Optymalizacja programów Open-Source

Krzysztof Lichota
lichota@mimuw.edu.pl

Ogólne zasady optymalizacji

- 10 zasad optymalizacji:
  1. Mierzenie
  2. Mierzenie
  3. Mierzenie
  4. Mierzenie
  5. Mierzenie
  6. Mierzenie
  7. Mierzenie
  8. Mierzenie
  9. Mierzenie
  10. Mierzenie
- „Przedwczesna optymalizacja to samo zło” ("Premature optimization is the root of all evil (or at least most of it) in programming." - D.Knuth)

Przedwczesna optymalizacja

- Najczęściej prawdziwe zatory powstają zupełnie gdzie indziej niż się spodziewamy [1]
- Optymalizacja na zapas to strata czasu i utrata modularności oraz czytelności kodu

ale...

- Kod musi być dobrze zaprojektowany, modularny, żeby można było go optymalizować
- Przepisanie jednego fragmentu programu to nie tragedia, tylko normalna praktyka [2]

[1] http://www.taylor.se/blog/2006/11/22/why-optimizing-without-profiling-is-inefficient/
[2] http://blogs.msdn.com/ricom/archive/2007/02/08/performance-problems-survery-results.aspx

Pomiary

- Podstawą optymalizacji jest posiadanie odpowiednich pomiarów, które dobrze obrazują to, co chcemy optymalizować
- Pomiary muszą być uczciwe i powtarzalne
- Pomiary „na oko” są mylne (ale można też to wykorzystać do „oszukiwania”, zajmiemy się tym na ostatnim wykładzie)

Co można optymalizować

- Czas uruchomienia
- Szybkość działania
- Interakcyjność (czas odpowiedzi)
- Przepustowość
- Działanie w przypadku działania innych programów (np. intensywnych obliczeń w tle).
- Przypadek szczególny – czas uruchomienia systemu

Wykonywanie pomiarów

- Każdy pomiar zaburza w jakiś sposób działanie, więc należy wybierać techniki pomiarowe, których wpływ na badane działanie jest niewielki
- W trakcie badania należy zminimalizować wpływ innych czynników (np. procesy zabierające dużo pamięci, zadania wykonywane w tle)
- Badania należy powtórzyć kilkukrotnie a ich wyniki uśrednić (minimalizuje wpływ innych czynników)
- Odchylenie standardowe wskazuje na stopień chaotyczności (i powtarzalności) pomiarów.

Wykonywanie pomiarów

- Pomiary należy w miarę możliwości wykonywać w docelowym środowisku.
- Pomiary powinny być powtarzalne, najlepiej wykonywanie pomiaru zautomatyzować i to od momentu uruchomienia systemu.
- Należy pamiętać, by kontekst wykonywania pomiaru był taki sam, tzn. przy tych samych uruchomionych programach, sterownikach, wersji jądra, itp.
- Kontekst najlepiej automatycznie zapisywać przy teście. Każdy czynnik może mieć wpływ na test.

Warm startup vs cold startup

- W szczególności nalezy rozróżniać testowanie „warm startup” od „cold startup”
- „Cold startup” - pierwsze uruchomienie programu (w szczególności po starcie systemu, kiedy nie wszystkie usługi działają albo się nie rozgrzały)
- „Warm startup” - kolejne uruchomienia programu, przeważnie dużo szybsze, z powodu umieszczenia różnych elementów w pamięciach podręcznych (fragmentów plików, wpisów DNS, optymalizatora zapytań SQL, ...)

Poziomy optymalizacji

- Optymalizacja wysokiego poziomu – zmiany architektury programu, algorytmów, struktur danych
- Optymalizacja średniego poziomu – zmiany w strukturze kodu, rezygnacja z pewnych funkcji języka (np. obiektowości, wyjątków).
- Optymalizacja niskiego poziomu – użycie określonych funkcji procesora/dysku/itp., pakowanie struktur danych, optymalizacja kodu pod kątem architektury procesora, ...

[Terminologia moja własna]

Poziomy optymalizacji

- Najczęściej największe optymalizacje uzyskuje się poprzez zmiany na wysokim poziomie (np. algorytm bubble-sort vs quick-sort).
- Optymalizacje niższego poziomu są tracone, jeśli ulepszymy coś na wyższym poziomie (patrz – przedwczesna optymalizacja)

Dlatego:

- Optymalizacje staramy się wykonywać na jak najwyższym poziomie.

Wady optymalizacji niskiego poziomu

- Duży koszt utrzymania (dostosowywanie do określonych architektur)
- Utrata czytelności kodu.

Poza tym:

- Często kompilator robi lepszą robotę niż człowiek!

Kiedy wykonywać optymalizacje na niskim poziomie

- Optymalizacje na niskim poziomie są wykonywane tylko kiedy:
  - Zrobiliśmy już wszystkie dostępne optymalizacje na wyższym poziomie lub
  - Nie możemy wykonać zmian na wyższym poziomie z różnych względów (kompatybilność, stopień trudności, czasochłonność, ...)

  oraz
  - Pomiary wskazują, że dany fragment kodu jest odpowiedzialny za znaczący spadek wydajności

Optymalizacje wysokiego poziomu

- Wymagają dobrego zrozumienia tego, co program ma robić i jak się zachowuje
- Pomaga znajomość dużej liczby dostępnych rozwiązań, by dobrać rozwiązanie do zastosowania
- Można osiągnąć duży efekt małym kosztem

Źródła zatorów

- Głupota albo nieświadomość programistów (wykonywanie zbędnych czynności albo powielanie już zrobionych)
- CPU
- Niedostatki pamięci
- Interakcja z dyskiem (również pośrednia, np. mmap i swap)
- Sieć

Źródła zatorów

- Interakcja z urządzeniami wejścia/wyjścia
- Wyświetlanie (w tym debugowanie)
- Błędy lub określone założenia w implementacji systemu operacyjnego, sterowników, bibliotek, itp.
- Inne priorytety (np. ważniejsze by system po starcie pracował bez przestojów, a nie szybko startował).
- Ułatwianie sobie implementacji (np. czekanie na DHCP przy starcie systemu)
- Synchronizacja (np. między procesorami lub węzłami w klastrze)

Źródła zatorów

- Zatory mogą się różnić w zależności od środowiska, użytkownika, itd.
- Kluczowe są pomiary i to w różnych środowiskach.
- Testowanie musi obejmować całość systemu i w docelowej konfiguracji, środowisku, itd.

Zatory w CPU

- Duże zużycie CPU
- Również w jądrze!
- Narzut na przełączanie kontekstu (dużo wątków)
- Obsługa przerwań

Zatory sieciowe

- Częsty problem – DNS
- Timeout TCP
- Czas powrotu odpowiedzi (roundtrip time, np. protokół X)
- Ograniczenia przepustowości
- Korkowanie TCP (TCP_CORK)

Najsłabsze ogniwo

- Program jest tylko tak szybki jak najsłabsze ogniwo w przetwarzaniu
- Eliminacja zatorów w najsłabszym ogniwie powoduje ujawnienie zatorów w innym (i tak w nieskończoność)
- Dlatego konieczne jest ustalenie „wystarczająco dobrej” wydajności.
- Jeśli najsłabszymi ogniwami są elementy niezależne od nas i ich wydajność jest gorsza niż nasza „docelowa” wydajność – nie ma sensu optymalizować.

Wykrywanie najsłabszego ogniwa

- Identyfikacja najsłabszego ogniwa i badanie maksymalnej wydajności systemu może się odbywać przez zastąpienie go „wirtualnym” urządzeniem o maksymalnej wydajności.
- To samo dotyczy całych modułów programu (np. backend do przechowywania danych, moduł szyfrujący).
- Można w ten sposób określić maksymalną wydajność programu, której nie da się przeskoczyć przy wymianie tylko tego elementu.

„Idealne” urządzenia

- CPU - użycie szybkiego komputera
- Pamięć - maszyna z dużą ilością pamięci
- Dysk
  - Użycie RAM-dysku
  - Użycie przeznaczenie dużej ilości pamięci na cache i warm start.
- Sieć
  - Komunikacja przez loopback (uwaga na narzut związany z uruchamianiem serwera na tym samym komputerze) lub przez szybką sieć
  - Cache DNS

„Idealne” urządzenia

- Wyświetlanie – buforujące X-servery, użycie innego drivera wyświetlania (FB, Ascii-art).
- Output – wysyłanie do /dev/null

Wirtualizacja systemu

- Wirtualizacja zdarzeń (wirtualizacja CPU, pamięci, itp.)
- Symulowanie sieci (np. Sam moduł algorytmu podłączony do symulatora sieci)
- Maszyny wirtualne

Uwagi końcowe

- Maksymalna automatyzacja jest wskazana - uruchamianie po raz setny programu z zadanymi parametrami, przy jednoczesnym uruchomieniu programu nadzorującego, zebraniu wyników, uśrednieniu i wyprodukowaniu wykresów może doprowadzić do pasji (albo załamania)
- Właściwe narzędzia to klucz do sukcesu, ale trzeba znać dobrze ich mocne i słabe strony oraz sposób działania (co dokładnie mierzą i jak, jaki jest narzut, itp.).

Plan zajęć (1)

1. Ogólne zasady optymalizacji programów, źródła opóźnień w programach (1 zajęcia).
2. Narzędzia analizy kodu wysokiego poziomu (logowanie zdarzeń, profilery wysokiego poziomu) (2-3 zajęcia).
3. Narzędzia analizy kodu niskiego poziomu (profilery kodu maszynowego i analizatory użycia pamięci podręcznej procesora). Sposoby restrukturyzacji kodu i użycie wstawek w instrukcjach maszynowych (2 zajęcia).

Plan zajęć (2)

4. Analiza zużycia pamięci i narzędzia do radzenia sobie z problemami z pamięcią (3 zajęcia).
5. Analiza korzystania z dysku i sposoby optymalizacji korzystania z dysku (2 zajęcia).
6. Optymalizacja czasu uruchamiania programów (1 zajęcia).

Plan zajęć (3)

7. Symulowanie lepszej wydajności, czyli techniki “oszukiwania” użytkownika (1 zajęcia).
8. Prezentacja wyników analizy wybranych programów przez studentów (2-3 zajęcia).

Lektura

- http://www.gnome.org/~lcolitti/gnome-startup/analysis/ - analiza procesu uruchamiania środowiska GNOME
- http://developer.gnome.org/doc/guides/optimisation/ - zalecenia jak optymalizować ze strony GNOME
- http://primates.ximian.com/~federico/news-2005-09.html#g - ciekawy blog jednego z programistów GNOME, między innymi o jego zmaganiach z optymalizacją czasu wyświetlania okienka otwarcia pliku w GNOME
- http://blogs.msdn.com/ricom/default.aspx - blog jednego z inżynierów Microsoftu zajmującego się optymalizacją