Szymon Giżecki
Chu Hong Hai
Mateusz Kopeć

Automatyczne testowanie jądra Linuksa - notatki

Metodologie przygotowywania i przeprowadzania testów poprawnościowych

Statyczna analiza kodu

Idea

Zanim przejdziemy do budowania aplikacji i testowania jej działania można się pokusić o sprawdzenie poprawności kodu źródłowego. Jest to możliwe dzięki technikom statycznej analizy kodu. Korzystając z metod matematycznych i informatycznych, staramy się dowodzić poprawności programu bądź też próbujemy pokazać, że program błędnie działa (i zlokalizować ten błąd).

Metody

Istnieje dużo technik służących statycznej analizie kodu, m.in.:

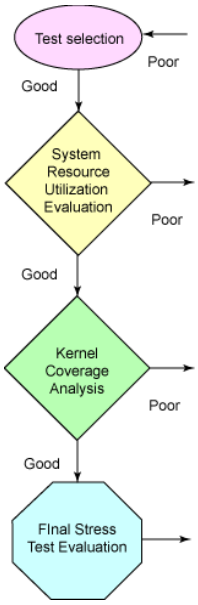
- Weryfikacja modelowa (model checking) – tworzymy uproszczony model z ograniczoną liczbą stanów, potem sprawdzamy czy ten model spełnia żądane wymagania.
- Analiza przepływu danych (data-flow analysis) – zbiera informacje o możliwych wartościach wszystkich zmiennych programu w pewnym momencie wykonania.
- Abstract interpretation – tworzy abstrakcyjną maszynę, która aproksymuje działanie pierwotnego systemu. Dobrze stworzona abstrakcyjna maszyna spełnia bardzo ważną zależność: każda prawdziwa właściwość w niej jest również prawdziwa dla pierwotnego systemu.
- Asercje – warunki logiczne, które muszą być spełnione przez kod programu.
- Semantyka – określenie semantyki (operacyjnej, denotacyjnej) programu pozwala wykryć pewne pułapki danego języka programowania.

Narzędzia

Można bez trudu znaleźć narzędzia pomocne w statycznej analizie kodu, zarówno darmowe jak i komercyjne. Warto tu wspomnieć o:

- Sparse – specjalnie zaprojektowany do szukania błędów w jądrze Linuksa.
- Cppcheck – wykrywa wycieki pamięci, przepełnienie bufora i inne typowe błędy.
- Coverity Prevent – komercyjna aplikacja do analizy kodu w C/C++, C#, Javie

Przygotowanie testów



Wybór zestawów testowych

Testy powinny mocno obciążyć główne zasoby (CPU, pamięć, I/O, networking) oraz pokryć dużą część kodu jądra. Pierwsze wymaganie zapewnia stabilność systemu w krytycznych sytuacjach. Drugie daje gwarancję, że jądro działa poprawnie w prawie wszystkich swoich czynnościach, a nie tylko tych najczęściej wykonywanych.

Podczas wyboru testów zwracamy uwagę również na to, czy są one automatyczne (lub półautomatyczne). Pozwala to na testowanie stabilności przez długi okres czasu (do 90 dni) bez ingerencji człowieka oraz powtarzanie testów w podobnych warunkach.

W przypadku projektu Open Source, jakim jest jądro Linuksa, dobrze jest również wybrać testy, które umożliwiają łatwą (darmową) publikację wyników w ogólnie przyjętym formacie. Dzięki temu różne zespoły pracujące nad tym samym projektem mogą korzystać z wysiłku innych i ta sama praca nie będzie wykonywana wielokrotnie.

Ocena wykorzystania zasobów systemowych

Cztery zasoby, które głównie odpowiadają za czas reakcji systemu i które powinny być mocno obciążone podczas testów:

- CPU: czas spędzony na procesowaniu danych przez jednostki procesorowe
- pamięć: czas spędzony na odczytaniu/zapisaniu danych z/do fizycznej pamięci
- I/O: czas spędzony na odczytaniu/zapisaniu danych z/do dysku fizycznego
- networking: -------||-------- sieci

Do oceny, czy dany zestaw testów odpowiednio obciąża dane zasoby, warto korzystać z narzędzi: **top**, **sar** i **iostat**

- **top** pozwala szybko sprawdzić, które zasoby (CPU, pamięć, I/O) są wykorzystywane przez testy
- **sar** zbiera i tworzy raport o aktywności systemu
- **iostat** monitoruje I/O, potrafi podać raporty co pewien czas (**iostat -x 1** podaje rozszerzone statystyki co 1 sekundę)

Gdy dana kombinacja testów jest już wybrana, trzeba uruchomić te testy przez odpowiednio długi okres czasu by dokładnie ocenić, czy spełniają nasze wymagania. Równolegle z testami narzędzie **sar** powinno być uruchamiane.

Analiza pokrycia kodu jądra

W normalnym użytkowaniu tylko mała część funkcjonalności jądra Linuksa jest wykorzystywana. Nawet testy intensywnie obciążające główne zasoby systemowe nie są w stanie pokryć dużej części kodu. Dlatego obowiązkiem testera jest ocena stopnia pokrycia kodu przez testy i ewentualne dobranie dodatkowych jednostek, by zapewnić uniwersalność całego zestawu. Istotną pomoc w tym zajęciu mogą stanowić 2 narzędzia: **gcov** i **lcov**.

Ocena końcowego testu obciążeniowego

Wybrany zestaw testowy należy uruchomić na stabilnej wersji jądra przez dłuższy okres czasu. Ma to dwojaki cel: wykonanie testu na stabilnym jądrze pozwala wykryć i usunąć konflikty między samymi testami, a wynik narzędzia **sar** posłuży jako punkt odniesienia dla przyszłych uruchomień tego zestawu testowego.

Po tych 4 krokach możemy uznać przygotowanie testów za zakończone i przystąpić do ich uruchomienia.

Przeprowadzenie testów

Kryterium oceny

Za miarę stabilności i niezawodności systemu uznajemy jego czas ciągłego i bezawaryjnego działania. W związku z tym testy są uruchomione i działają bez przerwy przez długi okres czasu (30, 60, 90 dni). Przed testem właściwym często jest uruchomiony 24-godzinny test wstępny w celu sprawdzenia poprawności systemu.

Podczas testowania odpowiednie narzędzia bez przerwy zbierają statystyki dotyczące systemu. Aby ustawić cykliczne zapisywanie danych statystycznych można używać narzędzia **cron**.

Metodologie przygotowywania i przeprowadzania testów wydajnościowych. Techniki pozwalające na poprawne odczytywanie informacji o wydajności

Co mierzymy

Przed przystąpieniem do przeprowadzenia testów wydajnościowych musimy zdefiniować sposób ilościowego mierzenia wydajności.

1. Na początku określamy miarę, której będziemy używać do testów np. ilość wysłanych wiadomości na sekundę.
2. Potem upewniamy się, że wybrane benchmarki są odpowiednie do mierzenia wydajności i obciążenia komponentów jądra.
3. Tworzymy również dokumentację, która pozwoli innym powtórzyć nasze testy.
4. Na końcu określamy, jakie dane są zebrane w celu badania wydajności systemu.

Zanim uruchomimy testy musimy odpowiednio dostosować sprzęt i oprogramowanie. Ma to

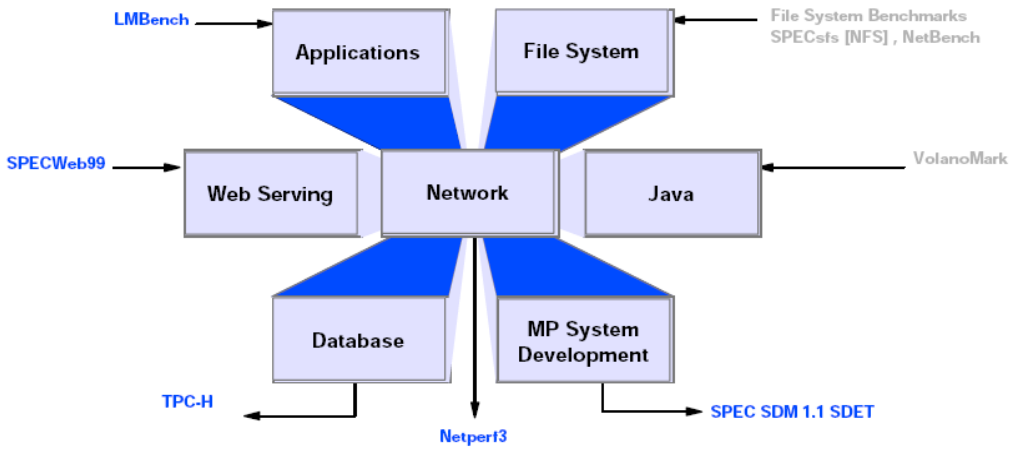
zapobiegać błędnym odczytom spowodowanym niechcianymi czynnikami. Sam proces polega na cyklicznym ustawianiu i mierzeniu obciążenia zasobów systemowych, jak również i parametrów hardware'u w pracy.

Narzędzia

Odczytanie informacji o wydajności wymaga użycia specjalnych programów. Na szczęście istnieje szereg narzędzi Open Source, które to zadanie wykonują. Korzystanie z tych programów oprócz oczywistej zalety w postaci ceny (a raczej braku) ma również inne, niebagatelne znaczenie: wyniki naszych badań będą mogli przetwarzać inni ludzie w Open Source Community.
Narzędzia, z których można korzystać to m.in.

- wirtualny system plików /proc
- lockmeter (SGI) – zajmuje się spinlockami
- kernprof (SGI) – profiler kernela
- trace facility (IBM) – narzędzie do trace'owania IBM
- sstat – monitoruje szynę SCSI
- schedret – zbiera statystyki o schedulerze
- acgparse – bierze ACG (annotated call graph) z kernprof i przetwarza dalej

Oprócz narzędzi OSC (Open Source Community) czasami potrzebne będą testy przeprowadzone sprawdzonymi, standardowymi (w przemyśle) benchmarkami.



Fault Injection i Linux Test Project

1). Mechanizm „Fault Injection”

Każde dobrze napisane oprogramowanie musi mieć mechanizmy obsługi błędów i wyjątków. Problem tkwi jednak w tym, iż jednocześnie staramy się uniknąć wykonywania tych ścieżek, ponieważ naszą oczywistą intencją jest pisanie programów bez błędów. Mechanizm „fault injection” pozwala właśnie na testowanie zachowania naszego oprogramowania w momencie wystąpienia błędu, poprzez jego sztuczne wywołanie.

2). Rodzaje „Fault Injection”

Rozróżniamy dwa rodzaje „fault injection”:
- software implemented fault injection
- runtime fault injection

Ta pierwsza polega na skompilowaniu programu z kodem który automatycznie „uruchomi” pożądany błąd.

Prosty przykład:
mamy funkcję `int fun(int a)` zwracającą nieujemne liczby oraz pewne miejsce w programie:

```
if(fun(a) > 10) {
     coś
}
else {
     coś2
}
```

możemy do programu dodać funkcję

```
int fj_foo(int a) {
     return a + 10;
}
```

i teraz obudowujemy naszą funkcję w foo, czyli:

```
if(foo(fun(a)) > 10) {
     coś
}
else {
     coś2
}
```

Jest to oczywiście bardzo prosty przykład, jednak można się domyślić, że używanie tego pierwszego rodzaju fault injection jest w istocie dosyć uciążliwe, gdyż za każdym razem musimy przekompilowywać nasz kod.

Drugi sposób, czyli „runtime fault injection” polega na bezpośrednim manipulowaniu pamięcią lub rejestrami CPU, tak aby sztucznie wywołać błąd programu lub systemowy w trakcie działania testowanego kawałka kodu. W Unixie mamy nawet dostępne biblioteki

pozwalające na użycie tego mechanizmu (np. fault.c, która przykładowo pozwala na wywołanie błędu Segmentation Fault, itd.)

Są również specjalne apliakcje ułatwiające korzystanie z ww. mechanizmu. Taką aplikacją jest np. FERRARI, czyli „Flexible Software-Based Fault and Error Injection System". Aplikacja ta niewątpliwie ułatwia używanie „fault injection", nie czyni go jednak prostym.

3). **Linux Test Project – Informacje ogólne**

Został zainicjowany przez konsorcjum SGI. Do tego momentu, aby testować linuxa, koniecznym było używanie wielu różnych programów z testami, a i tak duża część funkcjonalności jądra systemu nie mogła być testowana. Stąd pomysł na stworzenie jednej dużej aplikacji pozwalającej na kompleksowe testowanie jądra systemu.

Projekt ten został przejęty przez IBM, którego własnością jest do dzisiaj. Jest on tworzony jako cały czas rozwijane oprogramowanie Open Source. Zawiera obecnie ponad 3000 testów. Napisany jest głównie, bo w ponad 94%, w C; pozostałe języki to Perl i Shell, używane przede wszystkim do processing'u danych.

LTP jest używany głównie przez developerów jądra, a także przez entuzjastów oraz hackerów.

Przed uruchomieniem LTP zaleca się dokładne przetestowanie komputera (hardware'u), do czego posłużyć może Memtest86 oraz CPUBurn.

4). **LTP Rodzaje testów**

Unit testing – testowanie pojedynczych funkcjonalności, w możliwie jak największej izolacji od reszty systemu

Integration testing – kiedy już mamy przetestowane pojedyncze części systemu, to należy sprawdzić, czy i jak współpracują one ze sobą

System testing – testowanie API kernela

Regression testing – testowanie zgodności oprogramowania, które piszemy z różnymi wersjami oprogramowania/jądra

Strees testing – testowanie ww. elementów pod dużym obciążeniem systemu

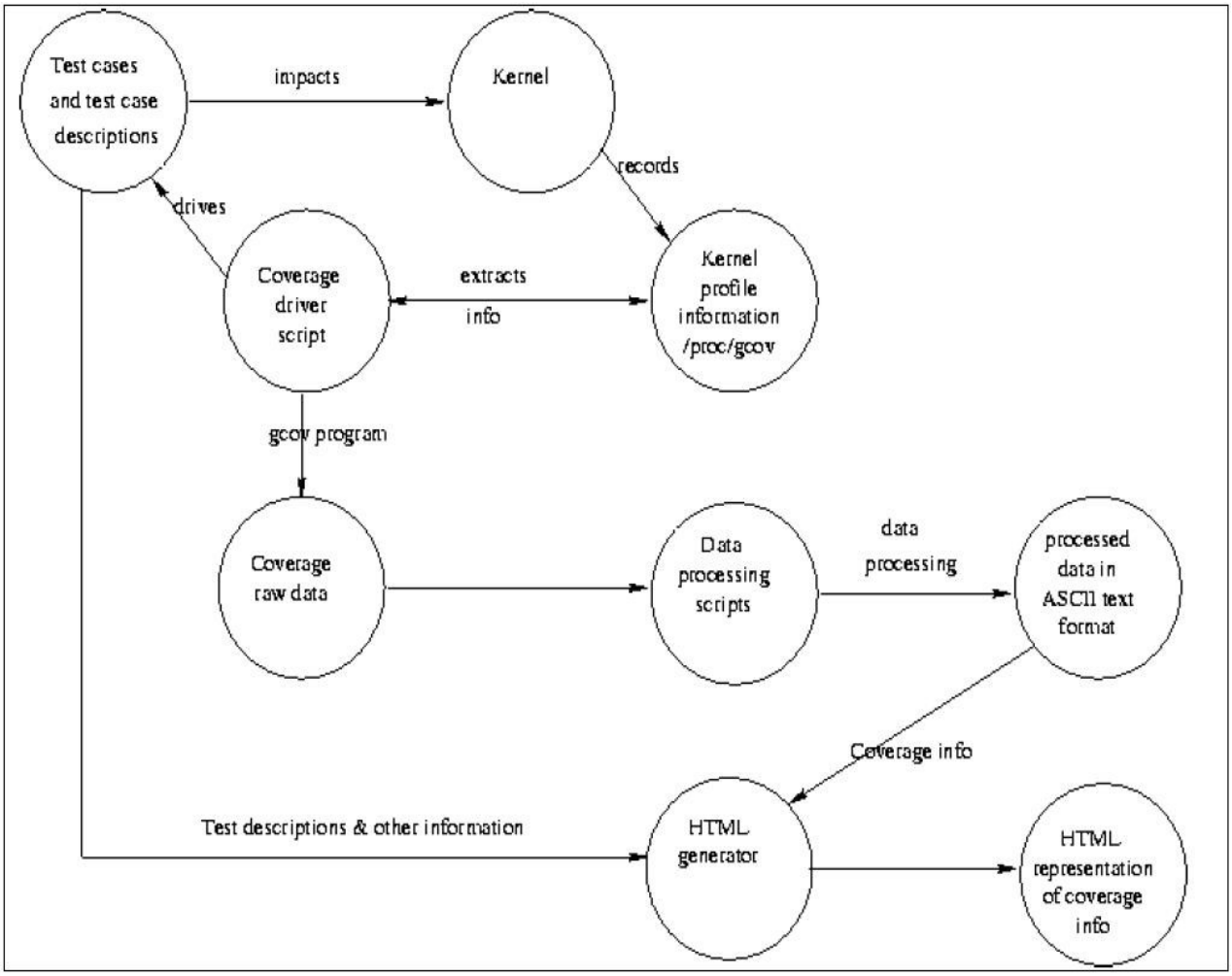
LTP natomiast nie udostępnia mechanizmów do testowania wydajności systemu (tzw. benchmarków) oraz nie pozwala na testowanie samej kompilacji programów.

5). **LTP – struktura**

LTP pozwala na testowanie wielu różnych części jądra systemu. Przede wszystkim API jądra, zarządzania pamięcią, scheduler'a, wątków jądra, przeróżnych systemów plików, podsystemu /proc, sieci, oraz sterowników wielu urządzeń (np. DVD, CDROM, FDD, a nawet już prawie nigdzie nie używanych taśm – TAPE)

Testy są ułożone w bardzo przejrzystą strukturę, co ułatwia korzystanie z LTP.

6). **LTP – diagram działania**



Zanim zostanie uruchomiony wybrany test, tworzony jest „obraz czystego systemu". Tzn. moduł GCOV zbiera informacje odnośnie ilości wykonań linii kodu systemu potrzebnych do jego normalnej pracy, aby później móc odfiltrować te wykonania od tych, które były uruchamiane przez nasz test.
Następnie uruchamiany jest wybrany test. Moduł GCOV w tle zbiera informacje na które linie kodu ma on wpływ. Wszystkie te dane zapisywane są jako kod ASCII, później przetwarzany do kodu HTML

7). **LTP – zbieranie danych**

Testy w LTP są ułożone w drzewiastą strukturę. Tzn. na samym spodzie są dostępne testy, które mają wpływ na wiele różnych części systemu, natomiast im schodzimy niżej w hierarchii katalogów, tym testy stają się coraz bardziej wyizolowane. Pomaga to w zdefiniowaniu relacji pomiędzy wybranymi testami, a modułami systemu, których będą one dotyczyć.

Twórcy LTP przy jego tworzeniu musieli dokonać wyboru odnośnie sposobu uruchamiania

testów. Wybór mianowicie dotyczył stopnia automatyzacji. Podejście manualne, bardzo elastyczne i pozwalające na dowolne personalizowanie ustawień, jednocześnie byłoby niewygodne przy testowaniu dużych systemów. Dlatego też twórcy zdecydowali się na automatyzację tego procesu, zostawiając jednakże pewne możliwości konfiguracji.

8). **LTP – Processing danych**

Jak już było wspomniane, LTP przy pomocy GCOV tworzy obraz systemu podczas normalnej pracy, tzw. Raw Coverage Data. Odfiltrowuje później te dane od informacji dotyczących pokrycia systemu w czasie działania testu, dodaje dane dotyczące przebiegu i działania tegoż testu i przetwarza je wszystkie do formatu HTML. Został on wybrany, ze względu na łatwość publikacji wyników działanie testów, co ma ułatwiać pracę grupową, a także ze względu na przenośność tego formatu.

W wyniku dostajemy

- informacje o rezultacie zakończenia się testu
- częściach jądra na które miał on wpływ
- częściach „dotkniętego" podsystemu, których test nie dotyczył
- oraz szczegółowe informacje o wykonywaniu się linii kodu (ile razy która linia została uruchomiona, z jaką częstotliwością)

9). **LTP – Rozwój**

Poniżej mamy przedstawiony diagram dotyczący pokrycia poszczególnych części systemu przez testy dostępne w LTP w wersji z roku 2002 i 2008:

LTP 2002

Directory	Coverage
fs	29,51%
include/asm	0,00%
include/linux	0,00%
include/net	0,00%
ipc	46,02%
kernel	25,63%
lib	0,00%
mm	24,50%
net	48,40%
security	0,00%

LTP marzec 2008

Directory	Coverage	
fs	52.9%	10778/20367 lines
include/asm	50.9%	613/1204 lines
include/linux	60.0%	2283/3812 lines
include/net	57.6%	1015/1762 lines
ipc	56.4%	1539/2729 lines
kernel	39.1%	10097/25837 lines
lib	43.2%	2159/4992 lines
mm	52.7%	7066/13396 lines
net	65.7%	633/964 lines
security	51.9%	666/1283 lines

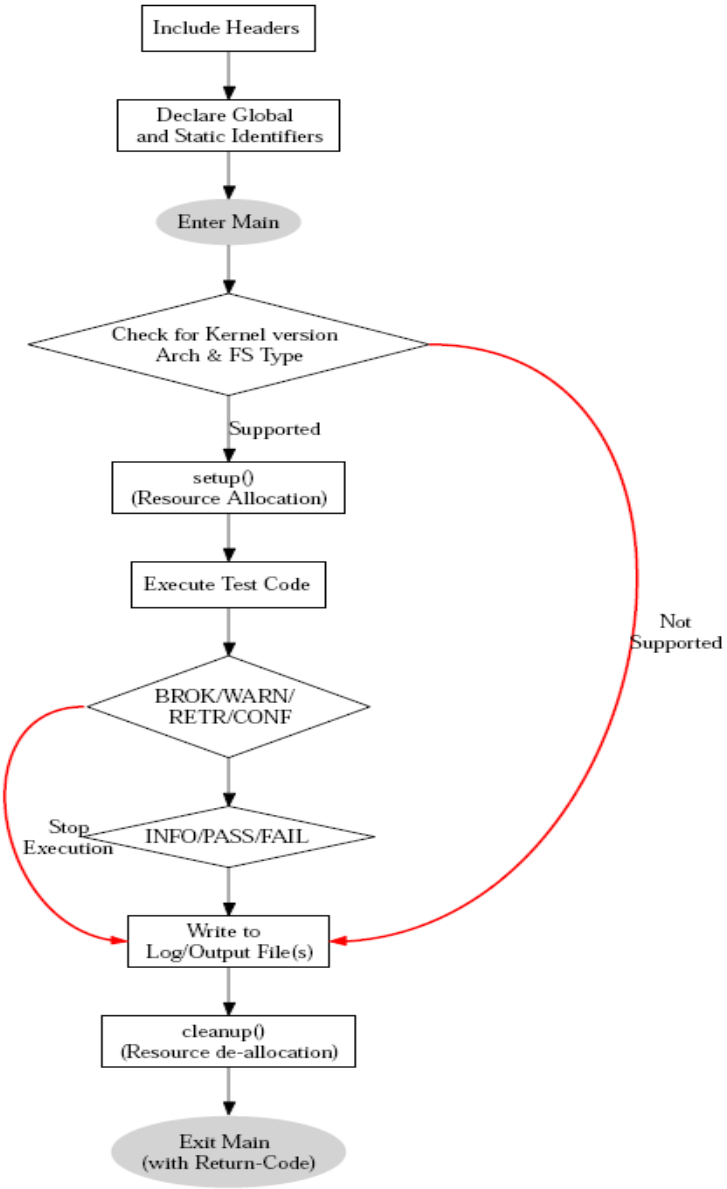
Jak widać uczyniony został ogromny postęp. Co ciekawe, w roku 2002 autorzy zakładali pokrycie równe 50%, za wiarygodne i nie wymagające dalszego rozwijania. Dzisiaj ta granica przesunęła się w okolice 66%.

Poniżej przedstawiam jeszcze jeden diagram pokazujący rozwój LTP w bardzo krótkim okresie 3 miesięcy:

Subsystem	% Increase
Filesystems	3.1
include/asm	1.5
include/linux	1.3
include/net	1.4
ipc	3.6
kernel	0.9
lib	1.0
mm	1.2
net	0.3
security	0

10). **LTP – Tworzenie własnych testów**

Oczywiście każdy może stworzyć testy, które można przesłać na stronę projektu i liczyć na włączenie do kolejnego wydania. Poniżej przedstawiam diagram działania takiego testu:



Najpierw oczywiście należy zadbać o dołączenie bibliotek LTP, oraz przydzielić naszemu testowi unikatowy numer. Oczywiście nasz test musi rozpoznać wersję i architekturę jądra. Po wykonaniu się właściwego testu musi on zapisać wszystkie dane do odpowiedniego pliku logu. Test póki co musi być napisany w C.
Bardzo ważne jest, aby uruchomienie testu nie wymagało żadnej manualnej pre-konfiguracji systemu. Takową, jeśli jest niezbędna, można przeprowadzić w treści samego testu.
LTP udostępnia nam również zestaw makr pomocnych przy pisaniu testu, jak np. TEST(), TEST_PAUSE, itd., oraz predefiniowane zmienne takie jak TEST_RETURN, czy TEST_ERRNO. Właściwy test powinien być funkcją zwracającą 1 lub 0, którą należy uruchomić wewnątrz makra TEST(). Oczywiście ustawia ona ww. zmienne.

API do tworzenia testów udostępnia nam również funkcje ułatwiające sterowanie przebiegiem

testu. Np. przy użyciu funkcji

```
char *parse_opts(int argc, char *argv[],
        option_t option_array[],
        void (*user_help_func)());
```

możemy w łatwy sposób mieszać i przekazywać opcje sterowania testem z opcjami samego testu. Opcje sterowania testem, to np. utworzenie n instancji testu działających równolegle itp.

11). **LTP – przykładowe wyjście**

LTP Output/Log (Report Generated on Mon Mar 31 12:14:29 CDT 2008)

PASSED	FAILED	WARNING	BROKEN	RETIRED	CONFIG-ERROR

Click Here for Detailed Report
Click Here for Summary Report

Detailed Report

No	Test-Name	Command-Line	Test-Output	Termination-id
1	abort01	ulimit -c 1024;abort01	abort01 1 PASS : Test passed	0
93	faccessat01	faccessat01	faccessat01 6 FAIL : faccessdat() Failed, errno=20 : Not a directory	1
94	fallocate01	fallocate01	fallocate01 0 WARN : System doesn't support execution of the test	0
183	ftruncate04	ftruncate04	ftruncate04 1 CONF : The filesystem where /tmp is mounted does not support mandatory locks. Cannot run this test.	0

Summary Report

Test Summary	Pan reported some Tests FAIL
LTP Version	LTP-20080331
Start Time	Mon Mar 31 12:14:29 CDT 2008
End Time	Mon Mar 31 13:05:21 CDT 2008
Log Result	/root/subrata/ltp/ltp-full-20080331/results
Output/Failed Result	/root/subrata/ltp/ltp-full-20080331/output
Total Tests	860
Total Failures	3
Kernel Version	2.6.21.3
Machine Architecture	i386
Hostname	sniff

LTP GCOV extension - code coverage report

Current view: overview - kernel
Test: gcov.1
Date: 2004-05-17 Instrumented lines: 9195
Code covered: 47.3 % Executed lines: 4353

Filename	Coverage	
capability.c	73.0 %	65 / 89 lines
cpu.c	0.0 %	0 / 28 lines
dma.c	0.0 %	0 / 27 lines
exec_domain.c	0.0 %	0 / 99 lines
exit.c	78.6 %	429 / 546 lines
extable.c	35.7 %	5 / 14 lines
fork.c	80.1 %	474 / 592 lines
futex.c	46.1 %	125 / 271 lines

Powyżej przedstawione są przykładowe wyjścia LTP. Po kliknięciu w żądaną bibliotekę w oknie raportu GCOV, otrzymamy szczegółowe informacje dotyczące jej pokrycia.

12). **LTP – Co dalej?**

Projekt LTP jest cały czas rozwijany. Oto główne cele postawione przez jego developerów:

- Umożliwienie pisania testów w Perlu i być może innych językach
- Dodanie nowych obszarów dostępnych do testowania. Przede wszystkim modułu odpowiedzialnego za zarządzanie mocą, ale także KDUMP i innych
- Przetwarzanie zebranych danych nie tylko do formatu HTML, ale również XML
- Dodanie testów wydajnościowych systemu
- A przede wszystkim pisanie coraz większej ilości testów, aby pokryć około 2/3 kodu wszystkich modułów na które testy mają wpływ

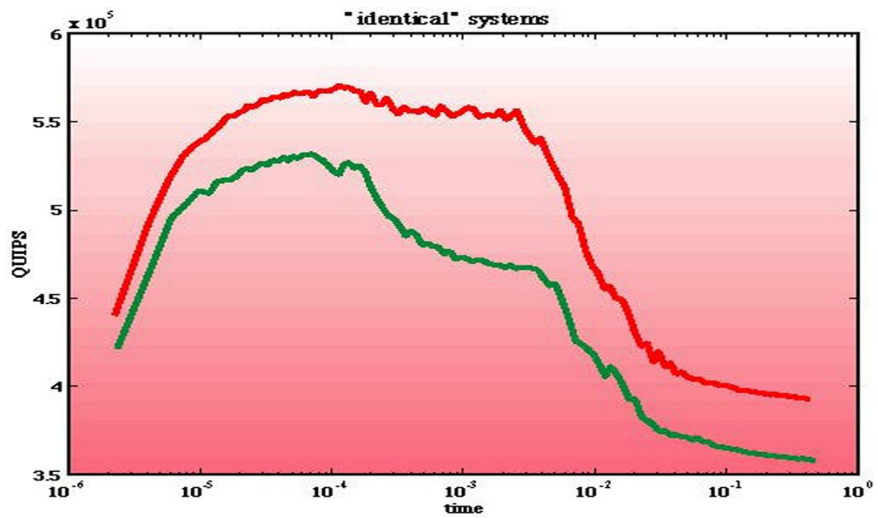
13). **Linux Benchmark Suite**

Jest to zbiór testów niejako uzupełniający LTP. Zawiera dużo mniej testów, ale ma między innymi:

- Kernel Lockmetering, czyli testowanie i zbieranie statystyk używania semaforów i spin-locków przez jądro
- Kernele Profiling, czyli szczegółowe informacje o aktywności jądra
- UnixBench, czyli zbiór benchark'ów testujący procesor i operacje wejścia wyjścia pod zróżnicowanym obciążeniem
- SPEC CPU2000, benchmark, który oprócz testowania CPU potrafi również mierzyć wydajność kompilacji

14). **HINT**

Jest to benchmark stworzony na zlecenie departamentu energii USA. Według zapewnień jego twórców używa zupełnie innego podejścia do mierzenia wydajności systemu, co przekłada się na większą dokładność, lepszą skalowalność i wysoką wiarygodność. Owo podejście zostało nazwane QUIPS, czyli QUality Improvement Per Second. W uproszczeniu jest to ilość uszczegółowień pewnej funkcji w czasie. Funkcja ta jest identyczna dla wolnych systemów oraz dla superkomputerów (w przeciwieństwie do innych benchmarków), przez co wyniki są bardzo łatwo porównywalne. Poniżej przedstawiony jest przykładowy rezultat działania tego testu. Co ciekawe był on wykonany na dwóch, teoretycznie identycznych maszynach (ten sam software i hardware).



Pokrycie kodu testami (GCOV)

1. **Czym jest COV?**

   Pokrycie kodu (ang. code coverage) jest miarą używaną w testowaniu oprogramowania. Opisuje ona stopień w którym sam kod programu został przetestowany. Ponieważ sprawdzanie kodu następuje w sposób bezpośredni, tę formę testów należy zaliczyć do rodzaju *white box testing*. (testy białej skrzynki) Warto dodać, że pokrycie kodu testami jest jedną z pierwszych wprowadzonych technik systematycznego testowania oprogramowania.

2. **Kryteria pokrycia**

   W celu zmierzenia, jak dokładnie zestaw testów sprawdza program, używa się jedego lub wielu kryteriów pokrycia. Można wymienić ich wiele, to są najważniejsze z nich:

   - Pokrycie funkcji (function coverage) – czy każda funkcja w programie została wykonana?
   - Pokrycie instrukcji (statement coverage) – czy każda instrukcja kodu źródłowego została wykonana?
   - Pokrycie decyzji/rozgałęzień (decision/branch coverage) – czy każda struktura kontrolna (jak na przykład instrukcja if) wyliczyła się zarówno do true, jak i false?
   - Pokrycie warunków (condition coverage) – czy każde podwyrażenie boolowskie wyliczyło się do true jak i do false? (to implikuje oczywiście pokrycie decyzji)
   - Pokrycie ścieżek (path coverage) – czy każde możliwe przejście przez dany fragment kodu zostało wykonane?
   - Pokrycie wejścia/wyjścia (entry/exit coverage) – czy każde możliwe wywołanie i każdy możliwy powrót z wywołania funkcji zostały wykonane?

   Aplikacje o znaczeniu krytycznym dla bezpieczeństwa (safety-critical) najczęściej muszą osiągnąć 100% pokrycia niektórych z powyższych kryteriów.

   Niektóre z kryteriów są połączone, na przykład, pokrycie ścieżek implikuje pokrycie decyzji, instrukcji i wejścia/wyjścia, a pokrycie decyzji wymusza pokrycie instrukcji, ponieważ każda instrukcja jest częścią jakiegoś rozgałęzienia.

   Łatwo się domyślić, że pełne pokrycie kodu testami jest zwykle niepraktyczne, a nawet niemożliwe. Każdy moduł posiadający n decyzji może mieć do 2^n ścieżek do przetestowania, a pętle mogą implikować nieskończone ilości możliwych przejść programu. (wiele modułów działa przecież w nieskończonych pętlach)

   Wiele ścieżek może też być niewykonywalnych dla żadnego możliwego wejścia programu, czyli mogą nie istnieć dane testowe które spowodują wykonanie pewnego fragmentu kodu. Niestety, ogólny algorytm szukania nieosiągalnych ścieżek nie istnieje (ponieważ mógłby być użyty jako rozwiązanie problemu stopu, którego brak udowodnił w 1936 roku Alan Turing). Praktycznie zatem stosuje się techniki, pozwalające na grupowanie ścieżek w klasy, różniące się jedynie ilością wykonań pętli, a następnie pokrywanie każdej z klas ścieżek, nie zaś każdej pojedynczej ścieżki.

4. **Analiza pokrycia testami kodu jądra**

   Osiągnięcie właściwego pokrycia testami kodu jądra należy do obowiązków zestawu testów systemu. Mimo że grupa testów może obejmować intensywne użycie wszystkich czterech głównych zasobów systemowych, niekoniecznie będzie ona wykorzystywać więcej niż małą

część kodu kernela. Jeżeli zatem chcemy posiadać test z prawdziwego zdarzenia, a nie jedynie generator obciążenia systemu, nie możemy zapomnieć o pokryciu kodu testami. Można wymienić dwa open-source'owe narzędzia pomocne w tym zadaniu:

- **gcov**: Program rozwijany głównie przez Linux Test Project, umożliwiający rozpoznanie ile razy dana linia kodu, funkcja lub gałąź aplikacji (w naszym wypadku – jądra) zostały wywołane. Użycie gcov-a wymaga skompilowania programu za pomocą gcc z użyciem odpowiednich flag. Zatem w naszym przypadku wymaga rekompilacji jądra, oraz dodatkowo nałożenia na nie odpowiedniej łatki. Gcov za pomocą systemu /proc potrafi wygenerować odpowiednie statystyki.

  Przykład wyników gcov dla programu tmp.c:

```
            -:    0:Source:tmp.c
           -:    0:Graph:tmp.gcno
           -:    0:Data:tmp.gcda
           -:    0:Runs:1
           -:    0:Programs:1
           -:    1:#include <stdio.h>
           -:    2:
           -:    3:int main (void)
function main called 1 returned 1 blocks executed 75%
           1:    4:{
           1:    5:  int i, total;
           -:    6:
           1:    7:  total = 0;
           -:    8:
          11:    9:  for (i = 0; i < 10; i++)
branch  0 taken 91% (fallthrough)
branch  1 taken 9%
          10:   10:    total += i;
           -:   11:
           1:   12:  if (total != 45)
branch  0 taken 0% (fallthrough)
branch  1 taken 100%
    #####:   13:    printf ("Failure\n");
call    0 never executed
           -:   14:  else
           1:   15:    printf ("Success\n");
call    0 called 1 returned 100%
           1:   16:  return 0;
           -:   17:}
```

- **lcov**: Aplikacja stworzona przez IBM i rozwijana przez Linux Test Project, będąca rozszerzeniem gcov-a. Zawiera zestaw skryptów w perlu budujących na podstawie tekstowych wyników gcov-a graficzną prezentację w formacie HTML. Wynik obejmuje pokrycie procentowe, wykresy i strony statystyk, umożliwiające sprawne przeglądanie uzyskanych danych.

Po załadowaniu modułu gcov-a, należy uruchomić wszystkie testy jądra z naszego zestawu. Mimo że w rzeczywistości część testów mogłaby być wykonywana równolegle, należy to zrobić iteracyjnie. Każdy test powinien być uruchomiony dokładnie raz. Takie pojedyncze, iteracyjnie wywoływanie ma służyć zmniejszeniu ilości nieprzewidywalnych i niecelowych wywołań kodu jądra, które próbowałoby zbalansować liczne i równoległe wątki różnych testów.
Po zakończeniu testów można przejść do analizy uzyskanych danych za pomocą lcov-a. Wyniki tego ostatniego obejmują każdą linię kodu jądra, której wywołania są agregowane w wyniki procentowe dla danych sekcji i plików kernel-a. Przykład pokazujący, jak takie wyniki mogą wyglądać:

LTP GCOV extension - code coverage report

Current view:	directory	Executed lines:	63612 / 256879
Test:	ppc64branch.info	Branches taken:	10110 / 72549
Date:	2004-01-08	Executed functions:	4129 / 12754
Code covered:	22.8 %		

Directory name	Coverage			
	Total	Statement	Branch	Function
arch/ppc64/kernel	36.1 %	38.9 %	23.4 %	44.9 %
arch/ppc64/lib	97.4 %	100.0 %	83.3 %	100.0 %
arch/ppc64/mm	41.0 %	43.7 %	28.0 %	52.3 %
arch/ppc64/oprofile	60.0 %	66.7 %	No branches	50.0 %
arch/ppc64/xmon	0.1 %	0.1 %	0.0 %	0.9 %
drivers/base	35.0 %	37.2 %	22.9 %	40.6 %
drivers/base/power	7.4 %	10.0 %	0.0 %	0.0 %
drivers/block	32.5 %	34.5 %	23.3 %	43.7 %
drivers/cdrom	3.4 %	4.6 %	0.3 %	6.8 %
drivers/char	38.4 %	42.6 %	25.2 %	45.7 %
drivers/gcov	57.9 %	63.0 %	40.4 %	75.0 %
drivers/input	37.8 %	43.9 %	19.3 %	54.1 %

```
                :
                : int init_module(void)
            1   : {
            1   :         int i;
                :
           11   :         for(i=0; i<10; i++) {
           10   :                 printk("This line should be executed 10 times\n");
           10   :                 printk("The value of i is %d\n",i);
                :
                :                 /* Comments are not even considered */
           10   :                 if (i == 100) {
            0   :                         printk("This line should have no coverage\n");
                :                 }
                :         }
                : }
```

Projektanci lcov-a zdefiniowali „sastysfakcjonujące pokrycie” (kolor zielony) w sposób odgórny, dlatego nie powinien to być wyznacznik jakości testu. Jednakże dostępne dokładne statystyki pozwalają każdemu na własną ocenę stanu pokrycia kodu testami. Tworzący zestaw testów może przeanalizować wyniki i uwzględnić widoczne braki za pomocą dodatkowych testów dołączonych do zestawu.

5. **Profilowanie czasu wykonania**

   Często wraz z programami testującymi pokrycie kodu (jak gcov) używa się programów sprawdzających czas wykonywania poszczególnych linii kodu , funkcji lub modułów(np. Gprof). Dzięki temu tzw. wąskie gardła (ang., bottle neck) mogą zostać szybciej zidentyfikowane i zlikwidowane.

Tracing

1. **Czym jest tracing?**

   W inżynierii oprogramowania, tracingiem nazywamy logowanie informacji o wykonaniu programu w trakcie jego wykonywania, głównie do celów odpluskwiania programów.

2. **logging vs tracing**

Logowanie zdarzeń	Tracing
Głównie używane przez administratorów systemów	Głównie używany przez developerów
Logowanie informacji wysokopoziomowych (jak np. błąd przy instalacji programu)	Logowanie informacji niskopoziomowych (jak np. rzucony wyjątek)
Nie może być zbytnio „rozrzutny” (zawierać wielu zduplikowanych informacji lub informacji nieprzydatnych dla użytkowników)	Może sobie na to pozwolić
Wymagany najczęściej ustalony format wyników	Niewielkie ograniczenia na format wyników
Wiadomości w wersjach językowych	Rzadko jest to brane pod uwagę
Elastyczność ze względu na nowe rodzaje zdarzeń nie jest wymagana	Elastyczność ze względu na nowe rodzaje zdarzeń jest kluczowa

   Tracing jest gorącym tematem w społeczności linuxowej, głównie ze względu na to, że Sun Microsystems wkłada mnóstwo energii w reklamowanie Solarisowego Dtrace'a. W Linuxie jednak również nie brakuje narządzi do trace-owania, natomiast dostępność ich dla szerszej publiczności powinna zostać poprawiona.

3. **Najważnejsze cechy narzędzia do trace'u:**
   - Zajrzenie w głąb działającego systemu
   - Wyszukiwanie wąskich gardeł systemu
   - Śledzenie wykonań procesów i środowiska w którym się wykonują (coś jak zaglądanie pod maskę samochodu jadącego 160 km/h)
   - Tryb flight-recorder – czarna skrzynka samolotu – obejrzenie archiwalnych danych w wypadku, gdy coś pójdzie źle.
   - Możliwość zaglądania zarówno w przestrzeń użytkownika jak i jądra

4. **Przykładowe narzędzia do tracingu:**
   - SystemTap

     SystemTap to open-sourcowe narzędzie zapewniające prosty interfejs poprzez linię poleceń i własny język skryptowy. Aktualnie wykorzystywany m.in. w: Red Hat, IBM, Intel, Hitachi i Oracle.

Przykład skryptu:

```
probe syscall.open
{
  printf ("%s(%d) open (%s)\n", execname(), pid(), argstr)
}
probe timer.ms(4000) # after 4 seconds
{
  exit ()
}
```

I jak to działa w praktyce:

```
vmware-guestd(2206) open ("/etc/redhat-release", O_RDONLY)
hald(2360) open ("/dev/hdc", O_RDONLY|O_EXCL|O_NONBLOCK)
hald(2360) open ("/dev/hdc", O_RDONLY|O_EXCL|O_NONBLOCK)
hald(2360) open ("/dev/hdc", O_RDONLY|O_EXCL|O_NONBLOCK)
df(3433) open ("/etc/ld.so.cache", O_RDONLY)
df(3433) open ("/lib/tls/libc.so.6", O_RDONLY)
df(3433) open ("/etc/mtab", O_RDONLY)
hald(2360) open ("/dev/hdc", O_RDONLY|O_EXCL|O_NONBLOCK)
```

- Dtrace for Linux

  DTrace, inaczej Dynamic Tracing Framework, to zaawansowane narzędzie służące do monitorowania i diagnozowania działania systemu i aplikacji, opracowane przez firmę Sun dla systemu operacyjnego Solaris. Prace nad wersją linuxową wciąż trwają, działa ona już dla wielu wersji jądra, jednak bez pełnej funkcjonalności.

- Kprobes

  Kernel Dynamic Probes (Kprobes) posiada lekki interfejs do umieszczania w kodzie tzw. probów i ustawiania ich obsługi. Probe to automatyczny breakpoint dynamicznie wstawiany w wybrany moduł działający w przestrzeni jądra, bez konieczności modyfikacji kodu źródłowego.

- LTTng (Linux Trace Toolkit) i LTTV (Linux Trace Toolkit Viewer)

  Jak sama nazwa wskazuje, LTTng jest programem do tracowania jądra systemu Linux. LTTV analizuje i przedstawia w sposób graficzny wyniki.

  W przeciwieństwie do kprobes, LTTng jest statyczny, w tym sensie, że probe'y muszą być na stałe umieszczone w kodzie jądra. Jego główne założenia dotyczą prostoty i wysokiej wydajności.

Przykładowe testy jądra

1. **Bootchart**

   Bootchart to narzędzie do analizy wydajności i wizualizacji procesu bootowania systemu Linux. Umożliwia ono zbadanie użycia zasobów (jak procesor i dysk twardy) oraz przyjrzenie się dokładnie, jakie procesy i w jakiej kolejności są uruchamiane przy ładowaniu systemu.

   Projekt ten powstał jako odpowiedź na wyzwanie postawione przez Owena Taylora na liście mailingowej developerów Fedory: (tłumaczenie własne)

   „Wyzwaniem jest stworzenie aplikacji ukazującej na jednym rysunku, co dzieje się podczas bootowania systemu i wskazanie szans na optymalizację tego procesu, poprzez ukazanie jak bardzo rzeczywiste bootowanie odbiega od idealnego, w którym nieustannie występuje 100% zapotrzebowanie na procesor i dostęp do dysku."

   Bootchart używa skryptu powłoki uruchamianego przez jądro w fazie `init`. Skrypt działa w tle i zbiera z systemu plików `/proc` informacje o procesach, statystyki użycia CPU i dysku. Dane są przechowywane w pamięci i zapisywane na dysk po zakończeniu ładowania systemu operacyjnego. Logi są następnie za pomocą aplikacji napisanej w Jave przetwarzane do graficznego formatu: PNG, SVG lub EPS. Wykres może być użyty do analizy zależności procesów i ogólnego zużycia zasobów systemowych w trakcie bootowania systemu.

   Jak to działa?

   System zamiast `/sbin/init` uruchamia logger `/sbin/bootchartd` przy starcie systemu, co uzyskujemy za pomocą odpowiedniej modyfikacji GRUB-a lub LILO. Program `/sbin/bootchartd` uruchamia od razu `/sbin/init,` ale jednocześnie zaczyna zbierać dane do wirtualnego systemu plików (tmpfs), ponieważ partycja root-a jest w trakcie bootowania zamontowana w trybie tylko do odczytu.

   Gdy tylko system plików `/proc` zostanie zamontowany, logger zbiera informacje z wielu plików:

   - `/proc/stat` – statystyki CPU dla: usera, systemu, IO, idle
   - `/proc/diskstats` – statystyki dyskowe: aktualne użycie i przepustowość (tylko w jądrach 2.6)
   - `/proc/[PID]/stat` – informacje o działających procesach: czas uruchomienia, PID rodzica, stan procesu, użycie CPU, itp.

   Zawartość tych plików jest dopisywana do odpowiednich logów, domyślnie co 0.2 sekundy. Zakończenie bootowania bootchart wykrywa poszukując specyficznych procesów, jak np. `gdmgreeter`, `kdm_greet`, itp. (run-level 5).

   Warto zwrócić uwagę na to, że krótko żyjący proces może zakłócić spójność zbieranych informacji, ponieważ nie zostanie zauważony przez logger. Jest jednak możliwość poradzenia sobie z tym problemem za pomocą odpowiedniej konfiguracji bootcharta i jądra – mianowicie użycia BSD process accounting v3 – oraz instalacji pakietu `psacct` lub `acct.`

   Wizualizacja

   Plik tar zawierający logi jest przekazywany aplikacji Javowej w celu przetworzenia danych. Typowa sekwencja bootowania używa setek procesów, zatem wykres Gantt'a

przedstawiający ładowanie systemu musi zostać odpowiednio uproszczony, co sprowadza się do usuwania krótko żyjących i bezczynnych procesów z wykresu, jak również grupowania podobnych procesów działających równolegle.

2. **Volanomark**

   Javowy benchmark mierzący wydajność serwera i skalowalność sieci. Tworzy połączenia między 20 osobowymi grupami klientów i oblicza jak długo zajmuje rozpowszechnienie wiadomości w grupach. Wynikiem jest średnia ilość wiadomości transerowanych przez serwer na sekundę.

3. **Lmbench**

   Zestaw prostych, przenośnych testów wydajnościowych kluczowych opcji systemu, zawiera testy obsługujące m.in.:

   Mierzenie wydajności:

   - odczytu plików w cache'u
   - kopiowania pamięci (bcopy)
   - odczytów z pamięci
   - zapisów do pamięci
   - pipe'ów
   - TCP

   Mierzenie opóźnień:

   - przełączania kontekstu
   - operacji sieciowych: ustanawiania połączeń, pipe'ów, TCP, UDP i RPC hot potato
   - tworzenia i usuwania systemów plików
   - tworzenia procesów
   - obsługi sygnałów
   - wywoływania funkcji systemowych

4. **netperf**

   Test sprawdzający wydajność różnych typów połączeń sieciowych. Mierzy przepustowość pośrednią, jak również opóźnienia całościowe (end-to-end). Środowiska obsługiwane przez netperf to: TCP i UDP przez BSD Sockets, DLPI, Unix Domain Sockets, Fore ATM API, oraz HP HiPPI Link Level Access.

5. **Fileio**

   Część sysbench'a, potrafiąca tworzyć różne rodzaje zestawów plików, a następnie wykonywać na nich konkretne operacje wejścia/wyjścia i badać ich wydajność.

6. **Iozone**

   Iozone to narzędzie to testowania systemu plików, generujące różnorodne operacje na plikach i umożliwiające dokładną analizę działania systemu operacyjnego związanego z zarządzaniem plikami. Testowane operacje: read, write, re-read, re-write, read backwards, read strided, fread, fwrite, random read, pread , mmap, aio_read, aio_write.

7. **Vmregress**

Testowy moduł jądra do badania zarządzania pamięcią wirtualną. (VM) Umożliwia on testowanie w sposób powtarzalny każdego z aspektów zarządzania pamięcią wirtualną, w tym rzeczywistej wydajności. Opiera się na kombinacji narzędzi z przestrzeni użytkownika i jądra w celu określenia aktualnego stanu jądra i wiarygodnego powtórzenia testów.

Używane moduły są podzielone na 4 części:

- core – zapewnia obsługę całego programu
- sense – udostępnia informacje o aktualnie działającym jądrze
- test – egzekwuje wybrane scenariusze testowe
- bench – mierzy wydajność różnych podsystemów

Interfejs VMRegress opiera się na systemie plików `/proc`.

8. Mmbench

Program ten tworzy obciążenie dla jednostki zarządzania pamięcią (MMU), podsystemu swap oraz odzyskiwania pamięci w celu zmierzenia wydajności zarządzania pamięcią przez jądro. Test używa wewnętrzego generatora liczb losowych w celu stworzenia strumienia logarytmicznie normalnego rozkładu częstotliwości dostępów do stron. Rozkład log-normalny precyzyjniej niż rozkład Zipfa odpowiada zachowaniu normalnych programów. Rozmiar używanej pamięci jest domyślnie tylko o 12% większy niż rzeczywista ilość dostępnego miejsca, w celu uzyskania krótkiego czasu wykonania programu.

Ten test zadziała na platformie Solaris, Windows i oczywiście Linux.

9. Tiobench

Tiobench potrafi zmierzyć przepustowość (MB/s) i opóźnienia (ms) funkcji odczytu/zapisu dla sekwencyjnych lub losowych prób zapisu/odczytu dla wielu równoległych wątków. Istnieje możliwość wyboru algorytmu szeregowania operacji wejścia/wyjścia. (czyli wybór spośród: anticipatory, cfq, noop, ddl scheduler)