Optymalizacja programów Open-Source

Profilery wysokiego poziomu
część 3

Krzysztof Lichota
lichota@mimuw.edu.pl

sysprof

sysprof

- Działa na zasadzie próbkowania (sprawdza backtrace okresowo)
- Narzut – niewielki (ale większy niż np. Google profiler)

Zalety sysprofa

- Bardzo łatwy w użyciu – wystarczy wstawić moduł do jądra i uruchomić program profilujący
- Pozwala na profiling całego systemu naraz
  - Pokazuje co się dzieje w systemie, co w innych sposobach profilowania łatwo można przeoczyć (np. demon, który się w trakcie testu uruchomił)
  - Pokazuje czas spędzony w procesach zależnych (np. Xorg)
- Pokazuje czas spędzony w jądrze
- Nie wymaga symboli do debugowania, żeby uzyskać sensowny profil

Zalety sysprofa (2)

- Pokazuje, co się dzieje w trakcie uruchomienia programu (linkowanie, inicjalizacja bibliotek, konstruktory, itp.) a nie tylko to, co się dzieje po starcie
- Można łatwo zachować ślad i go potem wczytać
- Ślad jest w formacie XML, więc można go potem łatwo obrabiać standardowymi narzędziami do XML
- Zawiera od razu standardowe narzędzie graficzne, które w obrazowy sposób prezentuje informacje i pozwala szybko znaleźć główne zatory

Wady sysprofa

- Wymaga uprawnień administratora, by wstawić moduł do jądra (ale na szczęście tylko po to)
- Nie pokazuje, kiedy program śpi
- Brak programu do uruchomienia profilera bez GUI (np. przy starcie systemu)
- Żeby zmienić częstotliwość próbkowania (domyślnie 200/s) trzeba zmienić źródła modułu
- Profiling jednego wybranego programu może wymagać przeklikania się przez długi stos wywołań

sysprof – na co zwrócić uwagę

- Profilowanie serwera X może wymagać rekompilacji (trzeba wymusić stosowanie modułów .so – szczegóły w README sysprofa)
- Maksymalna głębokość stosu wywołań to 512, można zmienić w źródłach modułu (SYSPROF_MAX_ADDRESSES)
- Maksymalna liczba przechowywanych naraz próbek to 256, można zmienić w źródłach modułu, gdy profiler nie nadąża z odczytywaniem (N_TRACES)

Logowanie

Zalety logowania

- Pozwala uzyskać bardzo szczegółowe informacje, co, gdzie, jak i dlaczego się dzieje
- Pozwala śledzić „historię” jakiegoś żądania, które przemieszcza się między różnymi modułami

Wady logowania

- Może mieć spory narzut (tym większy, im bardziej szczegółowo logujemy)
- Wymaga ręcznej ingerencji w kod programu
- Logowanie na tym samym komputerze może wprowadzać zakłócenia

Co logować

- Czas (z jak największą dokładnością)
- Wszystkie informacje pozwalające zlokalizować miejsce w programie, gdzie wystąpiło zdarzenie (nazwa pliku, linia, nazwa funkcji, unikalny identyfikator, np. napis)
- Najważniejsze parametry mające wpływ na wydajność (długość listy, rozmiar bufora, itp.)
- Numer wątku/procesora jeśli program jest wielowątkowy
- Inne parametry pozwalające określić kontekst (np. plik, do którego piszemy)

Co logować (2)

- Informacje pozwalające unikalnie zidentyfikować śledzoną strukturę (jeśli tworzymy historię)
- Linie logu powinny mieć ujednolicony format, żeby można je było przetwarzać

Śledzenie historii

- Ma głównie znaczenie przy optymalizowaniu latency i bandwidth w przetwarzaniu potokowym, gdzie przetwarzanie czegoś rozkłada się na wiele etapów i opóźnienia mogą wynikać z przestojów na jakimś etapie przetwarzania
- Przykład na bandwidth: serwer HTTP przepuszczający żądanie przez różne moduły, zablokowanie ostatniego (np. modułu logującego z powodu dostępu do dysku) powoduje przestój na wcześniejszych etapach i obniżenie bandwidth, chociaż system wydaje się nie obciążony

Śledzenie historii (2)

- Przykład na latency: żądanie odczytu z dysku wysłane z procesu, przechodzące przez podsystem wejścia/wyjścia (np. KIO), przesyłane do kolejnego procesu (KIO-slave), do serwera FTP i z powrotem do programu.
- Przykład na latency (2): program wyświetlający się na zdalnym serwerze X na wolnym łączu, żądanie przechodzi przez różne moduły w lokalnej bibliotece X, przez łącze, potem przez różne moduły w zdalnym serwerze X i w końcu dociera do karty graficznej.

Jak unikać narzutu logowania

Stosować systemy logujące, które:

- Nie blokują w momencie logowania (np. przekazują pamięcią dzieloną do innego procesu zamiast zapisywać na dysk)
- Jak się nie da, to przynajmniej nie zapisywać synchronicznie (np. na stderr), a w żadnym wypadku nie wyświetlać na konsoli w trybie graficznym!
- Logować przez sieć (unikać narzutu programu logującego zrzucającego dane na dysk)
- Logować na ramdysk

Na co należy zwrócić uwagę

- Logowanie z różnymi poziomami niekoniecznie powoduje obniżenie narzutu logowania – jeśli wyłączamy logowanie należy wyłączyć także np. generowanie argumentów, które są logowane, np. log(PERF)<<funkcja_wypisująca_wielką_strukturę()
- Wyłączenie logowania na poziomie LOG_DEBUG powoduje tylko nie wypisywanie wyjścia funkcji do logu, ale i tak zostanie ona wykonana!

Na co należy zwrócić uwagę (2)

- Należy używać makr, które rozwijają się do pustych instrukcji i inicjalizować argumenty wewnątrz nich, np.:
  DEBUG(log(PERF)<<funkcja_wypisująca_wielką_strukturę())
- Logowanie czasu za pomocą standardowych funkcji powoduje wywołanie funkcji systemowej i może mieć znaczący wpływ na wydajność programu! Jeśli to stanowi problem można logować czas na podstawie hardware counters.

Systemy logowania

Zastane systemy logowania

- Przeważnie każdy projekt ma już jakiś system logowania, który można wykorzystać (np. kdDebug() w KDE)
- Wystarczy dołożyć swoje makra, żeby np. automatycznie był dodawany czas, nazwa pliku czy numer linii
- Należy sprawdzić w jaki sposób zastany system logowania działa, żeby nie wprowadzać zbędnych opóźnień

Przydatne rzeczy do logowania

- __LINE__ - makro preprocesora rozwijające się do numeru bieżącej linii w programie
- __FILE__ - makro preprocesora rozwijające się do nazwy pliku źródłowego w programie
- __func__ (standard C99) i __FUNCTION__ (gcc) – literał zawierający nazwę funkcji (bez sygnatury)
- __PRETTY_FUNCTION__ (g++) - zawiera pełną nazwę funkcji z namespace i argumentami
- gettimeofday() - zwraca czas z dokładnością do mikrosekund (o ile sprzęt pozwala)

cout/cerr i logowanie do plików

- Wypisywanie na cerr jest kosztowne, bo każde wypisanie jest synchronizowane
- Jeśli potrzebny nam prosty system logowania, można logować do pliku i zrobić duży bufor wejścia/wyjścia, żeby zmniejszyć liczbę wywołań funkcji systemowych

log4cxx

- Zaawansowana i elastyczna biblioteka do logowania
- Pozwala logować do plików, przez sieć, na konsolę, do logu systemowego, itp.
- Pozwala logować asynchronicznie
- Do zmiany sposobu logowania wystarczy zmienić plik konfiguracyjny lub zmienną środowiskową, nie trzeba zmieniać kodu

Bibliografia

- http://www.daimi.au.dk/~sandmann/sysprof/
- http://primates.ximian.com/~federico/news-2005-07.html#
- http://logging.apache.org/log4cxx/
- http://gcc.gnu.org/onlinedocs/gcc/Function-Names.html
- http://primates.ximian.com/~federico/news-2006-03.html#
- http://primates.ximian.com/~federico/docs/2007-02-FOSD