GNU GProf i GCov

przygotował: Krzysztof Jurczuk
Politechnika Białostocka
Wydział Informatyki Katedra Oprogramowania
ul. Wiejska 45A
15-351 Białystok

Streszczenie: Dokument zawiera podstawowe informacje na temat narzędzia GNU GProf i GCov. Po krótkim wprowadzeniu do tematyki profilowania aplikacji, została w nim przedstawiona zasada pracy z ww. narzędziami (kompilacja, uruchamianie i analiza wyników).

1. Wprowadzanie

Profilowanie czasu wykonania aplikacji jest nieodłącznym elementem tworzenia zaawansowanych aplikacji. Głównym zadaniem narzędzi profilujących jest zbadanie testowanego oprogramowania pod kątem czasu poświęcanego na poszczególne linie kodu, funkcje czy też większe moduły. Pozwala to ukierunkować proces optymalizacji na fragmenty kodu, które zajmują najwięcej czasu wykonania. Dzięki temu tzw. wąskie gardła (ang., bottle neck) mogą zostać szybciej zidentyfikowane i zlikwidowane. Jednym z narzędzi do profilowania aplikacji pod systemem operacyjnym Linux jest GNU GProf. Umożliwia one zbieranie statystyk czasowych podczas wykonania aplikacji. Często wraz z nim używane jest narzędzie GNU GCov, które dostarcza szczegółowych statystyk na temat pokrycia kodu. Dzięki niemu możemy łatwiej przetestować aplikację (testy pokrycia kodu i kompletności danych wejściowych) oraz znaleźć tzw. martwy kod.

2. Użycie narzędzia GNU Gprof

Aby skorzystać z prezentowanego narzędzia należy wykonać trzy następujące czynności:

- skompilować program tak, aby dodawał niezbędne informacje dla profilera
- uruchomić wcześniej odpowiednio skompilowany program
- uruchomić gprof z wynikami kompilacji

a) kompilacja

Podstawową opcją, którą trzeba podać dla kompilatora jest **`-pg`**. Trzeba ją dodać zarówno podczas kompilacji oraz linkowania (chyba, ze wykonujemy te dwie operacje na raz). Powoduje ona zbieranie informacji podczas wykonania aplikacji, które później możemy analizować. Dodatkowo jeśli chcemy przeprowadzić profilowanie na szczeblu pojedynczych linii należy dodać opcję debugera **`-g`** oraz **`-a.`**

```
$ g++ -pg gprof_ex.cpp -o plikWykonywalny
```

b) uruchomienie

Po wykonaniu kompilacji z odpowiednimi flagami należy uruchomić aplikacje w sposób standardowy.

```
$ ./plikWykonywalny
```

Testowany program może działać trochę wolniej ze względu na konieczność zbierania informacji dla profilera. Należy pamiętać, aby odpowiednio dobierać dane wejściowe, ponieważ niewykonywane funkcje nie będą testowane. Wyniki są zapisywane do pliku **`gmon.out`** tuż przed zamknięciem wykonania aplikacji, o ile zakończy się ona prawidłowo, czyli przez wywołanie **`return`**, ewentualnie

exit.

c) analiza wyników

W celu przeprowadzenia analizy otrzymanych wyników z pliku **gmon.out** należy uruchomić profilera gprof z parametrem będącym nazwą pliku wykonywalnego oraz ewentualnie opcjami i plikiem z danymi do profilowania. Istnieje także możliwość użycia graficznych nakładek na gprof (np. KCacheGrind, KProf).

```
$ gprof [opcje] ./plikWykonywalny [plik_z_danymi_do_profilowania]
```

Opcje:

- -A – włącza tryb wydruku kodu z komentarzem
- -J – wyłącza tryb wydruku kodu z komentarzem
- -p – włącza **profil płaski**
- -P – wyłącza **profil płaski**
- -q – włącza **graf wywołań**
- -Q – wyłącza **graf wywołań**
- -b – wyłącza legendę
- -l – włącza profilowanie za poziomie pojedynczych linii
- -z – wyświetla także te funkcje, które nigdy się nie wykonały

Domyślnym zestawem opcji jest -p -q.

– profil płaski

Poniżej widać przykładowy wydruk analizy prostego programu przy pomocy profilu płaskiego:

```
Each sample counts as 0.01 seconds.
  %     cumulative  self              self     total
 time    seconds   seconds   calls    s/call   s/call name
 90.08     37.59    37.59    24000     0.00     0.00  sortTable(int*)
 10.64     42.03     4.44       24     0.19     1.75  test1()
  0.00     42.03     0.00       24     0.00     0.00  displayTable(int*)
  0.00     42.03     0.00       24     0.00     0.00  fillTable(int*)
```

%time
czas spędzony w danej funkcji względem całego czasu wykonania aplikacji

cumulative seconds
ilość sekund spędzonych w danej funkcji plus w funkcjach leżących nad nią w tabeli

self seconds
czas spędzony w danej funkcji

calls
ilość wywołań danej funkcji

self s/call
średni czas spędzony w danej funkcji

total s/call
średni czas spędzony w danej funkcji oraz w funkcjach, które zostały z niej wywołane

name
nazwa danej funkcji

– graf wywołań

Tryb ten przedstawia więcej informacji na temat czasu wykonania aplikacji. Pozwala na przykład

wykryć funkcje, które same w sobie wykonują się krótko, natomiast wywoływane z nich funkcje stanowią wąskie gardło.

Poniżej widać przykładowy wydruk analizy prostego programu przy pomocy grafu wywołań:

```
index % time    self  children    called         name
                                                 <spontaneous>
[1]    100.0    0.00   42.03                    main [1]
                4.44   37.59      24/24          test1() [2]
-----------------------------------------------
                4.44   37.59      24/24          main [1]
[2]    100.0    4.44   37.59      24             test1() [2]
               37.59    0.00   24000/24000       sortTable(int*) [3]
                0.00    0.00      24/24          fillTable(int*) [11]
                0.00    0.00      24/24          displayTable(int*) [10]
-----------------------------------------------
               37.59    0.00   24000/24000       test1() [2]
[3]     89.4   37.59    0.00   24000             sortTable(int*) [3]
-----------------------------------------------
                0.00    0.00      24/24          test1() [2]
[10]     0.0    0.00    0.00      24             displayTable(int*) [10]
-----------------------------------------------
                0.00    0.00      24/24          test1() [2]
[11]     0.0    0.00    0.00      24             fillTable(int*) [11]
-----------------------------------------------
```

Linie składające się z samych myślników dzielą tabele na wpisy odpowiadające pojedynczym funkcjom. Pojedynczy wpis dla funkcji może składać się więcej niż z jednej linii. Funkcja, której wpis dotyczy jest oznaczona indeksem w nawiasach kwadratowych. Funkcje znajdujące się powyżej są to funkcje ją wywołujące, tzw. "rodzice", natomiast funkcje poniżej są to funkcje przez nią wywoływane, tzw. funkcje "potomne". Wpisy są uporządkowane od najbardziej do najmniej czasochłonnej funkcji.

index
pozwala odnaleźć funkcję, której dotyczą podawane statystyki, jednocześnie jest to numer porządkowy nadawany kolejnym funkcjom w programie

%time
czas spędzony w danej funkcji (i funkcjach z niej wywoływanych) względem całego czasu wykonania aplikacji

%self
linijka główna – całkowity czas spędzony w danej funkcji
linijka "rodzica" – czas spędzony w danej funkcji, gdy została ona wywołana z podanego "rodzica"
linijka "potomka" – czas spędzony w funkcji potomnej, gdy została wywołana z danej funkcji

%children
linijka główna - całkowity czas spędzony w funkcjach potomnych danej funkcji
linijka "rodzica" – czas spędzony w funkcjach potomnych, gdy dana funkcja została wywołana z danego rodzica
linijka "potomka" – czas spędzony w funkcjach potomnych danego potomka danej funkcji

%called
linijka główna – ilość wywołań danej funkcji, w przypadku wywołań rekurencyjnych pojawiają się dwie liczby oddzielone znakiem '+', gdzie pierwsza liczba to ilość wywołań nierekursywnych, a druga rekursywnych
linijka "rodzica" – ilość wywołań danej funkcji z danego rodzica oraz ilość nierekursywnych wywołań danej funkcji ze wszystkich rodziców

linijka "potomka" – ilość wywołań danego potomka z danej funkcji oraz ilość wszystkich nierekursywnych wykonań danego potomka

%name

nazwa funkcji oraz jej numer porządkowy

Jeśli występują wywołania rekursywne, np. funkcja `silnia()` wywołuje funkcję `silnia()` itd., powstaje problem jak liczyć wartości **children** i **self**. W takim przypadku gprof numeruje kolejne cykle i dla każdego z nich umieszcza oddzielny wpis. Zawiera on informacje o czasie wywołania wszystkich funkcji w cyklu. "Rodzicami" są funkcję, które nie należą do cyklu i wywołały bezpośrednio jakieś funkcje cyklu. Natomiast funkcje potomne to wszystkie funkcje cyklu oraz funkcje bezpośrednio przez nie wywoływane.

- **profilowanie pojedynczych linii**

Tryb ten pozwala stwierdzić ile razy wykonała się dana funkcja oraz dane linie kodu. Wyliczone zostają także średnie czasy wykonania dla funkcji oraz jednej linii kodu. Aby móc uzyskać wszystkie informacje należy skompilować kod dodatkowo z opcjami **-g -a**.

3. Użycie narzędzia GNU GCov

Aby skorzystać z prezentowanego narzędzia należy wykonać trzy następujące czynności:

- skompilować program tak, aby dodawał niezbędne informacje dla profilera
- uruchomić wcześniej odpowiednio skompilowany program
- uruchomić gcov z wynikami kompilacji

a) kompilacja

Program profilowany narzędziem GNU GCov powinien zostać skompilowany z opcjami umożliwiającymi zbieranie informacji podczas jego działania: **-fprofile-arcs -ftest-coverage** (jeśli oddzielnie wykonujemy proces linkowania i kompilacji, trzeba je podać podczas tych dwóch operacji). Pierwsza opcja powoduje powstanie pliku *.gcda podczas kompilacji, który pomaga zrekonstruować graf działania aplikacji, druga natomiast powoduje powstanie plików *.gcno dla każdego pliku obiektowego podczas uruchomienia. Pliki z rozszerzeniem .gcno zawierają podsumowania wyników pokrycia kodu.

```
$ g++ -fprofile-arcs -ftest-coverage gcov_ex.cpp -o plikWykonywalny
```

b) uruchomienie

Po wykonaniu kompilacji z odpowiednimi flagami należy uruchomić aplikacje w sposób standardowy:

```
$ ./plikWykonywalny
```

Testowany program może działać trochę wolniej ze względu na konieczność zbierania informacji na temat pokrycia kodu.

c) analiza wyników

Aby uzyskać czytelny opis kodu wraz z częstotliwością wykonywania się poszczególnych linii należy uruchomić gcov z nazwą pliku źródłowego jako parametrem:

```
$ gcov gcov_ex.cpp
File `gcov_ex.cpp'
Lines executed:87.50% of 8
gcov_ex.cpp:creating `gcov_ex.cpp.gcov'
```

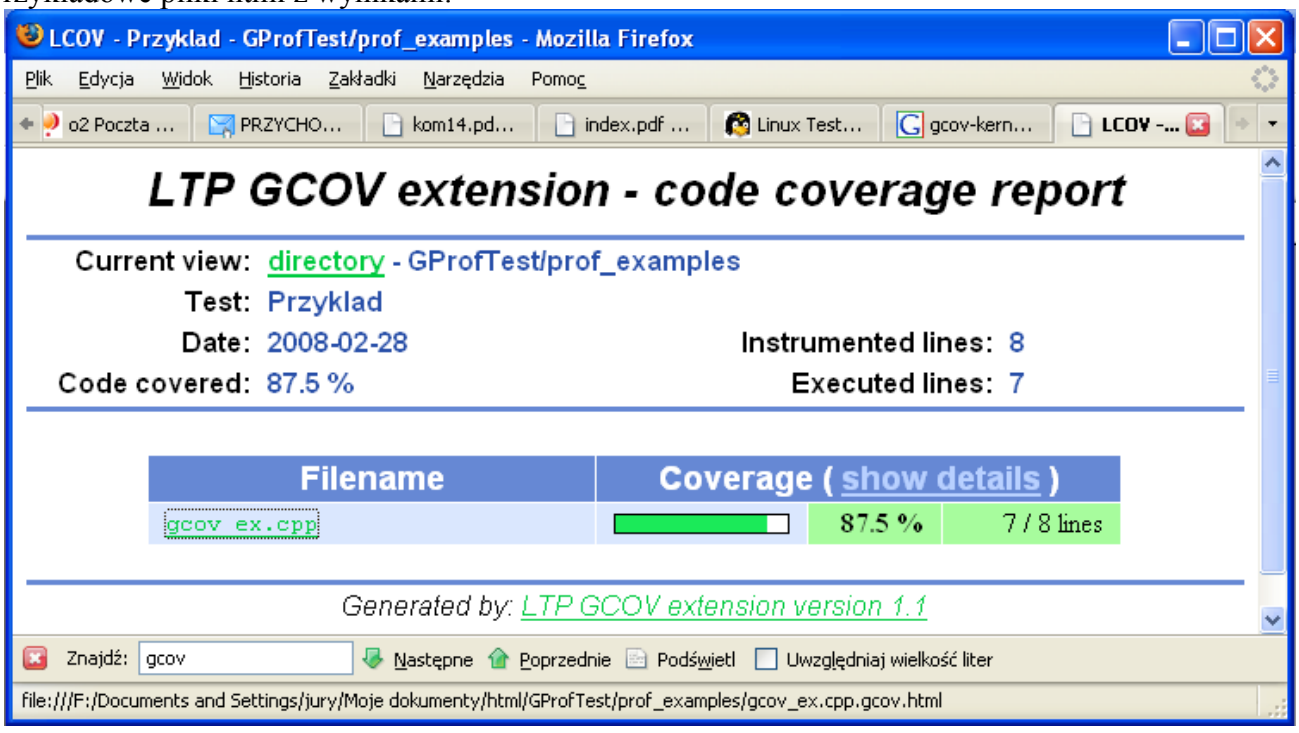
Spowoduje to powstanie pliku plikZródłowy.gcov, w którym zawarte są informacje na temat pokrycia kodu:

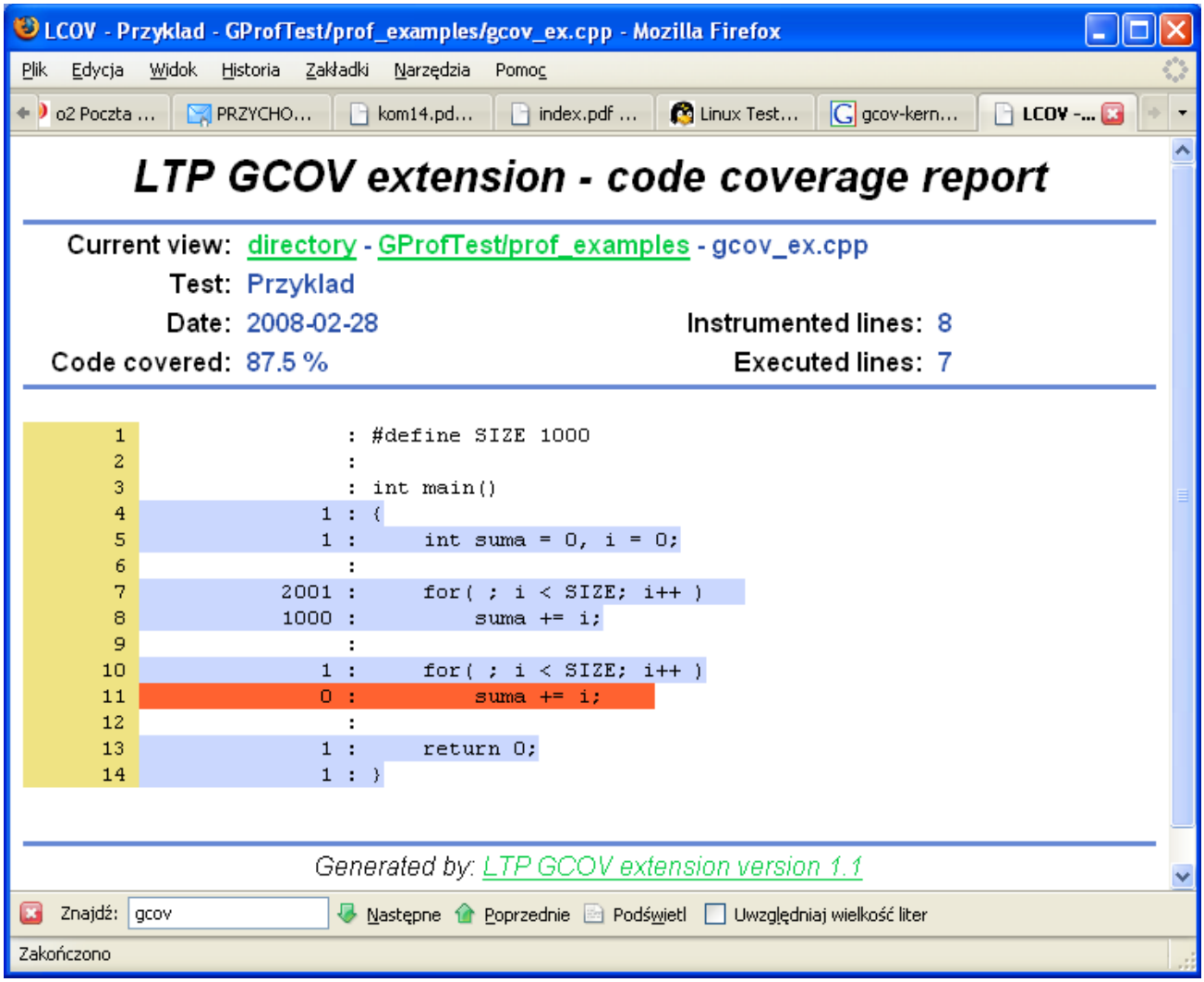
```
        -:    0:Source:gcov_ex.cpp
        -:    0:Graph:gcov_ex.gcno
        -:    0:Data:gcov_ex.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#define SIZE 1000
        -:    2:
        -:    3:int main()
function main called 1 returned 100% blocks executed 83%
        1:    4:{
        1:    5:    int suma = 0, i = 0;
        -:    6:
     2001:    7:    for( ; i < SIZE; i++ )
     1000:    8:        suma += i;
        -:    9:
        1:   10:    for( ; i < SIZE; i++ )
    #####:   11:        suma += i;
        -:   12:
        1:   13:    return 0;
        1:   14:}
```

Jeśli chcemy możemy także użyć programu lcov do generacji statystyk w postaci plików html. Przykład użycia:

```
$ g++ -pg gcov_ex.cpp -o plikWykonywalny
$ ./plikWykonywalny
$ lcov --capture --directory . --output-file nazwa.info -test-name "test"
$ genhtml nazwa.info -title "Przyklad" -show-details
```

Przykładowe pliki html z wynikami:




LCOV - Przyklad - GProfTest/prof_examples/gcov_ex.cpp - Mozilla Firefox
Plik Edycja Widok Historia Zakładki Narzędzia Pomoc
o2 Poczta ... PRZYCHO... kom14.pd... index.pdf ... Linux Test... gcov-kern... LCOV -...
LTP GCOV extension - code coverage report
Current view: directory - GProfTest/prof_examples - gcov_ex.cpp
Test: Przyklad
Date: 2008-02-28
Instrumented lines: 8
Code covered: 87.5 %
Executed lines: 7
1 : #define SIZE 1000
2 :
3 : int main()
4 1 : {
5 1 : int suma = 0, i = 0;
6 :
7 2001 : for(; i < SIZE; i++)
8 1000 : suma += i;
9 :
10 1 : for(; i < SIZE; i++)
11 0 : suma += i;
12 :
13 1 : return 0;
14 1 : }
Generated by: LTP GCOV extension version 1.1
Znajdź: gcov
Następne Poprzednie Podświetl Uwzględniaj wielkość liter
Zakończono