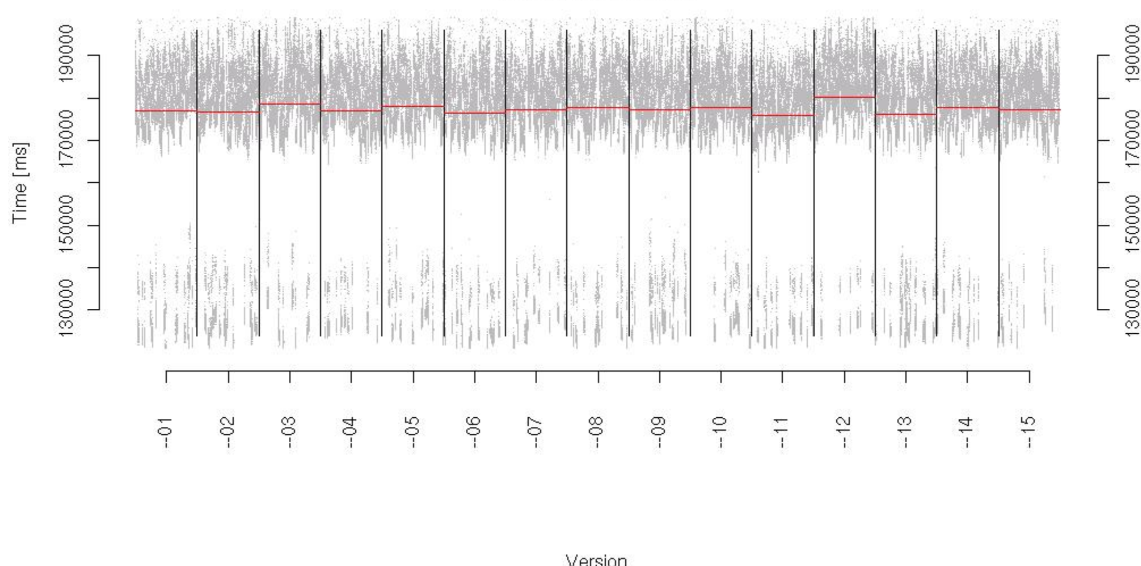
Následujúci graf zobrazuje výsledky takéhoto merania pre omni ORB

Experiment 1	
Sledovaný jav:	Stabilita výsledkov pri použití profileru
Popis experimentu:	Meranie výkonu so zapnutým profilerom
Predpokladaný výsledok:	Pokiaľ pri meraniach výkonu jednej verzie systému dostávame v priemere rovnaké výsledky, potom pri meraní so zapnutýmm profilerom s rovnomerne rozloženou réžiou budú výsledky tiež stabilne rovnaké a o konštantu rozdielne ako pôvodné
Skutočný výsledok:	Predpoklad skutočne platí s použitím OProfile na C++ programe
Obrázok:	Obr. 8

Pri použití OProfile je potrebné, aby všetky získané informácie boli v jednom súbore, pre jednoduchšie spracovanie. Splnenie nasledujúcich bodov je jedna z variant ako je možné pripraviť dáta v tomto formáte:

1) Spustiť merania so zapnutým OProfilerom. Meraný systém musí byť skompilovaný s parametrom poskytujúcim výpis debugovacích informácií
2) Zistiť zoznam zdielaných knižníc (s použitím nástrojov ako napríklad ldd v Linuxe) a binárnych súborov, ktoré boli použité pri behu merania
3) Zavolať *opannotate* s parametrami obsahujúcimi všetky zdielané knižnice a binárne súbory. Viac informácií ako sa pracuje s *opannotate* je možné nájsť na stránkach OProfile [10].
4) Použiť výstup z *opannotate* ako vstup pre vizualizačný nástroj

Pre prototyp vizualizačného nástroju bola implementovaná komponenta typu Input Parser, ktorá načíta takto získané data a pretransformuje ich do formátu VT Input Data, aby bolo možné ich dať na vstup Main Processoru. V tomto bode je možné vidieť použitie teoretického modelu vizualizácie v praxi, pretože z výstupu OProfileru sa vygenerujú HTML stránky, ktoré porovnávajú jednotlivé zdrojové kódy.



[Obr. 8] Stabilita časov pri zapnutom profilovaní OProfilerom

Pri testovaní sa vyskytol problém, že OProfile zahrnul do výsledkov opakovane jeden súbor. Takéto správanie spôsobilo, že DVT zobrazil v zozname súborov správne viac rovnakých položiek s iným počtom vzoriek, ale po rozkliknutí ukazoval vždy iba na jeden posledný súbor. Tento problém je možné odstrániť znížením hodnoty treshold pri volaní programu opannotate, prípadne ak je vyžadovaná informácia o počte všetkých vzoriek, treba počítať s chýbajúcou informáciou na najnižšej úrovni (riadky zdrojového kódu).

5.2.2 Filtrovanie rozdielov kódu v omniORBe

Na základe dlhodobejších meraní DSRG boli vybrané určité verzie, medzi ktorými pri benchmarkingu bola evidentná regresia prípadne zlepšenie výkonnosti.

Medzi týmito dvoma verziami bol spravený diff, teda kompletný zoznam zmien v zdrojovom kóde. Následne boli obe verzie skompilované pomocou g++ s parametrom -g, teda rozšírené o debugovaciu informáciu potrebnú k použitiu OProfileru. Prienik zoznamu súborov z diffu a zoznamu súborov, ktoré boli podľa profileru spustené tvorí množinu súborov, v ktorej je pravdepodobný výskyt regresie. Vytvorí sa teda medziverzia, ktorá ukázala, či predpokladané kritické miesto je skutočne to, kde sa mení výkonnosť systému.

Táto prvá metóda na lokalizáciu bola otestovaná na omniORBe viackrát a s rôznymi výsledkami. Napríklad pri porovnávaní verzií z dní 22.5.2005 a 24.6.2005 vyzeral zoznam zmenených súborov takto:

```
Počet zmenených súborov          46
Počet profilovaných súborov      54
```

Je vidieť, že zoznam súborov, v ktorých boli vykonané zmeny (výstup z diffu) je relatívne obsiahly a ručné analyzovanie jednotlivých zmien by zabralo dosť času. Pokiaľ sa ale zobere v úvahu zoznam súborov, ktoré boli zmenené a zároveň skutočne spustené pri behu benchmarku, potom dostávame relatívne krátky zoznam oproti pôvodnému so všetkými zmenami. Tento prienik predošlých dvoch zoznamov vyzerá takto:

```
src/lib/omniORB/orbcore/giopServer.cc
src/lib/omniORB/orbcore/poa.cc
src/lib/omniORB/orbcore/SocketCollection.cc
src/lib/omnithread/posix.cc
```

Cieľ zúžiť pôvodný diff sa celkom rozumne vydaril. Štatisticky je výsledok uspokojivý, ale na pohľad je jasné, že väčšina zmien je v examples, ktoré nemajú na výsledky benchmarkingu žiadny vplyv. Z výsledných súborov boli ešte na úrovni riadkov ručne vyselektované možnosti regresie a bolo zistené, že zo štyroch spomínaných súborov zostal jeden (SocketCollection.cc), v ktorom zrejme došlo k regresii. A to konkrétne v jednej z funkcií:

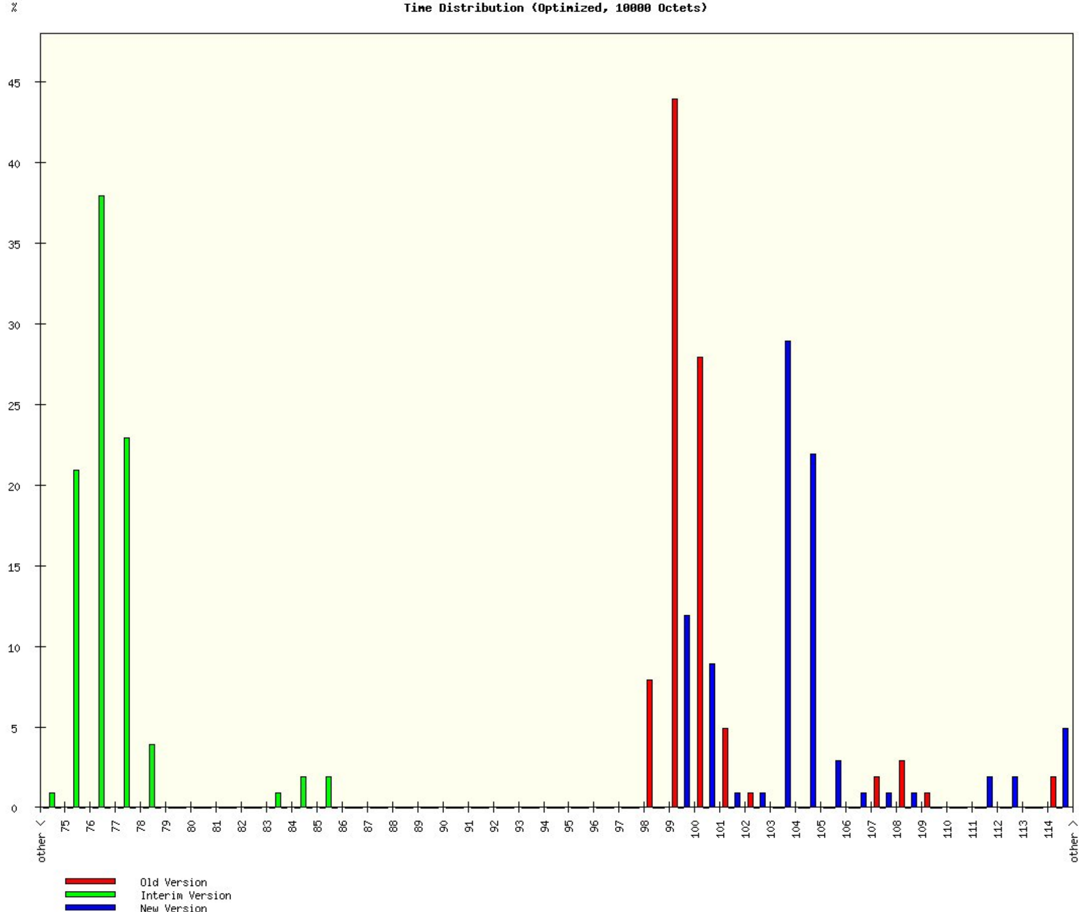
```
SocketCollection::Select()
SocketHolder::Peek()
```

Aj keď teoreticky prichádzali v úvahu ešte ďalšie zmeny, v novej verzii systému boli tieto dve funkcie nahradené pôvodnym kódom a takto vytvorená medziverzia následne skompilovaná a zmerané pre overenie tvrdenia.

Experiment 2	
Sledovaný jav:	Overenie správnosti lokalizácie zmien výkonnosti
Popis experimentu:	Vytvorenie medizverize, meranie jej výkonnosti a porovnanie s verziami 20050522-123900 a 20050624-154300
Predpokladaný výsledok:	Pokiaľ bola regresia správne lokalizovaná, mala by byť výkonnosť medziverzie zhruba rovnaká ako bola u pôvodnej verzie
Skutočný výsledok:	Medziverzia je značne rýchlejšia ako pôvodná verzia
Obrázok:	Obr. 9

Príčinu, prečo medziverzia je výraznejšie rýchlejšia je možné vysvetliť celkom jednoducho. Napríklad ak existujú dva body v zdrojovom kóde, kde sa mení výkonnosť a to tak, že v prvom bode dochádza k zlepšeniu na základe nepriameho vplyvu a v druhom bode naopak k zhoršeniu výkonnosti na základe zmeny typu *priamy vplyv*. Obe zmeny dohromady dávajú zhoršenie, pretože zmena v druhom bode je výraznejšia. V prvej fáze hľadania týchto zmien bola použitá metóda, pomocou ktorej je možné nájsť len zmeny typu *priamy vplyv*, preto sme našli iba zmenu v druhom bode. Tá bola nahradená pôvodnym kódom, takže medziverziu ovplyvnilo iba zrýchlenie v prvom bode.

Tento príklad bol výnimočný v tom, že sa podarilo ukázať správnosť nájdenej regresie. V ostatných prípadoch aj keď sa podarilo lokalizovať zmenu, nebolo možné vytvoriť medziverziu resp. nebolo možné ju skompilovať alebo spustiť. V



[Obr. 9] Graf porovnaní rýchlostí starej, novej a medziverzie

nasledujúcej kapitole popisujúce TAO sú uvedené príklady, kedy sa medziverzia musela upravovať a robiť určité kompromisy, aby ju bolo vôbec možné skompilovať a spustiť. V niektorých prípadoch sa ju nepodarilo vytvoriť vôbec.

5.2.3 Porovnávanie výsledkov profileru omniORBe

Táto metóda je založená na predpoklade, že zapnuté profilovanie ovplyvní benchmarking dvoch rôznych verzií zhruba rovnako, teda konštantne spomalí beh programov. Zameraná je na zobrazenie rozdielov v behu oboch verzií, miesto toho, aby zisťovala, aká je skutočná hodnota výkonnosti daného systému.

Konkrétne na dvoch porovnávaných verziách bol spustený benchmarking so zapnutým profilingom s rovnakou frekvenciou vzorkovania a porovnával sa rozdiel počtov vzoriek medzi oboma verziami. Najprv na úrovni súborov a potom na úrovni jednotlivých funkcií, či dokonca riadkov kódu.

Nasledujúci obrázok je iba výrez ako ukážka, že táto metóda dokázala detekovať zmenu vo výkonnosti na rovnakom mieste ako metóda filtrovania rozdielov:

Súbor						
src/lib/omniORB/orbcore/omniInternal.cc	210245	1.19%	34952	0.78%	175371	71.52%
src/lib/omniORB/orbcore/callDescriptor.cc	181081	1.03%	34768	0.78%	148320	68.71%
src/lib/omniORB/orbcore/cdrValueChunkStream.cc	164443	0.93%	15214	0.34%	167479	93.22%
src/lib/omniORB/orbcore/portableserver.cc	140487	0.80%	20886	0.47%	119609	74.12%
src/lib/omniORB/orbcore/poa.cc	127602	0.72%	19867	0.45%	107794	73.10%
src/lib/omniORB/orbcore/omniObjRef.cc	115463	0.66%	32469	0.73%	83059	56.15%
src/lib/omniORB/orbcore/SocketCollection.cc	98377	0.56%	1144528	25.70%	1184952	95.34%
src/lib/omniORB/orbcore/callHandle.cc	94517	0.54%	24149	0.54%	70584	59.48%
src/lib/omniORB/orbcore/ior.cc	64983	0.37%	8201	0.18%	56864	77.70%
src/lib/omniORB/orbcore/localIdentity.cc	64271	0.36%	12187	0.27%	52088	68.13%
src/lib/omniORB/orbcore/remoteIdentity.cc	58258	0.33%	17491	0.39%	40778	53.83%
src/lib/omniORB/orbcore/giopBiDir.cc	53488	0.30%	13059	0.29%	40433	60.76%
src/lib/omniORB/orbcore/omniIOR.cc	53113	0.30%	15553	0.35%	37593	54.75%

[Obr. 10] Ukážka lokalizovania zmeny výkonnosti

5.3 TAO

TAO je veľmi podobný projekt z pohľadu skúmania výkonnosti ako omniORB. Jedná sa o open source implementáciu CORBY v C++. Viac informácií k tomuto projektu je možné nájsť na [8].

5.3.1 Filtrovanie rozdielov kódu v TAO CORBE

Podobne ako v prípade omniORBU som začal testovať použiteľnosť prvej metódy hľadania zmien výkonnosti na verziách systému, na ktorých podľa DSRG evidentne k takejto zmene došlo. Na rozdiel od omniORBu sa nepodarilo nájsť taký jednoznačný prípad, pri ktorom by medziverzia vznikla presne nahradením podozrivých častí kódu a následné meranie výkonnosti by potvrdilo výskyt regresie v danej časti.

Následujúce dva príklady reprezentujú 2 skupiny, do ktorých je možné zaradiť všetky TAO regresie, kde som skúšal reálne lokalizovať danú regresiu.

Prvý vzorový príklad vznikol pri porovnávaní verzií s timestampami 20050713-171045 a 20050718-080035. Zoznam zmenených súborov a zoznam profilovaných súborov sú pre svoj väčší rozsah uvedené v prílohe A.1.

```
Počet zmenených súborov        131
Počet profilovaných súborov    314
```

Prienik týchto dvoch zoznamov znamená zoznam súborov, v ktorých je podozrenie na vznik regresie:

```
ACE wrappers/TAO/tao/GIOP Message Base.cpp
ACE wrappers/TAO/tao/IIOP Acceptor.cpp
ACE wrappers/TAO/tao/ORB Core.cpp
ACE wrappers/TAO/tao/ORB Core.i
ACE wrappers/TAO/tao/ORB Core.i
ACE wrappers/TAO/tao/Profile Transport Resolver.cpp
ACE wrappers/TAO/tao/Thread Lane Resources Manager.cpp
ACE wrappers/TAO/tao/Transport.cpp
ACE wrappers/TAO/tao/Transport.inl
```

```
ACE wrappers/TAO/tao/default resource.cpp
ACE wrappers/TAO/tao/params.cpp
ACE wrappers/TAO/tao/params.i
ACE wrappers/ace/OS NS stdio.cpp
```

po následnej riadkovej analýze bol tento zoznam zredukovaný na nasledujúce súbory:

```
ACE wrappers/TAO/tao/Transport.cpp
ACE wrappers/TAO/tao/GIOP Message Base.cpp
ACE wrappers/TAO/tao/Profile Transport Resolver.cpp
ACE wrappers/TAO/tao/ORB Core.i
ACE wrappers/TAO/tao/IIOP Acceptor.cpp
```

Na všetkých podozrivých miestach v týchto súboroch bol nahradený nový kód pôvodným. Bohužiaľ zmeny boli väčšinou kontroly nejakého ukazateľa, či jeho hodnota nie je null a následne sa volala funkcia objektu na ktorý tento ukazateľ smeroval. Po nahradení daných miest pôvodnou verziu končil beh benchmarky chybou segmentation fault na daných miestach, takže v nasledujúcich súboroch museli byť zmeny (návrat k pôvodnému kódu) zrušené:

```
ACE wrappers/TAO/tao/Transport.cpp
ACE wrappers/TAO/tao/GIOP Message Base.cpp
ACE wrappers/TAO/tao/Profile Transport Resolver.cpp
ACE wrappers/TAO/tao/IIOP Acceptor.cpp
```

Medziverzia nakoniec bola sputiteľná iba so zmenou v súbore

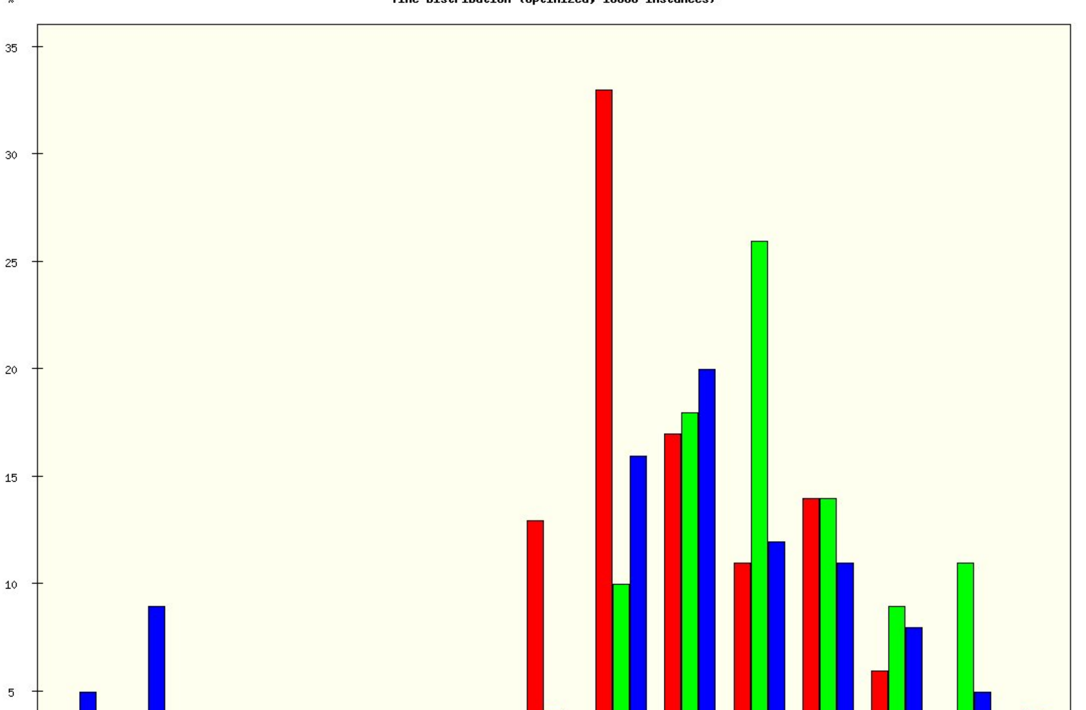
```
ACE wrappers/TAO/tao/ORB Core.i
```

čo je málo a odráža to aj výsledný experiment

Experiment 3	
Sledovaný jav:	Overenie správnosti lokalizácie regresie
Popis experimentu:	Vytvorenie medziverzie, meranie jej výkonnosti a porovnanie s verziami 20050713-171045 a 20050718-080035
Predpokladaný výsledok:	Pokiaľ bola regresia správne lokalizovaná, mala by byť výkonnosť medziverzie zhruba rovnaká ako výkonnosť pôvodnej verzie
Skutočný výsledok:	Graf ukazuje, že regresia nenastala v spomínanom jednom súbore, ale v jednom zo štyroch, ktoré nemohli byť zmenené a teda overené podozrenia.
Obrázok:	Obr. 11

Opäť sa podarilo výrazne zmenšiť počet podozrivých súborov, v ktorých nastali zmeny a ktoré mohli spôsobiť regresiu. Bohužiaľ sa to nepodarilo potvrdiť. Ale je možné sa domnievať, že zmeny typu

```
a()->b()->c();
```



[Obr. 11]Graf porovnaní rýchlostí starej, novej a medziverzie

za

```
object *o = a()->b(); if (o) o->c();
```

môžu mať za následok regresiu. Jedná sa totiž o vytváranie ukazateľa na objekt a pridanie podmienky na miestach, ktoré boli často volané.

Druhú skupinu reprezentuje príklad porovnaní verzií s timestampami 20051013-175104 a 20051014-080512. Zoznam zmenených a profilovaných súborov je uvedený v prílohe A.2. Rozsah zmien nie je až taký veľký ako v predchadzajúcom príklade, napriek tomu prienik oboch zoznamov dokázal tento počet podozrivých súborov výrazne zmenšiť:

```
ACE wrappers/TAO/tao/ORB Core.cpp
ACE wrappers/TAO/tao/Pseudo VarOut T.inl
ACE wrappers/TAO/tao/SystemException.inl
ACE wrappers/TAO/tao/TSS Resources.cpp
ACE wrappers/ace/RB Tree.cpp
```

A po následnej riadkovej analýze zostali nasledujúce súbory:

```
ACE wrappers/TAO/tao/Pseudo VarOut T.inl
ACE wrappers/TAO/tao/SystemException.inl
```

`ACE wrappers/ace/RB Tree.cpp`

Experiment 4	
Sledovaný jav:	Overenie správnosti lokalizovania regresie
Popis experimentu:	Vytvorenie medziverzie, zistenie jej výkonnosti a porovnanie s verziami 20051013-175104 a 20051014-080512
Predpokladaný výsledok:	Pokiaľ bola regresia správne lokalizovaná, mala by byť výkonnosť medziverzie zhruba rovnaká ako výkonnosť pôvodnej verzie
Skutočný výsledok:	Zmeny v týchto súboro boli natoľko previazané s ostatnými zmenami, že medziverziu na overenie správnosti nebolo možné ani skompilovať
Obrázok:	N/A

5.3.2 Porovnávanie výsledkov profileru v TAO CORBE

Nasledujúce obrázky (Obr. 12, Obr. 13, Obr. 14) ukazujú použitie DVT pri hľadaní zmien výkonnosti v TAO corbe.

Absolute (Relative)	Directory	old version time		new version time		difference	
	ACE wrappers/ace	71175282	40.72%	13474529	39.38%	89648836	105.91%
	/usr/include/bits	9	0.00%	0	0.00%	0	0.00%
	ACE wrappers/TAO/tao	91899089	52.58%	17940955	52.44%	71368216	64.97%
	PortableServer	11349151	6.49%	2697678	7.88%	8714552	62.04%
	ACE wrappers/TAO/tao/PortableServer	28846	0.02%	10920	0.03%	17925	45.08%
	ACE wrappers/TAO/orbsvcs/orbsvcs	6	0.00%	2	0.00%	8	100.00%
	Messaging	340387	0.19%	90817	0.27%	271444	62.95%
	PI	5	0.00%	2	0.00%	0	0.00%
	ACE wrappers/TAO/tao/CodecFactory	2	0.00%	0	0.00%	0	0.00%
	Naming	6	0.00%	2	0.00%	4	50.00%
	ACE wrappers/TAO/tao/Valuetype	0	0.00%	1	0.00%	0	0.00%
	IORTable	0	0.00%	1	0.00%	0	0.00%

[Obr. 12] Provnanie adresárov

Jedná sa len o ukážky použitia, správnosť výsledkov nie je možné overiť, pretože z prvej metódy sme nedostali žiadny podklad a rovnako nie je možné ani vytvoriť medziverziu. Takže správnosť lokalizácie zostáva na posúdení užívateľa.

Pri porovnávaní jednotlivých riadkov kódu sú farebne zvýraznené rozdiely medzi oboma verziami.

- modrá - rozdielny počet vzoriek pre riadok
- červená - riadok bol odobraný zo systému
- zelená - riadok bol pridaný do systému

Absolute (Relative)	Filename	old version time		new version time		difference	
	PortableServer/Root_POA.cpp	2576027	22.70%	621579	23.04%	1980020	61.92%
	PortableServer/Servant_Upcall.cpp	1190281	10.49%	296833	11.00%	893532	60.08%
	PortableServer/Object_Adapter.cpp	936255	8.25%	262379	9.73%	712848	59.47%
	PortableServer/Servant_Base.cpp	771503	6.80%	171464	6.36%	603529	64.00%
	PortableServer/Active_Object_Map.cpp	728898	6.42%	165445	6.13%	565045	63.18%
	PortableServer/ServantRetentionStrategyRetain.cpp	498736	4.39%	129087	4.79%	372275	59.30%
	PortableServer/Key_Adapters.cpp	423017	3.73%	105214	3.90%	317803	60.16%
	PortableServer/Upcall_Wrapper.cpp	417352	3.68%	85797	3.18%	331555	65.90%
	PortableServer/POA_Current_Impl.cpp	361781	3.19%	91762	3.40%	270019	59.54%
	PortableServer/LifespanStrategyTransient.cpp	342494	3.02%	79026	2.93%	263470	62.50%
	PortableServer/RequestProcessingStrategyAOMOnly.cpp	296625	2.61%	52384	1.94%	244241	69.98%
	PortableServer/Operation_Table_Perfect_Hash.cpp	270883	2.39%	44210	1.64%	226673	71.94%

[Obr. 13] Provnanie súborov

0	0	`    find_servant (system_id ACE_ENV_ARG_PARAMETER);`
1307	383	`}`
0	0	`int`
0	0	`TAO_Root_POA::unbind_using_user_id (const PortableServer::ObjectId &user_id)`
41711	0	`{`
0	10617	`{ /* TAO_Root_POA::unbind_using_user_id(CORBA::OctetSeq const&) total:`
0	0	`  return this->active_policy_strategies_.servant_retention_strategy()->`
0	0	`    unbind_using_user_id  (user_id);`
1561	396	`}`
0	0	`void`
0	0	`TAO_Root_POA::cleanup_servant (`
0	0	`  PortableServer::Servant servant,`
0	0	`  const PortableServer::ObjectId &user_id`
0	0	`  ACE_ENV_ARG_DECL)`
42002	8785	`{`
0	0	`  this->active_policy_strategies_.request_processing_strategy()->`

[Obr. 14] Porovnanie jednotlivých riadkov kódu

5.4 Java projekty - JORAM, ProActive

Projekty vytvorené v Jave patria do druhej skupiny podľa rozdelenia uvedeného na začiatku tejto kapitoly. Jedná sa o systémy, ktoré bežia nad vlastnou virtual machine. Virtual machine však rovnako ako kompilátor nie je súčasťou vývoju daného systému. Bližšie informácie o projektoch je možné nájsť na ich internetových stránkach [11][12].

Snažil som sa v tomto prípade použiť už výskúšaný a vyhovujúci Oprofile profiler. Chcel som použiť známy matematický postup a to previesť prípad na známy tvar, pre ktorý už je riešenie hotové. Teda upraviť beh Java programu tak, aby bol profilovateľný pomocou Oprofileru. V dnešnej dobe je možné skompilovať zdrojový kód v Jave do binárneho tvaru spustiteľného priamo operačným systémom

bez použitia Virtual Machine. Konkrétne bol vyskúšaný GCJ (GNU Compiler for the Java), ktorého funkčnosť je popísaná na stránkach[9].

V čase testovania tohto postupu, nemal ešte gcj dostatočné vyvinuté knižnice všeobecne potrebné pre beh daných systémov, navyše by bolo potrebné upravovať pre každý systém kompilačné skripty, pretože Ant skripty neboli podporované. Tento koncept však zostáva otvorený do budúcnosti.

5.4.1 Hľadanie vhodného profileru

Nakoľko sa java projekty nepodarilo skompilovať a spustiť merania použitím gcj, bolo nutné nájsť vhodný profiler, ktorý by spolupracoval s kompilátorom a Virtual Machine od firmy Sun. Ako prvý bol testovaný HPROF, ktorý je vývíjaný rovnakou firmou. HPROF poskytoval štandardný štatistický výstup z behu programu, ktorý obsahoval informácie o behu jednotlivých metód v takomto formáte:

```
rank    self     accum    count   trace    method
1       53.17%   53.17%   67      300027   java.util.zip.ZipFile.getEntry
2       17.46%   70.63%   22      300135   java.util.zip.ZipFile.getNextEntry
3       5.56%    76.19%   7       300111   java.lang.ClassLoader.defineClass2
4       3.97%    80.16%   5       300140   java.io.UnixFileSystem.list
5       2.38%    82.54%   3       300149   java.lang.Shutdown.halt0
6       1.59%    84.13%   2       300136   java.util.zip.ZipEntry.initFields
```

čo je bežný výstup pri použití profileru. Je to však nedostačujúce pre použitie definovaných metód, pretože nie je možné zistiť kde fyzicky v súboroch sa nachádza implementácia volaných metód. Aj keď vo všeobecnosti maju zdrojové kódy Javy vlastnosť, že názov balíka je totožný s adresárovou štruktúrou, nie je triviálne použiť túto vlastnosť, pretože meraný systém môže byť tvorený viacerými modulmi, ktoré sú kompilované zvlášť a priložené v podobe .jar súborov.

S podobným výsledkom skončilo aj testovanie ďalších profilerov (JAMon, JRat, JMP, JIP) a dokonca sa ukázalo, že pôvodný HPROF má niekoľkonásobne vyššiu réžiu ako niektoré "lightweight" profilery.

Ako už bolo spomínané v popise metódy Filtrovanie rozdielov kódu na použitie tejto metódy nie je priamo potrebný profiler, ale postačí nástroj, ktorý sa označuje ako Code Coverage. Takýto nástroj zobrazuje, ktoré časti zdrojových kódov boli volané pri spustení programu. Na rozdiel od profileru nezisťuje aký dlhý čas trávil program v jednotlivých častiach.

Začal som postupne skúmať použiteľnosť jednotlivých nástrojov:

- CodeCover – pracuje na princípe transformácie zdrojových kódov, kam pridáva svoje inštrukcie, ktoré zaznamenávajú beh aplikácie. Používa pri tom Javovské anotácie, ktoré sú k dispozícii od verzie Javy 1.5. Oba projekty (JORAM, ProActive) sú vytvorené pre nižšiu verziu Javy a nebolo možné ich skompilovať ani po niekoľko základných úpravách
- JVMDICover – pracuje na princípe použitia JVMDI (Java Virtual Machine Debug Interface), čo je rozhranie na získavanie informácií o behu programu priamo z Virtual Machine, nad ktorou beží. Ani pri použití tohto nástroja som nebol úspešný, pretože nevracal informácie o celom systéme, ale iba o jeho

časti, aj keď kompilácia systému prebehla aj so zapnutými debug informáciami. Príčina sa mi nepodarila zistiť.
- EMMA, Cobertura – sú samostatné Java programy, ktoré bežia nad Virtual Machine a nad nimi beží systém, o ktorom by mali poskytovať informácie. Projekt ProActive vytvára dynamicky pri svojom behu triedy, ktoré potom používa. S týmto bol problém, pretože tieto novovzniknuté triedy neboli v ceste pre code cover programy a tieto skončili s chybou, že dané triedy nemôžu nájsť. Projekt JORAM sa podarilo pomocou týchto nástrojov zanalyzovať

Ako sa ukázalo projekt ProActive je zrejme príliš komplikovaný na to, aby sa dal bežnými nástrojmi jednoducho a podrobne zanalyzovať jeho bed, preto som sa ďalej týmto projektom nezaoberal a výskušal otestovať aspoň jednu navrhnutú metódu na regresiách projektu JORAM.

5.4.2 Filtrovanie rozdielov kódu – JORAM

Podobne ako pri predošlých projektoch boli vytipované určité verzie, medzi ktorými došlo k jednoznačnej zmene vo výkonnosti. Medzi týmito verziami bol spravený diff, následne boli spustené a monitorované pomocou programu EMMA. Na základe prieniku rozdielov a spustených častí bola vytvorená medziverzia, teda podozrivé časti nahradené pôvodným kódom a zmeraná jej výkonnosť.

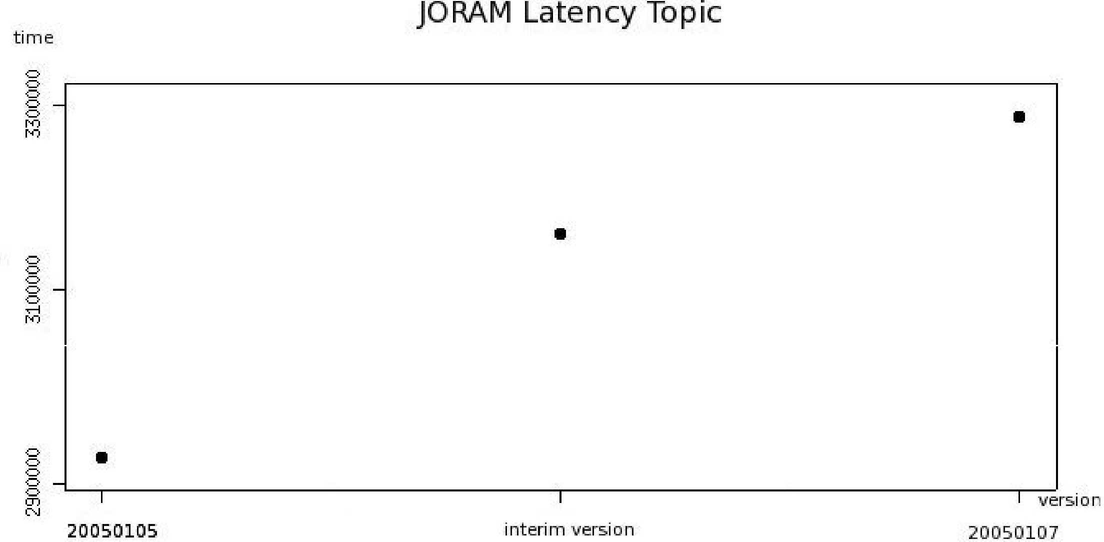
Rovnako ako pri C++ projektoch, bolo vytvorenie medziverzie závislé od previazanosti jednotlivých zmien medzi sebou. Takže nie vždy sa podarilo vytvoriť medziverziu, prípadne bolo nutné spraviť kompromis, ktorý znamenal, že iba časť podozrivého kódu mohla byť nahradená pôvodným.

Príkladom je porovnanie verzií 20050105 a 20050107:

```
Počet zmenených súborov         79
Počet profilovaných súborov     68
```

prienik týchto dvoch zoznamov tvorilo 17 súborov, ale z týchto všetkých podozrivých miest sa do medziverzie podarilo zahrnúť (nahradiť pôvodným kódom) iba 6. Po zmeraní výkonnosti medziverzie sa však ukázalo, že aj týchto 6 súborov čiastočne ovplyvnilo výkonnosť novej verzie.

Experiment 3	
Sledovaný jav:	Overenie správnosti lokalizácie regresie
Popis experimentu:	Vytvorenie medziverzie, meranie jej výkonnosti a porovnanie s verziami 20050105 a 20050107
Predpokladaný výsledok:	Pokiaľ bola regresia správne lokalizovaná, mala by byť výkonnosť medziverzie zhruba rovnaká ako výkonnosť pôvodnej verzie
Skutočný výsledok:	Graf ukazuje, že sa podarilo čiastočne lokalizovať zmenu výkonnosti meraného systému
Obrázok:	Obr. 15



[Obr. 15]Graf porovnaní rýchlostí starej, novej a medziverzie

Pri pozorovaní zmien v ostatných podozrivých súboroch bolo zistené, že sa jednalo predovšetkým o implementáciu nových funkcií do systému, takže zníženie výkonnosti je možné odôvodniť týmto.

6 Záver

Cieľom tejto práce bolo zdokonaliť vyhľadávanie príčin zmien výkonnosti v zdrojových kódoch a uľahčiť tak prácu pri testovaní softwaru. Pre tento účel boli navrhnuté dve metódy:

- Filtrovanie rozdielov kódu
- Porovnávanie výsledkov profileru

V prvej metóde šlo o vylepšenie už existujúcej metódy, ktorá hľadá príčinu zmien vo výsledku diffu zdrojového kódu oboch verzií. Táto metóda bola vylepšená o použitie profileru a teda zúženie tejto množiny na prienik výsledku z diffu a zoznamom profilovaných častí. Táto metóda je celkom dobre použiteľná pri zmenách typu *priamy vplyv*, kedy beh kódu je spomalený presne v tom istom mieste ako bol zmenený. Naopak ak spomalenie kódu je spôsobené na mieste, ktoré leží mimo profilovanú časť, tak táto metóda ho vôbec nezaznamená.

Druhá metóda je založená čisto na porovnávaní výsledkov z profileru. Dokáže detekovať všetky typy zmien, ale jej funkcionalita je závislá na fungovaní profileru. Často dochádza k lokálizácií zmien, ktoré nie je možné overiť vytvorením medziverzie a kladie väčší dôraz na znalosť systému a spôsobu volania jeho funkcií. Pre reálne použitie tejto metódy bol navrhnutý systém na porovnávanie a vizualizáciu výsledkov.

Obe prezentované metódy sú založené na použití profileru, ktorý môže spôsobiť, že skutočná zmena výkonnosti nebude zachytená a dokonca že vznikne nová a na inom mieste. Preto by som obe metódy nazval skôr približnými metódami. Takže nájdene podozrivé miesta v zdrojovom kóde, kde by sa mohla vyskytovať zmena vo výkonnosti, použitím týchto dvoch metód nie je exaktná lokalizácia, ale skôr lokalizácia s určitou pravdepodobnosťou úspechu. Konečné rozhodnutie ostáva naďalej v rukách človeka, ktorý svojími skúsenosťami posúdi, či daná zmena v kóde mohla mať za následok zmenu vo výkonnosti.

Ukázalo sa, že nájsť vhodné nástroje (profiler, code coverage) je netriviálna úloha, hlavne z časového hľadiska a jednoznačne je to najťažšia časť pri riešení lokalizácie zmeny výkonnosti prezentovanými metódami. Po preštudovaní dokumentácie a triviálne fungujúcej ukážky sa častokrát dostaví nepoužiteľnosť pri rozsiahlejšom projekte, ktorý využíva napr. nepodporované vlastností daného jazyka. Možno to bolo čiastočne spôsobené aj zameraním sa na voľne dostupné väčšinou open source nástroje.

Ciele práce sa podarilo čiastočne splniť a vytvoriť určité nové postupy. Naďalej ale výsledky v tejto oblasti zostávajú nedeterministické a niektoré anomálie správania sa systému pri meraní jeho výkonnosti neboli presne ujasnené

Pri ďalšom pokračovaní v skúmaní tejto oblasti by bolo dobré zamerať sa na otestovanie ďalších profilerov a hlavne použitie spomínaných metód v aplikáciach, ktoré potrebujú pre svoj beh zvláštnu virtual machine (MONO, Java). Priestor je

aj vo vylepšovaní vizualizačného nástroja, kde by som pridal možnosť porovnávať viac ako dve verzie a pridať grafy vývoja lokálnej zmeny výkonnosti.

7 Zdroje

[1] Kalibera T., Bulej, L., Tůma, P. (2005): *Quality Assurance in Performance: Evaluating Mono Benchmark Results.* in proceedings of the Second International Workshop on Software Quality (SOQUA 2005), Erfurt, Germany, Copyright (C) Springer-Verlag, Berlin, LNCS3712, ISBN 3-540-29033-8, ISSN 0302-9743, pp. 271-288

[2] Robert Bernecky (1989): *Profiling, performance and perfection.*

[3] Kalibera T., Bulej, L., Tůma, P. (2005): *Automated Detection of Performance Regressions: The Mono Experience.* in proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2005), Atlanta, GA, USA, Copyright (C) IEEE, Piscataway, New Jersey, USA, ISBN 0-7695-2458-3, ISSN 1526-7539, pp. 183-190

[4] Kalibera T., Bulej, L., Tůma, P. (2005): *Benchmark Precision and Random Initial State.* in Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2005), Cherry Hill, NJ, USA, Copyright (C) SCS, San Diego, CA, USA, pp. 853-862, ISBN 1-56555-300-4

[5] Bulej, L., Kalibera, T., Tůma, P. (2005): *Repeated Results Analysis for Middleware Regression Benchmarking.* in Performance Evaluation: An International Journal, Performance Modeling and Evaluation of High-Performance Parallel and Distributed Systems, vol. 60, pg. 345-358, Elsevier B.V., ISSN 0166-5316

[6] Buble. A, Bulej, L., Tůma, P. (2003): *CORBA Benchmarking: A Course With Hidden Obstacles.* in proceedings of the 2003 Internation Parallel and Distributed Processing Symposium (IPDPS 2003), Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEOPDS 2003), Nice, France, Copyright (C) 2003 IEEE, Piscataway, New Jersey, USA, ISBN 0-7695-1926-1, ISSN 1530-2075, pp. 279

7.1 Ostatné zdroje

[7] omniORB: .
http://omniorb.sourceforge.ne

[8] TAO CORBA: .
http://http://www.cs.wustl.edu/ schmidt/TAO.html

[9] The GNU Compiler for the Java Programming Language: .
http://gcc.gnu.org/java/

[10] OProfile profiler: .
http://oprofile.sourceforge.net

[11] JORAM: .
http://joram.objectweb.org

[12] ProActive: .
http://proactive.inria.fr/

8 Appendix A.

8.1 Porovnanie verzií 20050522-123900 a 20050624-154300

Zoznam menených súborov (výstup z diffu):

```
doc/tex/omniORB.tex
include/omniORB4/internal/SocketCollection.h
src/appl/utils/catior/catior.cc
src/appl/utils/convertior/convertior.cc
src/appl/utils/nameclt/nameclt.cc
src/examples/anyExample/anyExample clt.cc
src/examples/anyExample/anyExample impl.cc
src/examples/bidir/bd client.cc
src/examples/bidir/bd server.cc
src/examples/bidir/bd shutdown.cc
src/examples/boa/eg2 clt.cc
src/examples/boa/eg2 impl.cc
src/examples/call back/cb client.cc
src/examples/call back/cb server.cc
src/examples/call back/cb shutdown.cc
src/examples/dii/echo diiclt.cc
src/examples/dsi/echo dsiimpl.cc
src/examples/echo/eg1.cc
src/examples/echo/eg2 clt.cc
src/examples/echo/eg2 impl.cc
src/examples/echo/eg3 clt.cc
src/examples/echo/eg3 impl.cc
src/examples/echo/eg3 tieimpl.cc
src/examples/poa/implicit activation/eg1.cc
src/examples/poa/persistent objref/eg2 impl.cc
src/examples/poa/servant manager/servant activator.cc
src/examples/poa/servant manager/servant locator.cc
src/examples/poa/threading/mainthread.cc
src/examples/ssl echo/eg2 clt.cc
src/examples/ssl echo/eg2 impl.cc
src/examples/thread/diner.cc
src/examples/thread/prodcons.cc
src/examples/thread/thrspecdata.cc
src/examples/valuetype/simple/valimpl.h
src/examples/valuetype/simple/vclient.cc
src/examples/valuetype/simple/vcoloc.cc
src/examples/valuetype/simple/vserver.cc
src/lib/omniORB/dynamic/valueType.cc
src/lib/omniORB/omniidl be/cxx/impl/template.py
src/lib/omniORB/orbcore/giopServer.cc
src/lib/omniORB/orbcore/poa.cc
src/lib/omniORB/orbcore/SocketCollection.cc
src/lib/omnithread/posix.cc
src/tool/omniidl/cxx/idldump.cc
src/tool/omniidl/python/omniidl be/dump.py
update.log
```

Po následnom spustení meraní so zapnutým profilingom bol zoznam spúšťaných súborov tento:

```
src/lib/omniORB/orbcore/giopImpl12.cc
src/lib/omniORB/orbcore/giopStream.cc
src/lib/omniORB/orbcore/giopStrand.cc
src/lib/omniORB/orbcore/SocketCollection.cc
src/lib/omniORB/orbcore/GIOP S.cc
src/lib/omniORB/orbcore/cdrStream.cc
src/lib/omniORB/orbcore/giopRope.cc
src/lib/omniORB/orbcore/cdrMemoryStream.cc
src/lib/omniORB/orbcore/GIOP C.cc
src/lib/omniORB/orbcore/giopServer.cc
src/lib/omniORB/orbcore/omniObjRef.cc
src/lib/omniORB/orbcore/omniInternal.cc
src/lib/omniORB/orbcore/callHandle.cc
src/lib/omniORB/orbcore/giopStreamImpl.cc
src/lib/omniORB/orbcore/omniTransport.cc
src/lib/omniORB/orbcore/poa.cc
src/lib/omniORB/orbcore/remoteIdentity.cc
src/lib/omniORB/orbcore/giopWorker.cc
src/lib/omniORB/orbcore/giopBiDir.cc
src/lib/omniORB/orbcore/localIdentity.cc
src/lib/omniORB/orbcore/omniIOR.cc
src/lib/omniORB/orbcore/callDescriptor.cc
src/lib/omniORB/orbcore/invoker.cc
src/lib/omniORB/orbcore/giopRendezvouser.cc
src/lib/omniORB/orbcore/current.cc
src/lib/omniORB/orbcore/orbOptions.cc
src/lib/omniORB/orbcore/cs-8bit.cc
src/lib/omniORB/orbcore/corbaString.cc
src/lib/omniORB/orbcore/libcWrapper.cc
src/lib/omniORB/orbcore/uri.cc
src/lib/omniORB/orbcore/transportRules.cc
src/lib/omniORB/orbcore/initRefs.cc
src/lib/omniORB/orbcore/corbaOrb.cc
src/lib/omniORB/orbcore/ior.cc
src/lib/omniORB/orbcore/rmutex.cc
src/lib/omniORB/orbcore/portableserver.cc
src/lib/omniORB/orbcore/policy.cc
src/lib/omniORB/orbcore/poastubs.cc
src/lib/omniORB/orbcore/omniServant.cc
src/lib/omniORB/orbcore/omniORB.cc
src/lib/omniORB/orbcore/logIOstream.cc
src/lib/omniORB/orbcore/giopEndpoint.cc
src/lib/omniORB/orbcore/dynamicLib.cc
src/lib/omniORB/orbcore/corbaObject.cc
src/lib/omniORB/orbcore/corbaFixed.cc
src/lib/omniORB/orbcore/codeSets.cc
src/lib/omniORB/dynamic/typecode.cc
src/lib/omniORB/dynamic/dynException.cc
src/lib/omniORB/dynamic/valueFactory.cc
src/lib/omniORB/dynamic/valueBase.cc
src/lib/omniORB/dynamic/pseudoBase.cc
src/lib/omniORB/dynamic/poastub.cc
src/lib/omniORB/dynamic/environment.cc
src/lib/omniORB/dynamic/dynamicLib.cc
src/lib/omnithread/posix.cc
src/lib/omnithread/threaddata.cc
```

8.2 Porovanie verzií 20050713-171045 a 20050718-080035

Zoznam menených súborov (výstup z diffu):

ACE wrappers/ASNMP/ChangeLog
ACE wrappers/COPYING
ACE wrappers/ChangeLog
ACE wrappers/TAO/CIAO/COPYING
ACE wrappers/TAO/CIAO/ChangeLog
ACE wrappers/TAO/CIAO/DAnCE/NodeManager/NAM Map.h
ACE wrappers/TAO/CIAO/NEWS
ACE wrappers/TAO/COPYING
ACE wrappers/TAO/ChangeLog
ACE wrappers/TAO/NEWS
ACE wrappers/TAO/TAO IDL/be/be visitor valuebox/valuebox ci.cpp
ACE wrappers/TAO/docs/Options.html
ACE wrappers/TAO/docs/performance.html
ACE wrappers/TAO/orbsvcs/Logging Service/RTEvent Logging Service/RTEvent Logging Service.cpp
ACE wrappers/TAO/orbsvcs/examples/Notify/Federation/Agent/Agent.cpp
ACE wrappers/TAO/orbsvcs/examples/Notify/Federation/Gate/Gate.cpp
ACE wrappers/TAO/orbsvcs/examples/Notify/Federation/SpaceCraft/SpaceCraft.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/DsLogAdmin.mpc
ACE wrappers/TAO/orbsvcs/orbsvcs/ETCL.mpc
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/BasicLogFactory i.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/BasicLogFactory i.h
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/BasicLog i.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/BasicLog i.h
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/EventLogFactory i.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/EventLogFactory i.h
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/EventLog i.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/EventLog i.h
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/Iterator i.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/Iterator i.h
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/LogMgr i.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/LogMgr i.h
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/LogNotification.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/LogNotification.h
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/LogRecordStore.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/LogRecordStore.h
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/Log Compaction Handler.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/Log i.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/Log i.h
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/NotifyLogFactory i.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/NotifyLogFactory i.h
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/NotifyLog i.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/NotifyLog i.h
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/RTEventLogFactory i.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/RTEventLogFactory i.h
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/RTEventLog i.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/Log/RTEventLog i.h
ACE wrappers/TAO/orbsvcs/orbsvcs/Makefile.am
ACE wrappers/TAO/orbsvcs/orbsvcs/PortableGroup/PG PropertyManager.h
ACE wrappers/TAO/orbsvcs/orbsvcs/PortableGroup/Portable Group Map.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/PortableGroup/Portable Group Map.h
ACE wrappers/TAO/orbsvcs/performance-tests/RTEvent/RTCORBA Baseline/RTCORBA Baseline.mpc
ACE wrappers/TAO/orbsvcs/performance-tests/RTEvent/RTCORBA Callback/RTCORBA Callback.mpc
ACE wrappers/TAO/orbsvcs/tests/InterfaceRepo/Application Test/ifr dii client.cpp
ACE wrappers/TAO/tao/Any.h
ACE wrappers/TAO/tao/ClientRequestInterceptor Adapter Factory.h
ACE wrappers/TAO/tao/Codeset Manager.cpp
ACE wrappers/TAO/tao/Codeset Manager.h
ACE wrappers/TAO/tao/Codeset Translator Factory.h
ACE wrappers/TAO/tao/DynamicInterface/Context.cpp
ACE wrappers/TAO/tao/DynamicInterface/DII Invocation.cpp
ACE wrappers/TAO/tao/DynamicInterface/Request.cpp
ACE wrappers/TAO/tao/DynamicInterface/Request.h
ACE wrappers/TAO/tao/DynamicInterface/Request.inl
ACE wrappers/TAO/tao/DynamicInterface/Server Request.cpp
ACE wrappers/TAO/tao/DynamicInterface/Server Request.h
ACE wrappers/TAO/tao/DynamicInterface/Server Request.inl
ACE wrappers/TAO/tao/GIOP Message Base.cpp
ACE wrappers/TAO/tao/GIOP Message Lite.cpp
ACE wrappers/TAO/tao/IIOP Acceptor.cpp
ACE wrappers/TAO/tao/Makefile.am
ACE wrappers/TAO/tao/Messaging/Messaging.pidl
ACE wrappers/TAO/tao/Messaging/MessagingA.cpp
ACE wrappers/TAO/tao/Messaging/MessagingC.cpp
ACE wrappers/TAO/tao/Messaging/MessagingC.h
ACE wrappers/TAO/tao/Messaging/MessagingC.inl
ACE wrappers/TAO/tao/Messaging/MessagingS.cpp
ACE wrappers/TAO/tao/Messaging/MessagingS.h
ACE wrappers/TAO/tao/Messaging/MessagingS T.cpp
ACE wrappers/TAO/tao/Messaging/MessagingS T.h
ACE wrappers/TAO/tao/Messaging/MessagingS T.inl
ACE wrappers/TAO/tao/Messaging/Messaging No ImplA.cpp

```
ACE wrappers/TAO/tao/Messaging/Messaging No ImplC.cpp
ACE wrappers/TAO/tao/Messaging/Messaging No ImplC.h
ACE wrappers/TAO/tao/Messaging/Messaging RT PolicyA.cpp
ACE wrappers/TAO/tao/Messaging/Messaging RT PolicyC.cpp
ACE wrappers/TAO/tao/Messaging/Messaging RT PolicyC.h
ACE wrappers/TAO/tao/Messaging/Messaging SyncScope PolicyA.cpp
ACE wrappers/TAO/tao/Messaging/Messaging SyncScope PolicyC.cpp
ACE wrappers/TAO/tao/Messaging/Messaging SyncScope PolicyC.h
ACE wrappers/TAO/tao/Messaging/TAO ExtA.cpp
ACE wrappers/TAO/tao/Messaging/TAO ExtC.cpp
ACE wrappers/TAO/tao/Messaging/TAO ExtC.h
ACE wrappers/TAO/tao/Messaging/diffs/Messaging.diff
ACE wrappers/TAO/tao/Messaging/diffs/Messaging No Impl.diff
ACE wrappers/TAO/tao/NVList.cpp
ACE wrappers/TAO/tao/ORB.h
ACE wrappers/TAO/tao/ORB Core.cpp
ACE wrappers/TAO/tao/ORB Core.h
ACE wrappers/TAO/tao/ORB Core.i
ACE wrappers/TAO/tao/PortableServer/Fixed Array SArgument T.h
ACE wrappers/TAO/tao/PortableServer/Fixed Array SArgument T.inl
ACE wrappers/TAO/tao/Profile Transport Resolver.cpp
ACE wrappers/TAO/tao/Resource Factory.cpp
ACE wrappers/TAO/tao/Resource Factory.h
ACE wrappers/TAO/tao/ServerRequestInterceptor Adapter Factory.h
ACE wrappers/TAO/tao/Thread Lane Resources Manager.cpp
ACE wrappers/TAO/tao/Transport.cpp
ACE wrappers/TAO/tao/Transport.h
ACE wrappers/TAO/tao/Transport.inl
ACE wrappers/TAO/tao/default resource.cpp
ACE wrappers/TAO/tao/default resource.h
ACE wrappers/TAO/tao/extra core.mpb
ACE wrappers/TAO/tao/orbconf.h
ACE wrappers/TAO/tao/params.cpp
ACE wrappers/TAO/tao/params.h
ACE wrappers/TAO/tao/params.i
ACE wrappers/TAO/tao/tao.mpc
ACE wrappers/TAO/tests/CodeSets/libs/IBM1047 ISO8859/Char IBM1047 ISO8859 Factory.h
ACE wrappers/TAO/tests/CodeSets/libs/IBM1047 ISO8859/CodeSets libs IMB1047 ISO8859.mpc
ACE wrappers/TAO/tests/CodeSets/libs/UCS4 UTF16/CodeSets libs UCS4 UTF16.mpc
ACE wrappers/TAO/tests/CodeSets/simple/CodeSets simple.mpc
ACE wrappers/TAO/tests/CodeSets/simple/wcs test.conf
ACE wrappers/THANKS
ACE wrappers/ace/Log Record.cpp
ACE wrappers/ace/OS NS stdio.cpp
ACE wrappers/bin/MakeProjectCreator/config/global.features
ACE wrappers/bin/MakeProjectCreator/config/taoclient.mpb
ACE wrappers/bin/MakeProjectCreator/config/taoserver.mpb
ACE wrappers/bin/make release
ACE wrappers/bin/tao other tests.lst
ACE wrappers/etc/ciao DAnCE.doxygen
ACE wrappers/protocols/ace/RMCast/Flow.cpp
```

Zoznam profilovaných súborov:

```
ACE wrappers/ace/Message Block.cpp
ACE wrappers/ace/CDR Stream.cpp
ACE wrappers/ace/OS NS Thread.cpp
ACE wrappers/ace/String Base.cpp
ACE wrappers/ace/TP Reactor.cpp
ACE wrappers/ace/Malloc T.cpp
ACE wrappers/ace/Token.cpp
ACE wrappers/ace/Handle Set.inl
ACE wrappers/ace/Countdown Time.cpp
ACE wrappers/ace/Handle Set.cpp
ACE wrappers/ace/Select Reactor T.cpp
ACE wrappers/ace/Malloc Allocator.cpp
ACE wrappers/ace/Select Reactor Base.cpp
ACE wrappers/ace/Message Block.inl
ACE wrappers/ace/Event Handler.cpp
ACE wrappers/ace/INET Addr.cpp
ACE wrappers/ace/Condition Thread Mutex.cpp
ACE wrappers/ace/OS NS string.inl
ACE wrappers/ace/OS NS Thread.inl
ACE wrappers/ace/ACE.cpp
ACE wrappers/ace/Reactor.cpp
ACE wrappers/ace/Recursive Thread Mutex.cpp
ACE wrappers/ace/Addr.cpp
ACE wrappers/ace/Sock Connect.cpp
ACE wrappers/ace/CDR Stream.inl
ACE wrappers/ace/Array Base.cpp
ACE wrappers/ace/CDR Base.cpp
```

ACE wrappers/ace/Atomic Op.cpp
ACE wrappers/ace/Guard T.inl
ACE wrappers/ace/Thread Mutex.cpp
ACE wrappers/ace/OS NS stdlib.cpp
ACE wrappers/ace/Time Value.cpp
ACE wrappers/ace/Timer Queue T.cpp
ACE wrappers/ace/Object Manager Base.cpp
ACE wrappers/ace/Timer Heap T.cpp
ACE wrappers/ace/OS NS string.cpp
ACE wrappers/ace/Array Base.inl
ACE wrappers/ace/Service Repository.cpp
ACE wrappers/ace/Lock.cpp
ACE wrappers/ace/TP Reactor.inl
ACE wrappers/ace/OS NS stdio.cpp
ACE wrappers/ace/Time Value.inl
ACE wrappers/ace/OS NS sys socket.inl
ACE wrappers/ace/INET Addr.inl
ACE wrappers/ace/OS NS sys select.inl
ACE wrappers/ace/Functor.cpp
ACE wrappers/ace/ACE.inl
ACE wrappers/ace/OS NS sys uio.inl
ACE wrappers/ace/OS NS unistd.inl
ACE wrappers/ace/Dynamic Service Base.cpp
ACE wrappers/ace/Token.inl
ACE wrappers/ace/Addr.inl
ACE wrappers/ace/OS NS signal.inl
ACE wrappers/ace/Atomic Op.inl
ACE wrappers/ace/Service Object.inl
ACE wrappers/ace/CDR Base.inl
ACE wrappers/ace/Auto Ptr.inl
ACE wrappers/ace/Thread Mutex.inl
ACE wrappers/ace/OS NS arpa inet.inl
ACE wrappers/ace/OS NS stropts.inl
ACE wrappers/ace/Condition Thread Mutex.inl
ACE wrappers/ace/Pipe.inl
ACE wrappers/ace/Signal.inl
ACE wrappers/ace/Malloc.inl
ACE wrappers/ace/Unbounded Set.cpp
ACE wrappers/ace/Auto Ptr.h
ACE wrappers/ace/Service Config.cpp
ACE wrappers/ace/Local Memory Pool.cpp
ACE wrappers/ace/Service Types.cpp
ACE wrappers/ace/Node.cpp
ACE wrappers/ace/Synch Options.cpp
ACE wrappers/ace/Svc Conf l.cpp
ACE wrappers/ace/Svc Conf Lexer Guard.cpp
ACE wrappers/ace/Singleton.cpp
ACE wrappers/ace/Select Reactor Base.inl
ACE wrappers/ace/RW Thread Mutex.cpp
ACE wrappers/ace/Log Msg.cpp
ACE wrappers/ace/DLL Manager.cpp
ACE wrappers/ace/Containers T.cpp
ACE wrappers/ace/Argv Type Converter.cpp
ACE wrappers/ace/Arg Shifter.cpp
ACE wrappers/TAO/tao/Sequence.i
ACE wrappers/TAO/tao/Sequence.cpp
ACE wrappers/TAO/tao/Transport.cpp
ACE wrappers/TAO/tao/ORB Core.cpp
ACE wrappers/TAO/tao/Sequence T.cpp
ACE wrappers/TAO/tao/Object.cpp
ACE wrappers/TAO/tao/GIOP Message Base.cpp
ACE wrappers/ace/Lock Adapter T.cpp
ACE wrappers/TAO/tao/Thread Lane Resources.cpp
ACE wrappers/ace/RB Tree.cpp
ACE wrappers/TAO/tao/IOP IORC.cpp
ACE wrappers/TAO/tao/GIOP Message State.cpp
ACE wrappers/TAO/tao/Profile.cpp
ACE wrappers/TAO/tao/ObjectKey Table.cpp
ACE wrappers/TAO/tao/MProfile.cpp
ACE wrappers/TAO/tao/OctetSeqC.cpp
ACE wrappers/TAO/tao/Environment.cpp
ACE wrappers/TAO/tao/CONV FRAMEC.cpp
ACE wrappers/TAO/tao/Stub.cpp
ACE wrappers/ace/Hash Map Manager T.cpp
ACE wrappers/TAO/tao/Tagged Components.cpp
ACE wrappers/TAO/tao/Synch Queued Message.cpp
ACE wrappers/TAO/tao/IIOP Profile.cpp
ACE wrappers/TAO/tao/Object KeyC.cpp
ACE wrappers/TAO/tao/Invocation Adapter.cpp
ACE wrappers/TAO/tao/Synch Invocation.cpp
ACE wrappers/TAO/tao/Objref VarOut T.cpp

ACE wrappers/TAO/tao/Transport Cache Manager.cpp
ACE wrappers/ace/CDR Stream.inl
ACE wrappers/TAO/tao/IIOP Transport.cpp
ACE wrappers/TAO/tao/CORBA String.cpp
ACE wrappers/TAO/tao/LF Invocation Event.cpp
ACE wrappers/TAO/tao/GIOP Message Generator Parser 12.cpp
ACE wrappers/ace/Guard T.inl
ACE wrappers/TAO/tao/default resource.cpp
ACE wrappers/TAO/tao/TAO Singleton.cpp
ACE wrappers/ace/TSS T.cpp
ACE wrappers/TAO/tao/TAO Server Request.cpp
ACE wrappers/TAO/tao/IIOP Endpoint.cpp
ACE wrappers/TAO/tao/Connection Handler.cpp
ACE wrappers/TAO/tao/Argument.cpp
ACE wrappers/TAO/tao/Leader Follower.cpp
ACE wrappers/TAO/tao/Queued Message.cpp
ACE wrappers/TAO/tao/Synch Reply Dispatcher.cpp
ACE wrappers/TAO/tao/Wait On Leader Follower.cpp
ACE wrappers/TAO/tao/Muxed TMS.cpp
ACE wrappers/TAO/tao/Profile Transport Resolver.cpp
ACE wrappers/TAO/tao/Remote Invocation.cpp
ACE wrappers/TAO/tao/IIOP Connection Handler.cpp
ACE wrappers/ace/RB Tree.inl
ACE wrappers/TAO/tao/Default Thread Lane Resources Manager.cpp
ACE wrappers/ace/Message Block.inl
ACE wrappers/TAO/tao/ORB Core.i
ACE wrappers/TAO/tao/Refcounted ObjectKey.cpp
ACE wrappers/TAO/tao/LF Event.cpp
ACE wrappers/TAO/tao/Invocation Base.cpp
ACE wrappers/TAO/tao/Connector Registry.cpp
ACE wrappers/TAO/tao/PolicyC.cpp
ACE wrappers/TAO/tao/LF Strategy Complete.cpp
ACE wrappers/TAO/tao/default client.cpp
ACE wrappers/TAO/tao/IIOP Connector.cpp
ACE wrappers/TAO/tao/TSS Resources.cpp
ACE wrappers/TAO/tao/IIOP Acceptor.cpp
ACE wrappers/TAO/tao/Default Stub Factory.cpp
ACE wrappers/TAO/tao/Object.i
ACE wrappers/TAO/tao/Base Transport Property.cpp
ACE wrappers/TAO/tao/GIOP Message Generator Parser.cpp
ACE wrappers/ace/Svc Handler.cpp
ACE wrappers/TAO/tao/operation details.cpp
ACE wrappers/TAO/tao/Bind Dispatcher Guard.cpp
ACE wrappers/TAO/tao/Profile Transport Resolver.inl
ACE wrappers/ace/OS NS Thread.inl
ACE wrappers/TAO/tao/PICurrent Impl.cpp
ACE wrappers/TAO/tao/CDR.i
ACE wrappers/TAO/tao/params.cpp
ACE wrappers/TAO/tao/Policy Current.cpp
ACE wrappers/TAO/tao/Tagged Profile.cpp
ACE wrappers/TAO/tao/Leader Follower.i
ACE wrappers/TAO/tao/PICurrent Copy Callback.cpp
ACE wrappers/TAO/tao/Incoming Message Queue.cpp
ACE wrappers/TAO/tao/Reply Dispatcher.cpp
ACE wrappers/TAO/tao/Request Dispatcher.cpp
ACE wrappers/TAO/tao/Default Protocols Hooks.cpp
ACE wrappers/TAO/tao/CDR.cpp
ACE wrappers/TAO/tao/Cache Entries.cpp
ACE wrappers/TAO/tao/Endpoint.i
ACE wrappers/TAO/tao/Adapter Registry.cpp
ACE wrappers/TAO/tao/ORB.i
ACE wrappers/ace/Vector T.cpp
ACE wrappers/TAO/tao/Resume Handle.cpp
ACE wrappers/TAO/tao/ORB Table.cpp
ACE wrappers/ace/Vector T.inl
ACE wrappers/TAO/tao/Service Context.cpp
ACE wrappers/ace/Array Base.cpp
ACE wrappers/TAO/tao/Pluggable Messaging Utils.cpp
ACE wrappers/TAO/tao/MProfile.i
ACE wrappers/ace/Containers T.inl
ACE wrappers/TAO/tao/Invocation Endpoint Selectors.cpp
ACE wrappers/TAO/tao/CORBA String.inl
ACE wrappers/TAO/tao/Stub.i
ACE wrappers/TAO/tao/TAO Server Request.i
ACE wrappers/TAO/tao/Transport Connector.cpp
ACE wrappers/TAO/tao/Default Endpoint Selector Factory.cpp
ACE wrappers/TAO/tao/Sequence T.i
ACE wrappers/TAO/tao/Thread Lane Resources Manager.cpp
ACE wrappers/TAO/tao/LRU Connection Purging Strategy.cpp
ACE wrappers/ace/Reverse Lock T.cpp
ACE wrappers/TAO/tao/ORB Core Auto Ptr.cpp

ACE wrappers/TAO/tao/Policy Set.cpp
ACE wrappers/ace/OS NS string.inl
ACE wrappers/TAO/tao/Wait Strategy.cpp
ACE wrappers/TAO/tao/operation details.i
ACE wrappers/TAO/tao/Policy ForwardC.cpp
ACE wrappers/TAO/tao/Abstract Servant Base.cpp
ACE wrappers/TAO/tao/LocalObject.cpp
ACE wrappers/TAO/tao/Acceptor Registry.cpp
ACE wrappers/TAO/tao/Transport.inl
ACE wrappers/TAO/tao/Pseudo VarOut T.inl
ACE wrappers/TAO/tao/Endpoint.cpp
ACE wrappers/TAO/tao/Transport Descriptor Interface.cpp
ACE wrappers/TAO/tao/Managed Types.i
ACE wrappers/TAO/tao/params.i
ACE wrappers/ace/IPC SAP.inl
ACE wrappers/TAO/tao/Profile.i
ACE wrappers/ace/Array Base.inl
ACE wrappers/TAO/tao/ORB Core Auto Ptr.inl
ACE wrappers/ace/Thread Mutex.inl
ACE wrappers/TAO/tao/Tagged Components.i
ACE wrappers/TAO/tao/target specification.i
ACE wrappers/TAO/tao/Acceptor Filter.cpp
ACE wrappers/TAO/tao/Cache Entries.inl
ACE wrappers/ace/String Base.inl
ACE wrappers/ace/Hash Map Manager T.inl
ACE wrappers/ace/Array Map.inl
ACE wrappers/ace/RB Tree.h
ACE wrappers/TAO/tao/Pluggable Messaging Utils.i
ACE wrappers/TAO/tao/Connection Handler.inl
ACE wrappers/TAO/tao/Remote Object Proxy Broker.cpp
ACE wrappers/ace/INET Addr.inl
ACE wrappers/TAO/tao/VarOut T.inl
ACE wrappers/TAO/tao/Acceptor Registry.i
ACE wrappers/TAO/tao/Incoming Message Queue.inl
ACE wrappers/TAO/tao/Transport Descriptor Interface.inl
ACE wrappers/TAO/tao/Refcounted ObjectKey.inl
ACE wrappers/TAO/tao/Resume Handle.inl
ACE wrappers/TAO/tao/Tagged Profile.i
ACE wrappers/ace/Addr.inl
ACE wrappers/TAO/tao/LF Event.inl
ACE wrappers/TAO/tao/Seq Var T.inl
ACE wrappers/TAO/tao/Service Context.inl
ACE wrappers/TAO/tao/GIOP Message Version.inl
ACE wrappers/TAO/tao/Transport Connector.inl
ACE wrappers/TAO/tao/LF Event Loop Thread Helper.inl
ACE wrappers/ace/Thread.inl
ACE wrappers/TAO/tao/PICurrent Impl.inl
ACE wrappers/TAO/tao/Policy Current Impl.inl
ACE wrappers/TAO/tao/GIOP Message State.inl
ACE wrappers/ace/SOCK IO.inl
ACE wrappers/TAO/tao/Bind Dispatcher Guard.i
ACE wrappers/ace/Time Value.inl
ACE wrappers/TAO/tao/Connector Registry.i
ACE wrappers/TAO/tao/Reply Dispatcher.i
ACE wrappers/TAO/tao/Transport Cache Manager.inl
ACE wrappers/TAO/tao/GIOP Message Generator Parser Impl.inl
ACE wrappers/TAO/tao/Fault Tolerance Service.i
ACE wrappers/TAO/tao/Transport Acceptor.inl
ACE wrappers/TAO/tao/Objref VarOut T.h
ACE wrappers/TAO/tao/Policy Manager.i
ACE wrappers/TAO/tao/ORB Table.inl
ACE wrappers/TAO/tao/ORB Table.h
ACE wrappers/TAO/tao/IIOP Endpoint.i
ACE wrappers/ace/Functor.inl
ACE wrappers/TAO/tao/Base Transport Property.inl
ACE wrappers/ace/Unbounded Set.cpp
ACE wrappers/ace/Intrusive List Node.cpp
ACE wrappers/ace/Dynamic Service.cpp
ACE wrappers/ace/Containers T.cpp
ACE wrappers/ace/Connector.cpp
ACE wrappers/ace/Auto Ptr.inl
ACE wrappers/TAO/tao/default server.cpp
ACE wrappers/TAO/tao/TAO Singleton.inl
ACE wrappers/TAO/tao/Protocol Factory.cpp
ACE wrappers/TAO/tao/ParameterModeA.cpp
ACE wrappers/TAO/tao/ORB.cpp
ACE wrappers/TAO/tao/Messaging PolicyValueA.cpp
ACE wrappers/TAO/tao/LongDoubleSeqA.cpp
ACE wrappers/TAO/tao/IIOP Factory.cpp
ACE wrappers/TAO/tao/IIOPA.cpp
ACE wrappers/TAO/tao/DoubleSeqA.cpp

```
ACE wrappers/TAO/tao/Codeset Manager Factory Base.cpp
ACE wrappers/ace/Hash Map Manager T.inl
ACE wrappers/ace/Hash Map Manager T.cpp
ACE wrappers/ace/Map T.cpp
ACE wrappers/ace/OS NS string.inl
ACE wrappers/ace/Map Manager.inl
ACE wrappers/TAO/tao/Sequence.i
ACE wrappers/TAO/tao/Seq Var T.inl
ACE wrappers/TAO/tao/Pseudo VarOut T.inl
ACE wrappers/ace/Atomic Op T.cpp
ACE wrappers/TAO/tao/Object.i
ACE wrappers/ace/Map Manager.cpp
ACE wrappers/TAO/tao/TAO Server Request.i
ACE wrappers/ace/Dynamic Service.cpp
ACE wrappers/TAO/tao/Objref VarOut T.cpp
ACE wrappers/ace/Functor T.inl
ACE wrappers/TAO/tao/Seq Out T.inl
ACE wrappers/ace/Active Map Manager T.inl
ACE wrappers/ace/Guard T.inl
ACE wrappers/TAO/tao/PortableServer/POA Current Impl.inl
ACE wrappers/TAO/tao/MProfile.i
ACE wrappers/TAO/tao/Acceptor Registry.i
ACE wrappers/TAO/tao/Sequence T.i
ACE wrappers/ace/Pair T.inl
ACE wrappers/TAO/tao/ORB.i
ACE wrappers/ace/Active Map Manager.inl
ACE wrappers/ace/OS NS Thread.inl
ACE wrappers/ace/String Base.inl
ACE wrappers/TAO/tao/ORB Core.i
ACE wrappers/TAO/tao/Objref VarOut T.h
ACE wrappers/ace/Thread Mutex.inl
ACE wrappers/TAO/tao/Stub.i
ACE wrappers/TAO/tao/ORB Core Auto Ptr.inl
ACE wrappers/ace/Array Base.inl
ACE wrappers/TAO/orbsvcs/orbsvcs/CosNamingC.cpp
ACE wrappers/TAO/tao/Object.i
ACE wrappers/TAO/tao/Objref VarOut T.h
ACE wrappers/TAO/tao/Valuetype/Valuetype Adapter Impl.h
ACE wrappers/TAO/tao/Pseudo VarOut T.cpp
ACE wrappers/TAO/tao/Object.i
ACE wrappers/TAO/tao/Objref VarOut T.cpp
ACE wrappers/ace/String Base.cpp
ACE wrappers/ace/Hash Map Manager T.cpp
ACE wrappers/TAO/tao/Sequence T.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/CosNamingS.cpp
```

8.3 Porovanie verzií 20051013-175104 a 20051014-080512

Zoznam menených súborov (výstup z diffu):

```
ACE wrappers/ChangeLog
ACE wrappers/TAO/CIAO/ChangeLog
ACE wrappers/TAO/CIAO/examples/BasicSP/descriptors/basicNodeDaemon.pl
ACE wrappers/TAO/CIAO/tools/Config Handlers/RT-CCM/CIAOServerResources.cpp
ACE wrappers/TAO/CIAO/tools/Config Handlers/RT-CCM/CIAOServerResources.hpp
ACE wrappers/TAO/CIAO/tools/Config Handlers/RT-CCM/TPL Handler.cpp
ACE wrappers/TAO/CIAO/tools/Config Handlers/XMLSchema/Types.hpp
ACE wrappers/TAO/CIAO/tools/Config Handlers/XSCRT/Elements.hpp
ACE wrappers/TAO/ChangeLog
ACE wrappers/TAO/orbsvcs/examples/ImR/Combined Service/ImR Combined Service.mpc
ACE wrappers/TAO/orbsvcs/orbsvcs/CosEvent/CEC DynamicImplementation.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/CosEvent/CEC Event Loader.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/CosEvent/CEC ProxyPushSupplier.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/CosEvent/CEC Reactive ConsumerControl.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/CosEvent/CEC Reactive SupplierControl.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/CosEvent/CEC TypedEventChannel.cpp
ACE wrappers/TAO/orbsvcs/orbsvcs/CosEvent/CEC TypedEventChannel.h
ACE wrappers/TAO/orbsvcs/orbsvcs/CosEvent/CEC TypedEventChannel.i
ACE wrappers/TAO/orbsvcs/orbsvcs/CosEvent/CEC TypedProxyPushConsumer.cpp
ACE wrappers/TAO/tao/ORB Core.cpp
ACE wrappers/TAO/tao/PI/ClientRequestInterceptor Adapter Impl.cpp
ACE wrappers/TAO/tao/PI/ClientRequestInterceptor Adapter Impl.h
ACE wrappers/TAO/tao/PI/ClientRequestInterceptor Factory Impl.cpp
ACE wrappers/TAO/tao/Pseudo VarOut T.h
```

ACE wrappers/TAO/tao/Pseudo VarOut T.inl
ACE wrappers/TAO/tao/SystemException.cpp
ACE wrappers/TAO/tao/SystemException.inl
ACE wrappers/TAO/tao/TSS Resources.cpp
ACE wrappers/TAO/tao/TSS Resources.h
ACE wrappers/TAO/tao/UserException.cpp
ACE wrappers/TAO/tao/UserException.inl
ACE wrappers/ace/RB Tree.cpp
ACE wrappers/ace/RB Tree.h
ACE wrappers/ace/TTY IO.cpp
ACE wrappers/ace/config-g++-common.h
ACE wrappers/ace/config-win32-common.h
ACE wrappers/protocols/ace/HTBP/HTBP Session.cpp

Zoznam profilovaných súborov:

ACE wrappers/ace/Message Block.cpp
ACE wrappers/ace/CDR Stream.cpp
ACE wrappers/ace/OS NS Thread.cpp
ACE wrappers/ace/TP Reactor.cpp
ACE wrappers/ace/Malloc T.cpp
ACE wrappers/ace/String Base.cpp
ACE wrappers/ace/Handle Set.inl
ACE wrappers/ace/Message Block.inl
ACE wrappers/ace/Handle Set.cpp
ACE wrappers/ace/Malloc Allocator.cpp
ACE wrappers/ace/Countdown Time.cpp
ACE wrappers/ace/Token.cpp
ACE wrappers/ace/Select Reactor Base.cpp
ACE wrappers/ace/Select Reactor T.cpp
ACE wrappers/ace/Event Handler.cpp
ACE wrappers/ace/Reactor.cpp
ACE wrappers/ace/Condition Thread Mutex.cpp
ACE wrappers/ace/INET Addr.cpp
ACE wrappers/ace/OS NS string.inl
ACE wrappers/ace/OS NS Thread.inl
ACE wrappers/ace/Recursive Thread Mutex.cpp
ACE wrappers/ace/ACE.cpp
ACE wrappers/ace/CDR Base.cpp
ACE wrappers/ace/Object Manager Base.cpp
ACE wrappers/ace/Addr.cpp
ACE wrappers/ace/Sock Connect.cpp
ACE wrappers/ace/Array Base.cpp
ACE wrappers/ace/CDR Stream.inl
ACE wrappers/ace/Atomic Op.cpp
ACE wrappers/ace/Array Base.inl
ACE wrappers/ace/Time Value.cpp
ACE wrappers/ace/Guard T.inl
ACE wrappers/ace/TP Reactor.inl
ACE wrappers/ace/Timer Heap T.cpp
ACE wrappers/ace/Timer Queue T.cpp
ACE wrappers/ace/OS NS stdlib.cpp
ACE wrappers/ace/Service Repository.cpp
ACE wrappers/ace/INET Addr.inl
ACE wrappers/ace/Lock.cpp
ACE wrappers/ace/OS NS sys socket.inl
ACE wrappers/ace/Thread Mutex.cpp
ACE wrappers/ace/OS NS string.cpp
ACE wrappers/ace/Reactor Token T.cpp
ACE wrappers/ace/ACE.inl
ACE wrappers/ace/OS NS unistd.inl
ACE wrappers/ace/OS NS sys select.inl
ACE wrappers/ace/Time Value.inl
ACE wrappers/ace/OS NS stropts.inl
ACE wrappers/ace/Dynamic Service Base.cpp
ACE wrappers/ace/Token.inl
ACE wrappers/ace/Auto Ptr.inl
ACE wrappers/ace/Condition Thread Mutex.inl
ACE wrappers/ace/Addr.inl
ACE wrappers/ace/OS NS sys uio.inl
ACE wrappers/ace/Service Object.inl
ACE wrappers/ace/Atomic Op.inl
ACE wrappers/ace/OS NS signal.inl
ACE wrappers/ace/Thread Mutex.inl
ACE wrappers/ace/CDR Base.inl
ACE wrappers/ace/OS NS arpa inet.inl
ACE wrappers/ace/Signal.inl
ACE wrappers/ace/Pipe.inl
ACE wrappers/ace/Malloc.inl
ACE wrappers/ace/Unbounded Set.cpp
ACE wrappers/ace/Auto Ptr.h

ACE wrappers/ace/Service Types.cpp
ACE wrappers/ace/Node.cpp
ACE wrappers/ace/Service Config.cpp
ACE wrappers/ace/Local Memory Pool.cpp
ACE wrappers/ace/Select Reactor T.inl
ACE wrappers/ace/Object Manager.cpp
ACE wrappers/ace/OS NS fcntl.inl
ACE wrappers/ace/Arg Shifter.cpp
ACE wrappers/ace/Shared Object.cpp
ACE wrappers/ace/SOCK Connector.cpp
ACE wrappers/ace/Reactor Timer Interface.cpp
ACE wrappers/ace/Dynamic.cpp
ACE wrappers/ace/DLL.cpp
ACE wrappers/ace/Cleanup.cpp
ACE wrappers/ace/ARGV.cpp
ACE wrappers/TAO/tao/Sequence.i
ACE wrappers/TAO/tao/Sequence.cpp
ACE wrappers/TAO/tao/Transport.cpp
ACE wrappers/TAO/tao/ORB Core.cpp
ACE wrappers/TAO/tao/Object.cpp
ACE wrappers/TAO/tao/Sequence T.cpp
ACE wrappers/TAO/tao/GIOP Message Base.cpp
ACE wrappers/ace/Lock Adapter T.cpp
ACE wrappers/TAO/tao/GIOP Message State.cpp
ACE wrappers/TAO/tao/Thread Lane Resources.cpp
ACE wrappers/TAO/tao/IOP IORC.cpp
ACE wrappers/ace/RB Tree.cpp
ACE wrappers/TAO/tao/Profile.cpp
ACE wrappers/TAO/tao/MProfile.cpp
ACE wrappers/TAO/tao/Stub.cpp
ACE wrappers/ace/CDR Stream.inl
ACE wrappers/TAO/tao/Invocation Adapter.cpp
ACE wrappers/TAO/tao/OctetSeqC.cpp
ACE wrappers/TAO/tao/Tagged Components.cpp
ACE wrappers/TAO/tao/CONV FRAMEC.cpp
ACE wrappers/TAO/tao/ObjectKey Table.cpp
ACE wrappers/ace/Guard T.inl
ACE wrappers/TAO/tao/IIOP Profile.cpp
ACE wrappers/TAO/tao/Transport Cache Manager.cpp
ACE wrappers/TAO/tao/Environment.cpp
ACE wrappers/TAO/tao/Object KeyC.cpp
ACE wrappers/TAO/tao/IIOP Endpoint.cpp
ACE wrappers/TAO/tao/IIOP Transport.cpp
ACE wrappers/TAO/tao/GIOP Message Generator Parser 12.cpp
ACE wrappers/TAO/tao/Synch Queued Message.cpp
ACE wrappers/TAO/tao/Objref VarOut T.cpp
ACE wrappers/ace/Hash Map Manager T.cpp
ACE wrappers/TAO/tao/Synch Invocation.cpp
ACE wrappers/TAO/tao/Connection Handler.cpp
ACE wrappers/TAO/tao/TAO Singleton.cpp
ACE wrappers/TAO/tao/LF Invocation Event.cpp
ACE wrappers/ace/TSS T.cpp
ACE wrappers/TAO/tao/Muxed TMS.cpp
ACE wrappers/TAO/tao/ORB Core.i
ACE wrappers/TAO/tao/default resource.cpp
ACE wrappers/TAO/tao/CORBA String.cpp
ACE wrappers/TAO/tao/Queued Message.cpp
ACE wrappers/TAO/tao/Wait On Leader Follower.cpp
ACE wrappers/TAO/tao/Leader Follower.cpp
ACE wrappers/TAO/tao/Argument.cpp
ACE wrappers/TAO/tao/Remote Invocation.cpp
ACE wrappers/TAO/tao/Profile Transport Resolver.cpp
ACE wrappers/TAO/tao/TAO Server Request.cpp
ACE wrappers/TAO/tao/Connector Registry.cpp
ACE wrappers/TAO/tao/IIOP Connection Handler.cpp
ACE wrappers/TAO/tao/LF Event.cpp
ACE wrappers/TAO/tao/Synch Reply Dispatcher.cpp
ACE wrappers/TAO/tao/Object.i
ACE wrappers/ace/RB Tree.inl
ACE wrappers/TAO/tao/Default Thread Lane Resources Manager.cpp
ACE wrappers/TAO/tao/Invocation Base.cpp
ACE wrappers/TAO/tao/LF Strategy Complete.cpp
ACE wrappers/TAO/tao/Refcounted ObjectKey.cpp
ACE wrappers/ace/Message Block.inl
ACE wrappers/TAO/tao/default client.cpp
ACE wrappers/TAO/tao/Base Transport Property.cpp
ACE wrappers/TAO/tao/operation details.cpp
ACE wrappers/ace/Atomic Op T.cpp
ACE wrappers/TAO/tao/IIOP Acceptor.cpp
ACE wrappers/ace/Svc Handler.cpp
ACE wrappers/ace/OS NS Thread.inl

ACE wrappers/TAO/tao/TSS Resources.cpp
ACE wrappers/TAO/tao/IIOP Connector.cpp
ACE wrappers/TAO/tao/Reply Dispatcher.cpp
ACE wrappers/TAO/tao/Tagged Profile.cpp
ACE wrappers/TAO/tao/CDR.cpp
ACE wrappers/TAO/tao/CDR.i
ACE wrappers/TAO/tao/MProfile.i
ACE wrappers/TAO/tao/Profile Transport Resolver.inl
ACE wrappers/TAO/tao/Incoming Message Queue.cpp
ACE wrappers/TAO/tao/CORBA String.inl
ACE wrappers/TAO/tao/Bind Dispatcher Guard.cpp
ACE wrappers/TAO/tao/Service Context.cpp
ACE wrappers/TAO/tao/LRU Connection Purging Strategy.cpp
ACE wrappers/TAO/tao/operation details.i
ACE wrappers/TAO/tao/Default Stub Factory.cpp
ACE wrappers/TAO/tao/Cache Entries.cpp
ACE wrappers/TAO/tao/Invocation Endpoint Selectors.cpp
ACE wrappers/TAO/tao/PolicyC.cpp
ACE wrappers/ace/OS NS string.inl
ACE wrappers/TAO/tao/Policy Current.cpp
ACE wrappers/TAO/tao/GIOP Message Generator Parser.cpp
ACE wrappers/TAO/tao/Resume Handle.cpp
ACE wrappers/ace/Vector T.inl
ACE wrappers/TAO/tao/Request Dispatcher.cpp
ACE wrappers/TAO/tao/ORB.i
ACE wrappers/TAO/tao/Wait Strategy.cpp
ACE wrappers/TAO/tao/Adapter Registry.cpp
ACE wrappers/TAO/tao/TAO Server Request.i
ACE wrappers/TAO/tao/Default Endpoint Selector Factory.cpp
ACE wrappers/TAO/tao/Leader Follower.i
ACE wrappers/TAO/tao/Transport Connector.cpp
ACE wrappers/TAO/tao/Sequence T.i
ACE wrappers/TAO/tao/Acceptor Registry.cpp
ACE wrappers/TAO/tao/Pluggable Messaging Utils.cpp
ACE wrappers/ace/Vector T.cpp
ACE wrappers/TAO/tao/Endpoint.i
ACE wrappers/TAO/tao/Policy ForwardC.cpp
ACE wrappers/TAO/tao/params.cpp
ACE wrappers/TAO/tao/Stub.i
ACE wrappers/TAO/tao/ORB Core Auto Ptr.cpp
ACE wrappers/TAO/tao/Policy Set.cpp
ACE wrappers/TAO/tao/Thread Lane Resources Manager.cpp
ACE wrappers/TAO/tao/Abstract Servant Base.cpp
ACE wrappers/TAO/tao/Pseudo VarOut T.inl
ACE wrappers/TAO/tao/Default Protocols Hooks.cpp
ACE wrappers/TAO/tao/Transport Descriptor Interface.cpp
ACE wrappers/ace/Hash Map Manager T.inl
ACE wrappers/TAO/tao/ORB Table.cpp
ACE wrappers/TAO/tao/Managed Types.i
ACE wrappers/ace/Reverse Lock T.cpp
ACE wrappers/TAO/tao/target specification.i
ACE wrappers/ace/IPC SAP.inl
ACE wrappers/ace/Containers T.inl
ACE wrappers/ace/Thread Mutex.inl
ACE wrappers/TAO/tao/Connection Handler.inl
ACE wrappers/TAO/tao/Endpoint.cpp
ACE wrappers/TAO/tao/Profile.i
ACE wrappers/TAO/tao/LocalObject.cpp
ACE wrappers/TAO/tao/Policy Current Impl.inl
ACE wrappers/TAO/tao/Tagged Components.i
ACE wrappers/TAO/tao/Acceptor Filter.cpp
ACE wrappers/TAO/tao/Remote Object Proxy Broker.cpp
ACE wrappers/TAO/tao/Tagged Profile.i
ACE wrappers/ace/Array Map.inl
ACE wrappers/TAO/tao/params.i
ACE wrappers/TAO/tao/Cache Entries.inl
ACE wrappers/ace/INET Addr.inl
ACE wrappers/TAO/tao/Seq Var T.inl
ACE wrappers/TAO/tao/Transport.inl
ACE wrappers/TAO/tao/ORB Core Auto Ptr.inl
ACE wrappers/ace/Array Base.inl
ACE wrappers/TAO/tao/VarOut T.inl
ACE wrappers/TAO/tao/Refcounted ObjectKey.inl
ACE wrappers/TAO/tao/Transport Descriptor Interface.inl
ACE wrappers/TAO/tao/Resume Handle.inl
ACE wrappers/TAO/tao/GIOP Message Generator Parser Impl.inl
ACE wrappers/TAO/tao/GIOP Message Version.inl
ACE wrappers/TAO/tao/LF Event Loop Thread Helper.inl
ACE wrappers/TAO/tao/LF Event.inl
ACE wrappers/ace/Thread.inl
ACE wrappers/TAO/tao/Service Context.inl

ACE wrappers/TAO/tao/Transport Connector.inl
ACE wrappers/TAO/tao/Fault Tolerance Service.i
ACE wrappers/TAO/tao/Pluggable Messaging Utils.i
ACE wrappers/TAO/tao/Incoming Message Queue.inl
ACE wrappers/ace/Addr.inl
ACE wrappers/ace/Time Value.inl
ACE wrappers/TAO/tao/Connector Registry.i
ACE wrappers/TAO/tao/Transport Cache Manager.inl
ACE wrappers/TAO/tao/GIOP Message State.inl
ACE wrappers/TAO/tao/Acceptor Registry.i
ACE wrappers/TAO/tao/Bind Dispatcher Guard.i
ACE wrappers/ace/SOCK IO.inl
ACE wrappers/ace/String Base.inl
ACE wrappers/TAO/tao/Objref VarOut T.h
ACE wrappers/TAO/tao/Policy Manager.i
ACE wrappers/TAO/tao/ORB Table.inl
ACE wrappers/TAO/tao/Transport Acceptor.inl
ACE wrappers/TAO/tao/Reply Dispatcher.i
ACE wrappers/TAO/tao/ORB Table.h
ACE wrappers/ace/Functor.inl
ACE wrappers/TAO/tao/IIOP Endpoint.i
ACE wrappers/TAO/tao/Base Transport Property.inl
ACE wrappers/ace/Atomic Op T.inl
ACE wrappers/ace/Dynamic Service.cpp
ACE wrappers/ace/Malloc T.cpp
ACE wrappers/ace/Connector.cpp
ACE wrappers/TAO/tao/Server Strategy Factory.cpp
ACE wrappers/TAO/tao/LF Follower.cpp
ACE wrappers/TAO/tao/FILE Parser.cpp
ACE wrappers/TAO/tao/Exception.cpp
ACE wrappers/ace/Hash Map Manager T.inl
ACE wrappers/ace/Hash Map Manager T.cpp
ACE wrappers/ace/Map T.cpp
ACE wrappers/ace/OS NS string.inl
ACE wrappers/TAO/tao/Pseudo VarOut T.inl
ACE wrappers/ace/Map Manager.inl
ACE wrappers/TAO/tao/Object.i
ACE wrappers/TAO/tao/Sequence.i
ACE wrappers/ace/Map Manager.cpp
ACE wrappers/TAO/tao/Seq Var T.inl
ACE wrappers/ace/Active Map Manager T.inl
ACE wrappers/TAO/tao/TAO Server Request.i
ACE wrappers/ace/Dynamic Service.cpp
ACE wrappers/ace/Functor T.inl
ACE wrappers/TAO/tao/Objref VarOut T.cpp
ACE wrappers/TAO/tao/Seq Out T.inl
ACE wrappers/TAO/tao/PortableServer/POA Current Impl.inl
ACE wrappers/ace/Guard T.inl
ACE wrappers/TAO/tao/MProfile.i
ACE wrappers/ace/OS NS Thread.inl
ACE wrappers/TAO/tao/Sequence T.i
ACE wrappers/ace/Active Map Manager.inl
ACE wrappers/TAO/tao/Objref VarOut T.h
ACE wrappers/TAO/tao/ORB.i
ACE wrappers/ace/Pair T.inl
ACE wrappers/TAO/tao/ORB Core.i
ACE wrappers/ace/Atomic Op.inl
ACE wrappers/TAO/tao/SystemException.inl
ACE wrappers/TAO/tao/Acceptor Registry.i
ACE wrappers/ace/String Base.inl
ACE wrappers/ace/Thread Mutex.inl
ACE wrappers/TAO/tao/Stub.i
ACE wrappers/TAO/tao/ORB Core Auto Ptr.inl
ACE wrappers/ace/Map T.inl
ACE wrappers/TAO/tao/AnyTypeCode/TypeCode Base Attributes.inl
ACE wrappers/TAO/tao/AnyTypeCode/TypeCode.inl
ACE wrappers/TAO/tao/AnyTypeCode/Any Dual Impl T.cpp
ACE wrappers/TAO/tao/Object.i
ACE wrappers/TAO/tao/Objref VarOut T.h
ACE wrappers/ace/Atomic Op T.cpp
ACE wrappers/TAO/tao/AnyTypeCode/Objref TypeCode Static.inl
ACE wrappers/TAO/tao/Object.i
ACE wrappers/TAO/tao/LocalObject.i
ACE wrappers/ace/String Base.cpp
ACE wrappers/ace/Hash Map Manager T.cpp
ACE wrappers/ace/Hash Map Manager T.cpp