LagHunter:
Wykrywanie problemów wydajnościowych 'w dziczy'

Maciej Strzelczyk

Na podstawie:
Catch Me If You Can: Performance Bug Detection in the Wild
Milan Jovic, Andrea Adamoli, Matthias Hauswirth

Dwa rodzaje wydajności

- Surowa
  - Łatwo się mierzy automatycznymi profilerami, które mierzą częstotliwość wykonywania różnych części kodu.
  - Poprawianie tej wydajności polega na optymalizacji 'gorącego' kodu, wskazywanego przez profilery.
  - Poprawienie surowej wydajności wcale nie musi poprawić komfortu użytkowania aplikacji.

Dwa rodzaje wydajności

- Odczuwalna
  - Jak szybko program reaguje na akcje użytkownika.
  - Bywa dużo bardziej pożądana przez zwykłego użytkownika.
  - Poprawia komfort pracy z aplikacją.
  - Może zależeć od dużo większej liczby czynników, przez co mierzenie jej jest na ogół trudniejsze.

LagHunter

- Program do mierzenia opóźnień w aplikacjach napisanych w Javie.
- Nie wymaga modyfikacji badanej aplikacji.
- Nie wpływa na wydajność badanego programu.
- Składa się z wielu agentów i jednego serwera, który gromadzi i podsumowuje zdobyte dane.

Ogólny schemat działania

- Aplikacje są wypuszczane w świat razem z agentem LagHuntera, który gromadzi informacje o wydajności w trakcie pracy aplikacji.
- Na koniec pracy aplikacji agent przesyła na serwer zdobyte informacje.
- Serwer gromadzi, analizuje i podsumowuje zdobyte informacje.
- Developerzy wiedzą co poprawiać lub mają przekonanie, że wszystko jest OK.

Jakie informacje zbierać?

- Najlepiej byłoby śledzić każde wywołanie funkcji i zapisywać stemple czasowe jej początku i końca.
- Powyższe niestety nie jest możliwe. Danych byłoby zbyt wiele, a samo ich gromadzenie spowodowałoby potworny spadek wydajności.
- Potrzebna jest więc jakaś metoda wyróżnienia funkcji, które chcielibyśmy śledzić.
- Do tego, informacje powinny być w formie pozwalającej na łatwe agregowane.

Jakie informacje zbierać?

- Konieczne jest wybranie pewnego zbioru funkcji kluczowych dla prędkości działania interfejsu aplikacji.
- Funkcje te będziemy nazywać kamieniami milowymi.
- Dzięki istnieniu pewnych standardów w zrębach GUI, możliwe jest automatyczne wykrywanie funkcji, które powinny zostać kamieniami milowymi.

Co zrobić jak program jest w kamieniu milowym?

- Najważniejsze jest oczywiście zarejestrowanie kiedy wywołanie funkcji będącej kamieniem milowym się zaczęło i zakończyło.
- Dodatkowo co losowy odstęp czasu pobierane są migawki stosu wywołań dla danego wątku.
- W ten sposób będziemy mieli zgrubne pojęcie co się działo podczas wykonywania kamienia milowego, a nie umniejszymy wydajności.

Jak wybierać kamienie milowe?

- Wybór funkcji, które mają stać się kamieniami milowymi jest kluczowy dla skutecznego zebrania informacji.
- Jeśli będzie ich za mało, to możemy nie dowiedzieć się o wielu problemach.
- Jeśli będzie ich za dużo, to wydajność aplikacji może się pogorszyć, przez co zaburzymy nasze badanie.

Jak wybierać kamienie milowe?

- Powinny pokrywać większość wykonania programu, abyśmy nie przeoczyli żadnych problemów.
- Powinny być to funkcje wywoływane podczas obsługi akcji wykonywanych przez użytkownika.
- Nie powinny być zbyt często wywoływane. (Z perspektywy komputera).

Kandydaci na kamienie milowe

- Event dispatch method (zręb)
- Funkcje obsługujące poszczególne zdarzenia (zręb)
- Użycia wzorca projektowego Polecenie
- Obserwatorzy
- Granice modułów (przydatne dla wtyczek)
- Pozostałe, zależne już od konkretnej aplikacji

Analiza

- Po zakończeniu pracy programu agent przesyła informacje na centralny serwer.
- Serwer podsumowuje i łączy ze sobą raporty posługując się głównie migawkami stosu wywołań funkcji.
- Wyszukuje funkcje, które ogólnie zajmują najwięcej czasu.
- Generuje czytelny raport z wykresami, tabelkami itp.

Implementacja

- Agenci korzystają z JVMTI (Java Virtual Machine Tool Interface).
- Podczas uruchamiania agent przepisuje wszystkie klasy programu przygotowując odpowiednio kamienie milowe.
- Podczas działania aplikacji zbiera próbki stosu wywołań funkcji oraz informacje o działaniu odśmiecacza.
- Podczas wyłączania aplikacji wysyła wyniki do serwera.

Wykrywanie kamieni milowych

- Agent z łatwością wykrywa event dispatch method zrębów SWT i Swing.
- Obserwatorów wykrywa wykrywając użycia interfejsu EventListener.
- Użycia metody paint() we wszystkich klasach dziedziczących po java.awt.Component (dla Swinga, bo SWT używa do malowania obserwatorów).
- Wywołania metod JNI, mogą one długo trwać.

Narzuty

- Przepisywanie klas i znajdywanie kamieni milowych potrafi zająć sporo czasu, dlatego wyniki przepisania są zapisywane.
- Nie wszystkie wywołania kamieni milowych są zapisywane. Większość z nich jest poniżej pewnego progu, więc są odrzucane.

Narzuty przy uruchamianiu

- Przeprowadzono testy na Eclipse i NetBeans w celu zmierzania narzutów.
- Oryginalne czasy startu aplikacji: 1.1s i 2.85s
- Z LagHunterem bez cache'a: 3.44s i 7.28s
- Z LagHunterem z cache'm: 1.58s i 3.2s

- Jak widać, narzut przy starcie jest zauważalny jedynie dla pierwszego uruchomienia. Jest to nieuniknione, ale też raczej bezbolesne.

Narzuty podczas działania

- Średni czas pobrania próbki stosu wahał się od 0.5ms do 2.23ms. Jest to zaniedbywalne przy fakcie, że człowiek jest w stanie zauważyć zdarzenia co najmniej 100ms.
- Ustawienie progu dla kamieni milowych na poziomie 3ms powoduje zmniejszenie zapisów około 100 krotnie. Jest to kluczowe dla dbania o wydajność LagHuntera.

Odczucia użytkowników/testerów

- Twórcy dali swoim studentom do użytku trzy pogramy podpięte pod LagHuntera: BlueJ, Informa i Eclipse.
- Jedyna uwaga od uczniów: Aplikacja nie wyłącza się natychmiast.
- Było to zrozumiałe, gdyż właśnie wtedy wysyłane są raporty.

Testowanie LagHuntera

- Działanie LagHuntera było analizowane głównie przy pomocy badania Eclipse'a.
- Użycie tylko jednej aplikacji może budzić podejrzenia, że LagHunter nie sprawdzi się gdzie indziej.
- Jest to jednak jeden z największym programów interaktywnych napisanych w Javie, do tego posiada on bardzo wiele wtyczek.
- Niektóre wtyczki są bardziej skomplikowane niż niejedne osobne aplikacje.

Ocena działania

- Błędne zgłoszenia: LagHunter oferuje developerom podsumowanie z działania programu, to oni muszą podjąć decyzję, czy jakieś działanie jest błędem czy nie.
- Brak zgłoszenia błędu: jeśli opóźnienie występuje poza kamieniem milowym, to nie zostanie wykryte. Jest to jednak zgodne ze specyfikacją LagHuntera.
- LagHunter nie ma raczej wpływu na współbieżność programów.
- Brak automatycznej optymalizacji – nigdy nie była planowana.

Ocena działania

- Trudność w reprodukcji błędów. Można dodać do LagHuntera opcję robienia zrzutów ekranu i śledzenia działań użytkownika, jednak to uderza w jego prywatność.
- Koszt próbkowania stosu. Przy zbyt dużej częstotliwości przestaje mieć sens. Na szczęście 10Hz jest wystarczające.
- Niedokładność próbkowania stosu. Kompilatory JIT potrafią trochę popsuć próbki, jednak kiedyś zostaną poprawione i problemu powinno nie być.