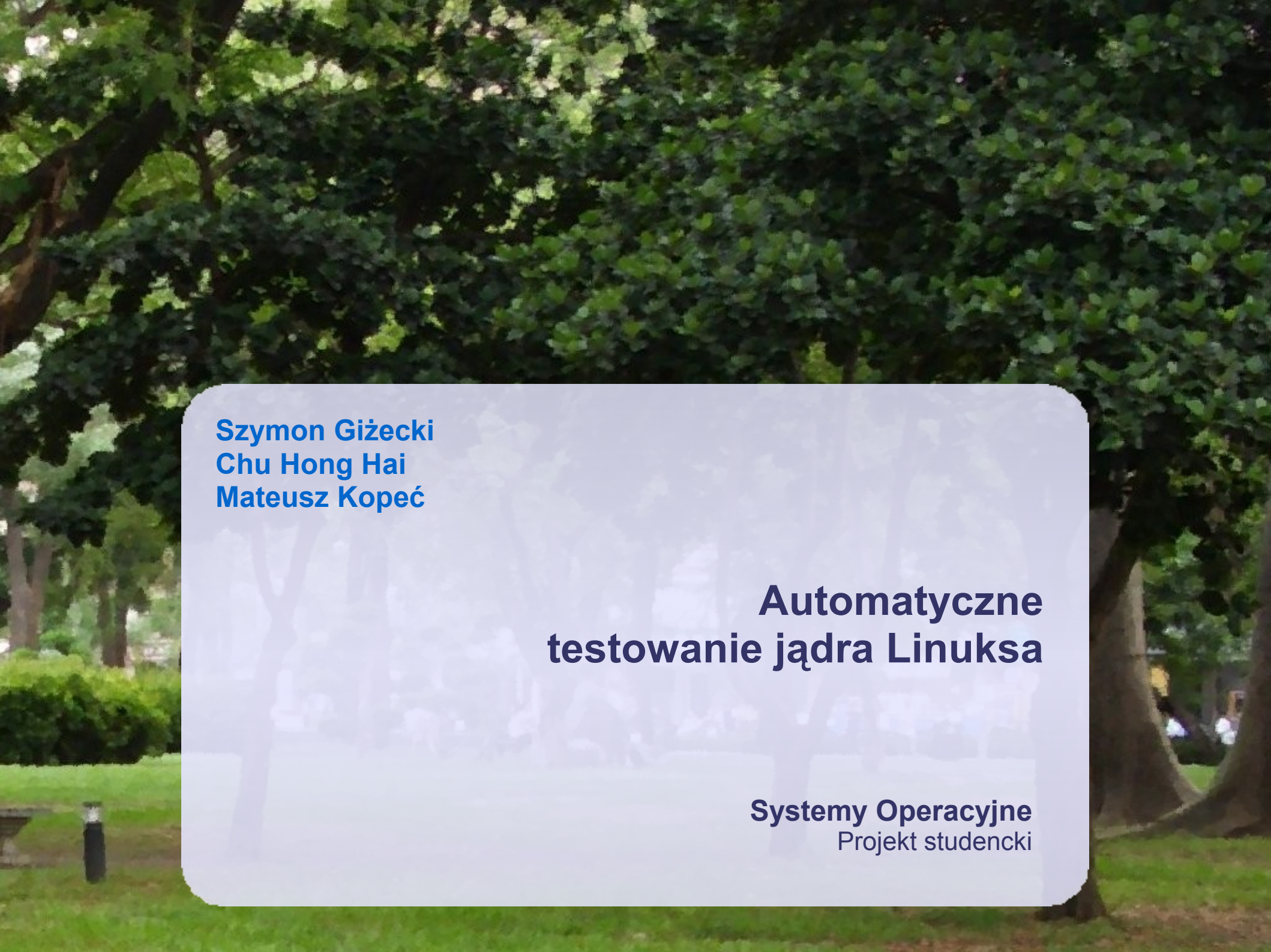


Szymon Giżecki
Chu Hong Hai
Mateusz Kopeć

Automatyczne testowanie jądra Linuksa

Systemy Operacyjne
Projekt studencki

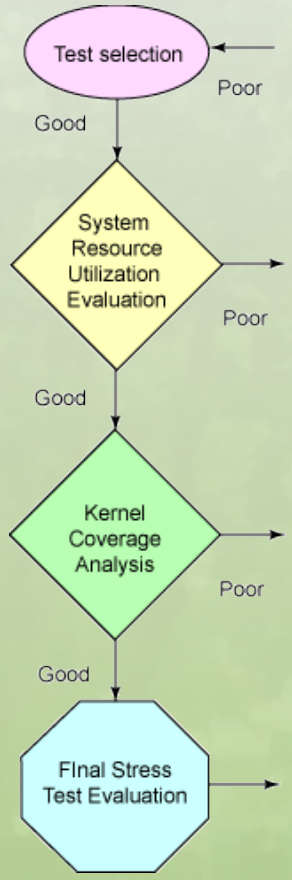
Plan prezentacji

...czyli o czym będziemy mówili:

- Metodologie testowania jądra Linuksa
- Testy poprawnościowe
- Testy wydajnościowe
- Mechanizmy testowania
- Fault Injection
- Zestawy testów – Linux Test Project
- gcov i pokrycie kodu jądra testami
- Tracing
- Analiza procesu bootowania - Bootchart

Testy poprawnościowe

- Statyczna analiza kodu
- Przygotowanie testów
- Przeprowadzenie testów



Statyczna analiza kodu

Idea: szukanie błędów przed budowaniem aplikacji

- Weryfikacja modelowa (model checking)
- Analiza przepływu danych (data-flow analysis)
- Abstract interpretation
- Asercje
- Semantyka

Narzędzia:

- Sparse
- Cppcheck
- Coverity Prevent

Wybór zestawów testowych

- Obciążenie głównych zasobów
- Pokrycie dużej części kodu
- Automatyzacja
- Publikacja wyników

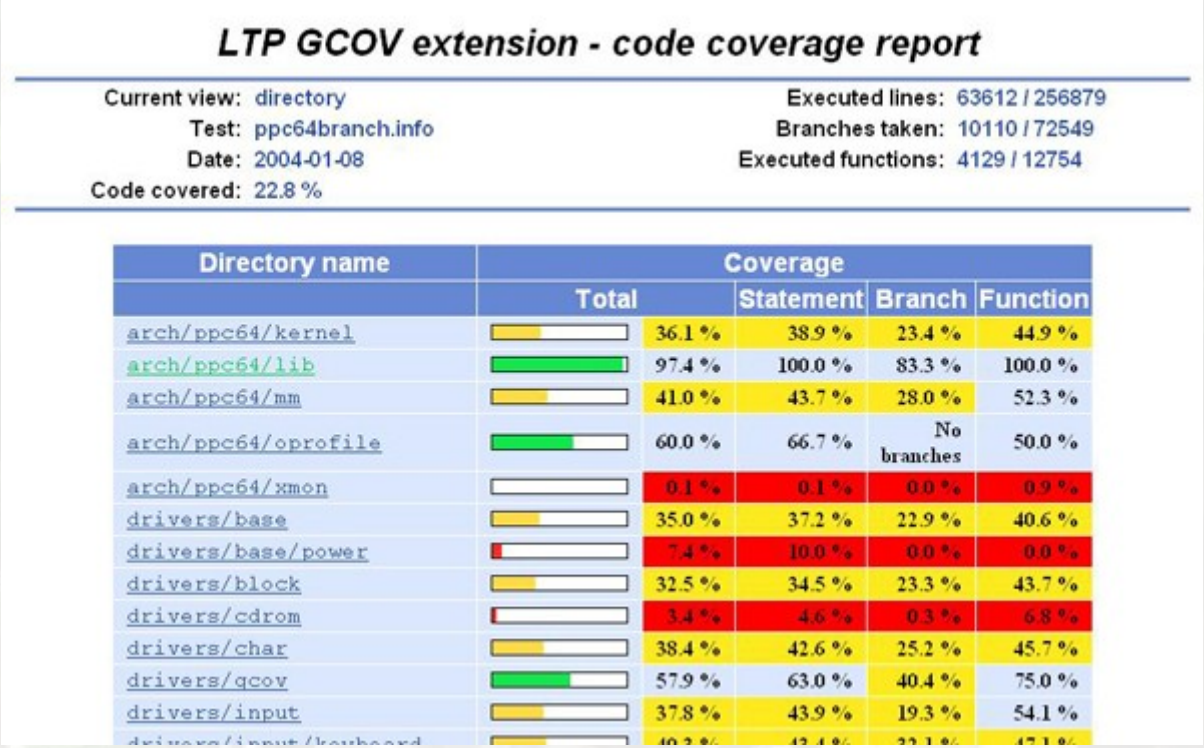
Ocena wykorzystania zasobów

- CPU
- Pamięć
- I/O
- Networking

Narzędzia: **top**, **sar** i **iostat (iostat -x 1)**

Analiza pokrycia kodu jądra

- gcov
- lcov



LTP GCOV extension - code coverage report

Current view: directory
Test: ppc64branch.info
Date: 2004-01-08
Code covered: 22.8 %

Executed lines: 63612 / 256879
Branches taken: 10110 / 72549
Executed functions: 4129 / 12754

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 %

Ocena końcowego testu

Przed przeprowadzeniem testów:

- Uruchomienie wybranego zestawu na stabilnej wersji jądra
  - Wykrycie konfliktów między testami
  - Usunięcie
- Zbieranie statystyk narzędziem **sar**
  - Punkt odniesienia dla przyszłych “prawdziwych” testów

Bardzo ważne pytanie: skąd wziąć testy?

Odpowiedź: LTP

Przeprowadzenie testu

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 konfiguracji systemu.

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

Testy wydajnościowe

- Co mierzymy
- Narzędzia

Co mierzymy

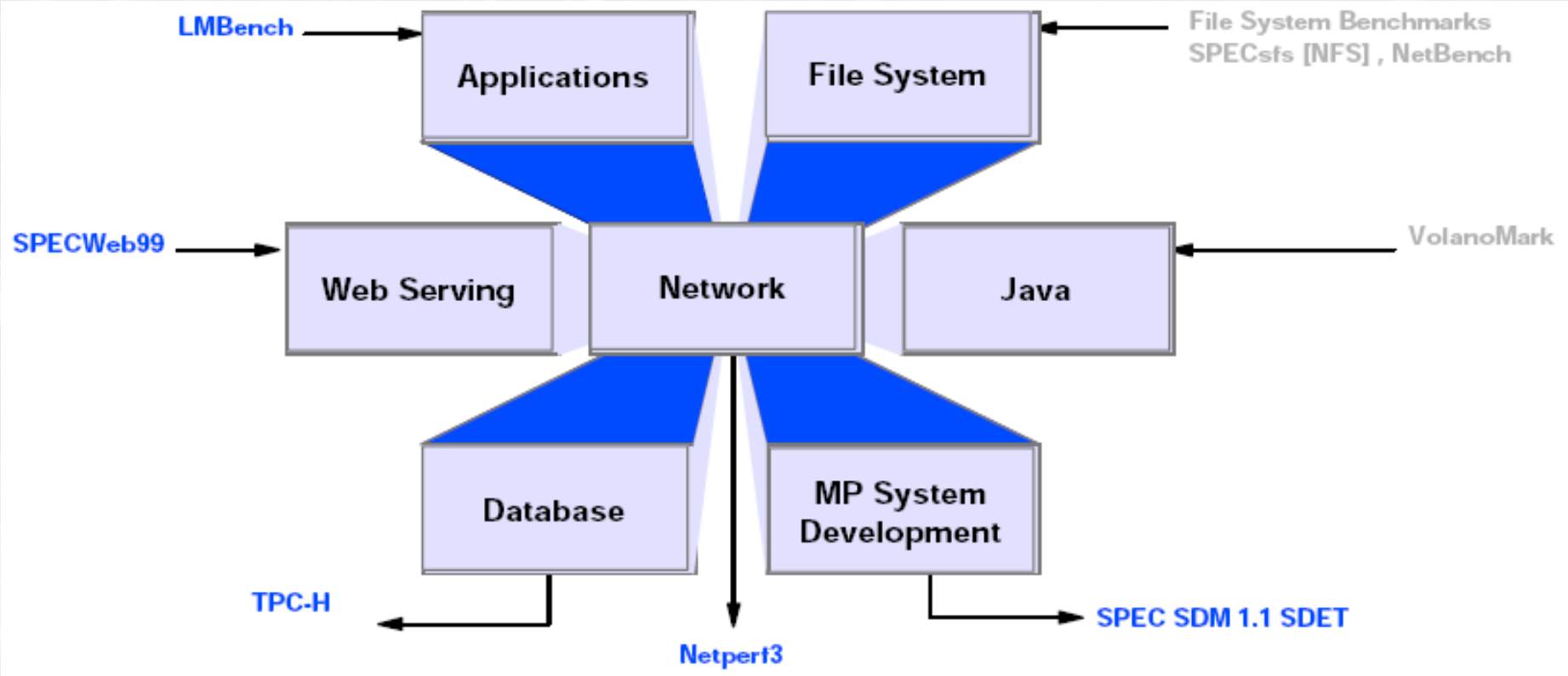
- Określenie miary np. ilość wysłanych wiadomości na sekundę
- Dobór odpowiednich benchmarków
- Tworzenie dokumentacji
- Wybór danych, które będą zbierane

Zanim uruchomimy testy musimy odpowiednio dostosować sprzęt i oprogramowanie.

Narzędzia

- Wirtualny system plików /proc
- Lockmeter (SGI)
- Kernprof (SGI)
- Trace facility (IBM)
- sstat
- schedret
- acgparse

Narzędzia komercyjne



Mechanizmy testowania – Fault Injection

- Czym jest mechanizm fault injection
- Rodzaje fault injection
- A jak to się robi w Unix'ie?
  - Wparcie w Kernel'u
  - FERRARI – A Flexible Software-Based Fault and Error Injection System

Linux Test Project

- **Linus's Law** *(according to Eric S. Raymond)*

  “Given enough eyeballs, all bugs are shallow"

- Projekt założony w roku 2000 przez konsorcjum SGI (Silicon Graphics)
- Pierwszy zunifikowane podejście do testowania jądra linuxa
- Utrzymywany jako projekt OpenSource
- Na dzień dzisiejszy największy projekt testujący jądro

Linux Test Project

- Napisany głównie w C (~94.5%), pozostałe części Perl (~0.5%) oraz Shell (~5%)
- Ponad 3000 testów
- Kto i kiedy powinien używać
- Środowisko
- Zanim uruchomimy LTP

LTP – rodzaje testów

- Unit testing
- Integration testing
- System testing
- Regression testing
- Stress testing
- Nie ma *compilation* ani *performance*

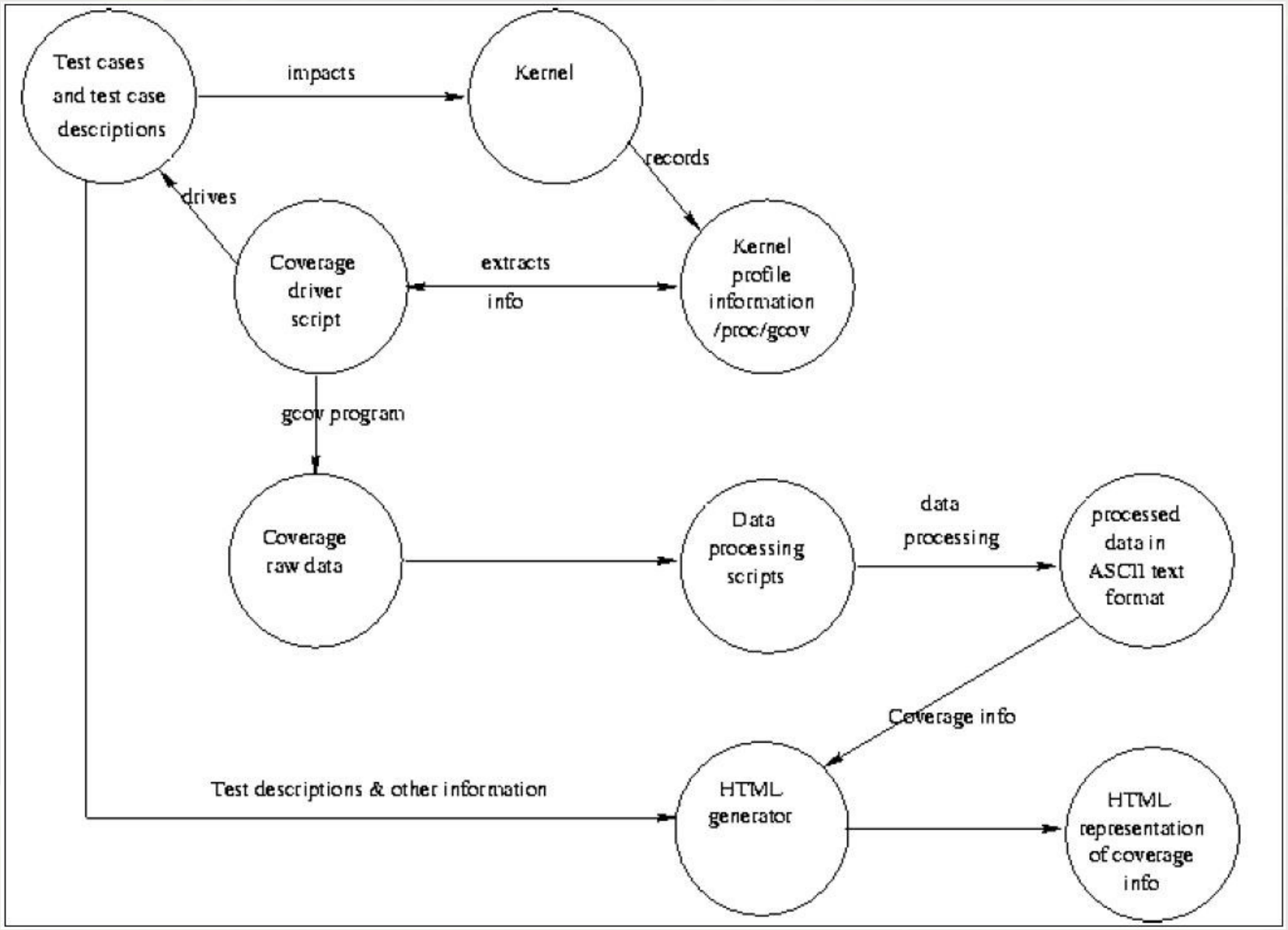
LTP – zbiór testowanych właściwości

- Przejrzysta struktura

- LTP testuje:

  Zarządzanie pamięcią, Scheduler, Wątki jądra, System plików, LVM , EVMS, Kompresje, /proc, Zdalne polecenia, Sockets, NFS, NIS, IP/V6, Sieć (Ethernet i Wireless), FDDI, ATM, Napędy (CDROM, DVD, FDD, a nawet Tape), IDE, Sterowniki SCSI / RAID

LTP – jak to działa?



LTP – Zbieranie danych

- Podejścia do zbierania danych
  - Podział ze względu na logikę i funkcjonalności.
  - Łatwe do zdefiniowania relacje
  - „Drzewo”
  - Pół-automatyzacja
  - Dane jako kod ASCII

LTP – Processing danych

- Przetwarzanie danych
  - Raw test data + coverage data + raw coverage data
  - W wyniku dokument HTML, zawierający:
    - Części kernela, na które test miał wpływ
    - Części kernela z powiązanego podsystemu, na które test nie miał wpływu
    - Częstotliwość i liczba uruchomionych linii kodu

LTP – rozwój 1

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

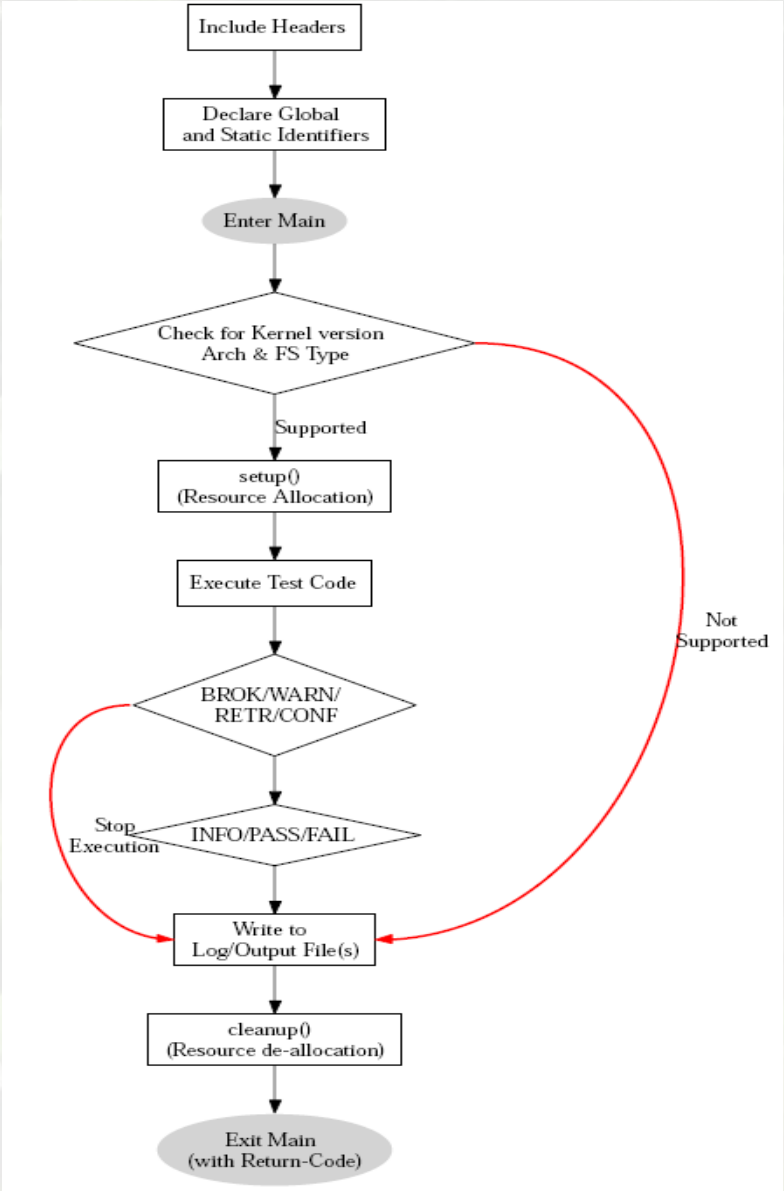
Directory	Coverage	
fs	52.9%	10778/20367 lines
include/asm	50.9%	613/1204 lines
include/linux	60.0%	2283/3812 lines
include/net	57.6%	1015/1762 lines
ipc	56.4%	1539/2729 lines
kernel	39.1%	10097/25837 lines
lib	43.2%	2159/4992 lines
mm	52.7%	7066/13396 lines
net	65.7%	633/964 lines
security	51.9%	666/1283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

LTP – rozwój 2

Przyrost pomiędzy grudniem 2007
a marcem 2008

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

LTP – Tworzenie własnych testów



- Napisane w C
- - macra TEST() i zmienne TEST_RETURN TEST_ERRNO
- Nie powinien wymagać żadnej pre-konfiguracji

LTP – Tworzenie własnych testów

Bardzo przydatna funkcja parse_opts

```
#include "test.h"
#include "usctest.h"
char *parse_opts(int argc, char *argv[],
                 option_t option_array[],
                 void (*user_help_func)());
typedef struct {
   char *option;
   int  *flag;
   char **arg;
} option_t;
```

LTP – Przykładowe wyjście 1

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

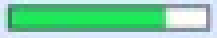
Summary Report

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 – Przykładowe wyjście 2

LTP GCOV extension - code coverage report

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

LTP co dalej?

- Dodanie wsparcia dla różnych języków
- Dodanie nowych obszarów, np. zarządzania mocą, KDUMP, kontrolerów
- Nowe techniki raportowania, np. XML
- Dodanie benchmarków
- Oczywiście rozwój testów, w szczególności duży nacisk kładziony na *functional* i *regression testing*.

Jeśli nie LTP, to: Linux Benchmark Suite

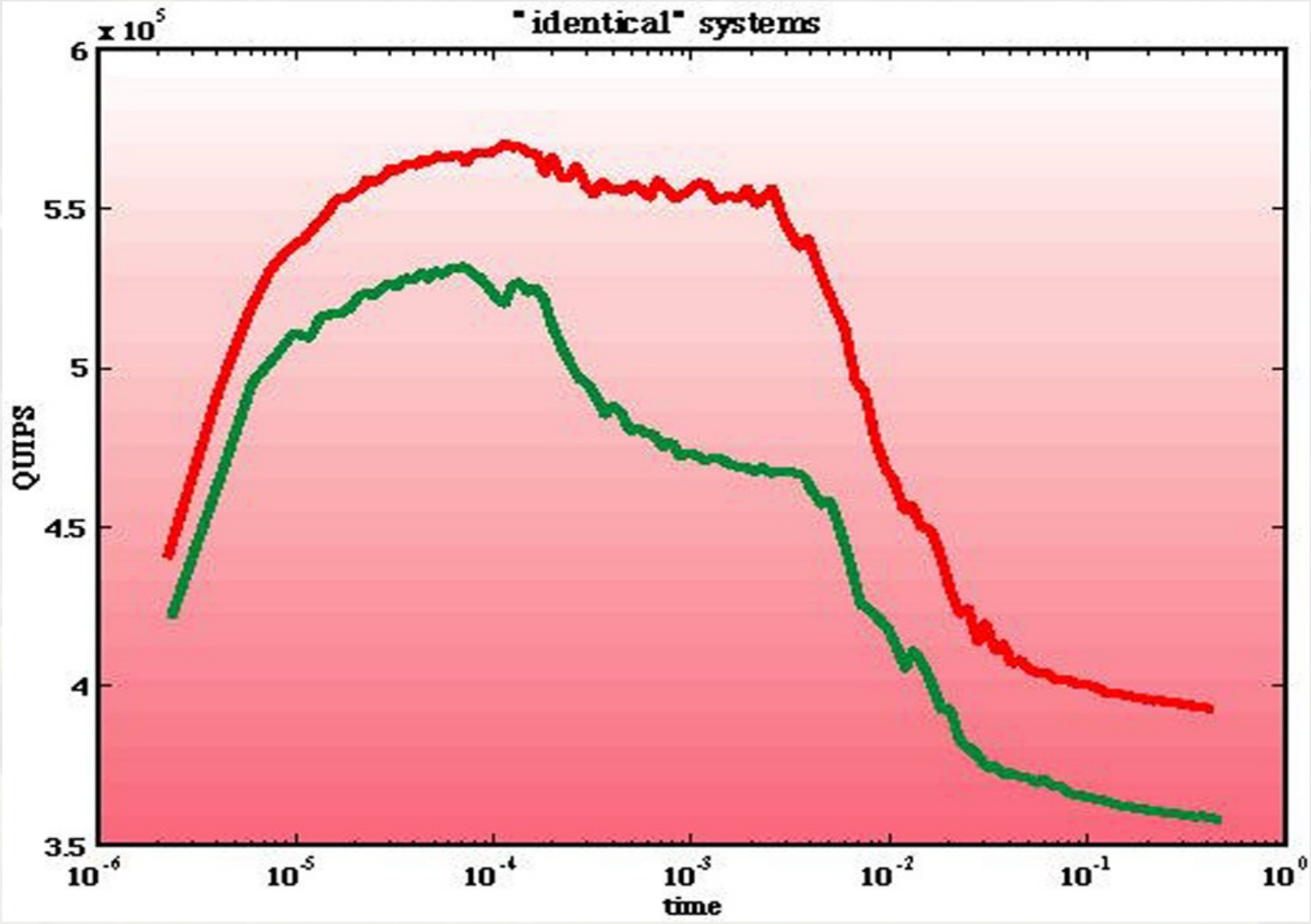
Czego nie ma w LTP:

- Kernel Lockmetering
- Kernel Profiling
- UnixBench – Wysokopoziomowy zbiór benchmarków integrujący testy CPU i I/O, oraz zachowanie systemu pod zróżnicowanym obciążeniem
- SPEC CPU2000 – Benchmark testujący „surowe CPU” oraz kompilację

Jeśli nie LTP, to: HINT

- HINT (http://hint.byu.edu/)
- (Hierarchical INTegration) – Benchmark, używający świeżego podejścia QUIPS (QUality Improvement Per Second).
- HINT odzwierciedla zmiany prędkości działanie maszyny w miarę wzrastania uptime'u.
- Bardzo łatwo skalowalny i dostępny na wiele architektur

Jeśli nie LTP, to: HINT



GCOV i pokrycie kodu jądra testami

- Czym jest COV?
  - Sprawdzenie stopnia przetestowania kodu
  - White box testing
  - Jedna z pierwszych technik
- Kryteria pokrycia:
  - Pokrycie funkcji
  - Pokrycie instrukcji (statement coverage)
  - Pokrycie decyzji/rozgałęzień (decision/branch coverage)
  - Pokrycie warunków (condition coverage)
  - Pokrycie ścieżek (path coverage)
  - Pokrycie wejścia/wyjścia (entry/exit coverage)

GCOV i pokrycie kodu jądra testami

- Istnieją zależności między kryteriami
- Niektóre aplikacje potrzebują 100% pokrycia
  - Niemniej jednak, na ogół jest to niemożliwe

- Problemy z pełnym pokryciem:
  - Duża ilość ścieżek
  - Nieskończona ilość ścieżek
  - Ścieżki niewykonywalne przy żadnym wejściu – problem stopu
- Techniki radzenia sobie z tym:
  - Grupowanie ścieżek w klasy

GCOV i pokrycie kodu jądra testami

- Pokrycie testami kodu jądra – czemu tak potrzebne?
  - Intensywne użycie zasobów nie zawsze implikuje intensywne użycie kodu jądra.
- Użyteczne narzędzia:
  - gcov
  - lcov
- Jak używamy?
  - Sekwencyjnie!

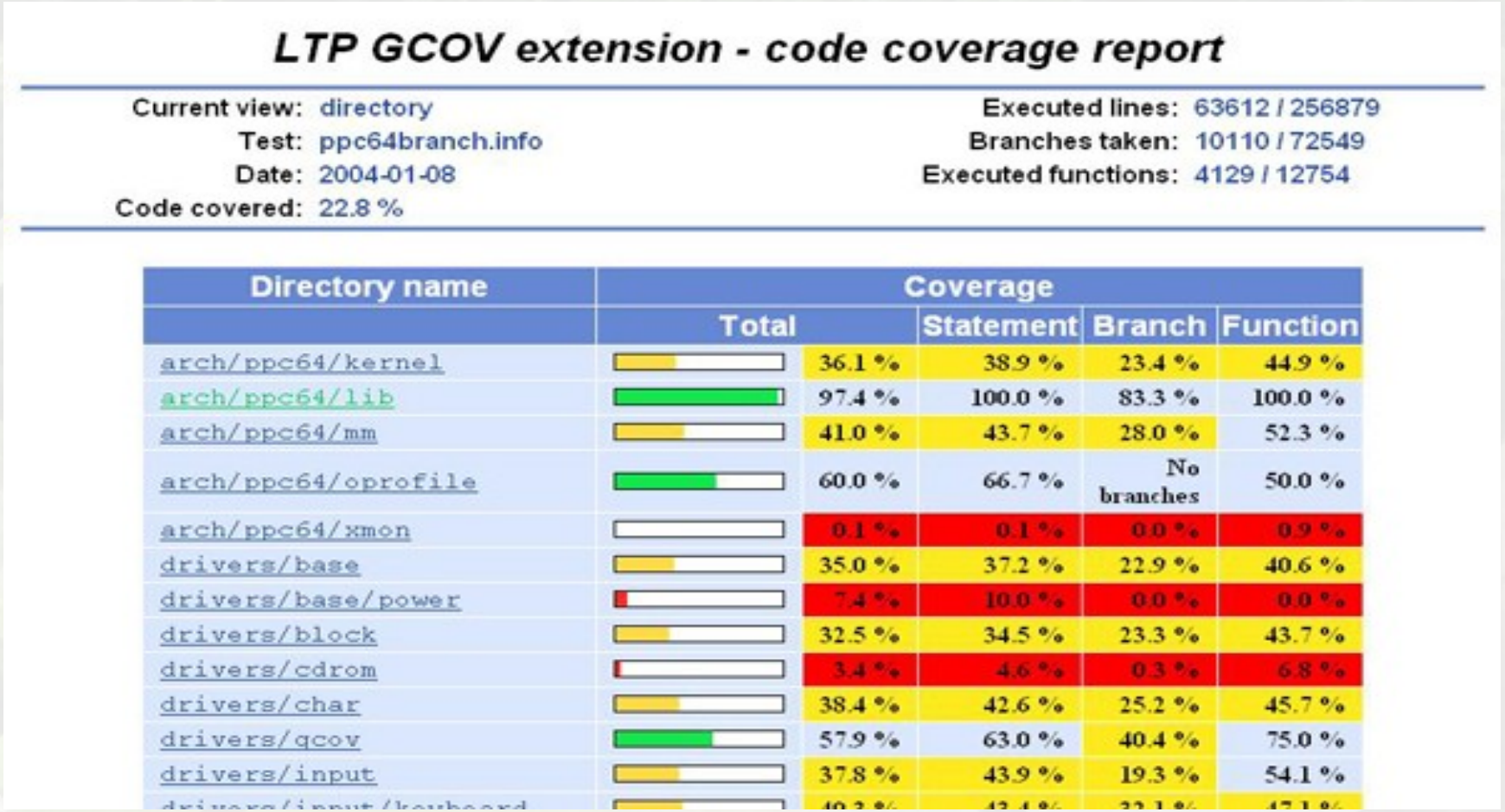
GCOV i pokrycie kodu jądra testami

- gcov – przykładowe wyjście

```
            -:    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:}
```

GCOV i pokrycie kodu jądra testami

- lcov – przykładowe wyjście



LTP GCOV extension - code coverage report

Current view: directory
Test: ppc64branch.info
Date: 2004-01-08
Code covered: 22.8 %

Executed lines: 63612 / 256879
Branches taken: 10110 / 72549
Executed functions: 4129 / 12754

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 %

GCOV i pokrycie kodu jądra testami

- lcov – przykładowe wyjście (szczegółowe)

```
                :
                : 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");
                :                 }
                :         }
                : }
```

Tracing

- Czym jest tracing?
- 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

- Najważniejsze cechy narzędzia do trace'u:
  - Zajrzenie w głąb działającego systemu
  - Wyszukiwanie wąskich gardeł systemu
  - Śledzenie wykonań procesów i środowiska, w którym się wykonują
  - Tryb flight-recorder (coś jak czarna skrzynka samolotu)
  - Możliwość zaglądania w przestrzeń użytkownika i jądra
- Przykładowe narzędzia:
  - SystemTap
  - Dtrace for Linux
  - Kprobes
  - LTTng i LLTV

Tracing

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

SystemTap - przykład

```
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)
```

Przykładowe testy jądra - Bootchart

- Bootchart – narzędzie do analizy procesu ładowania systemu operacyjnego Linux
- Powstał jako odpowiedź na wyzwanie postawione przez Owena Taylora:
  - *„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.”*

Przykładowe testy jądra - Bootchart

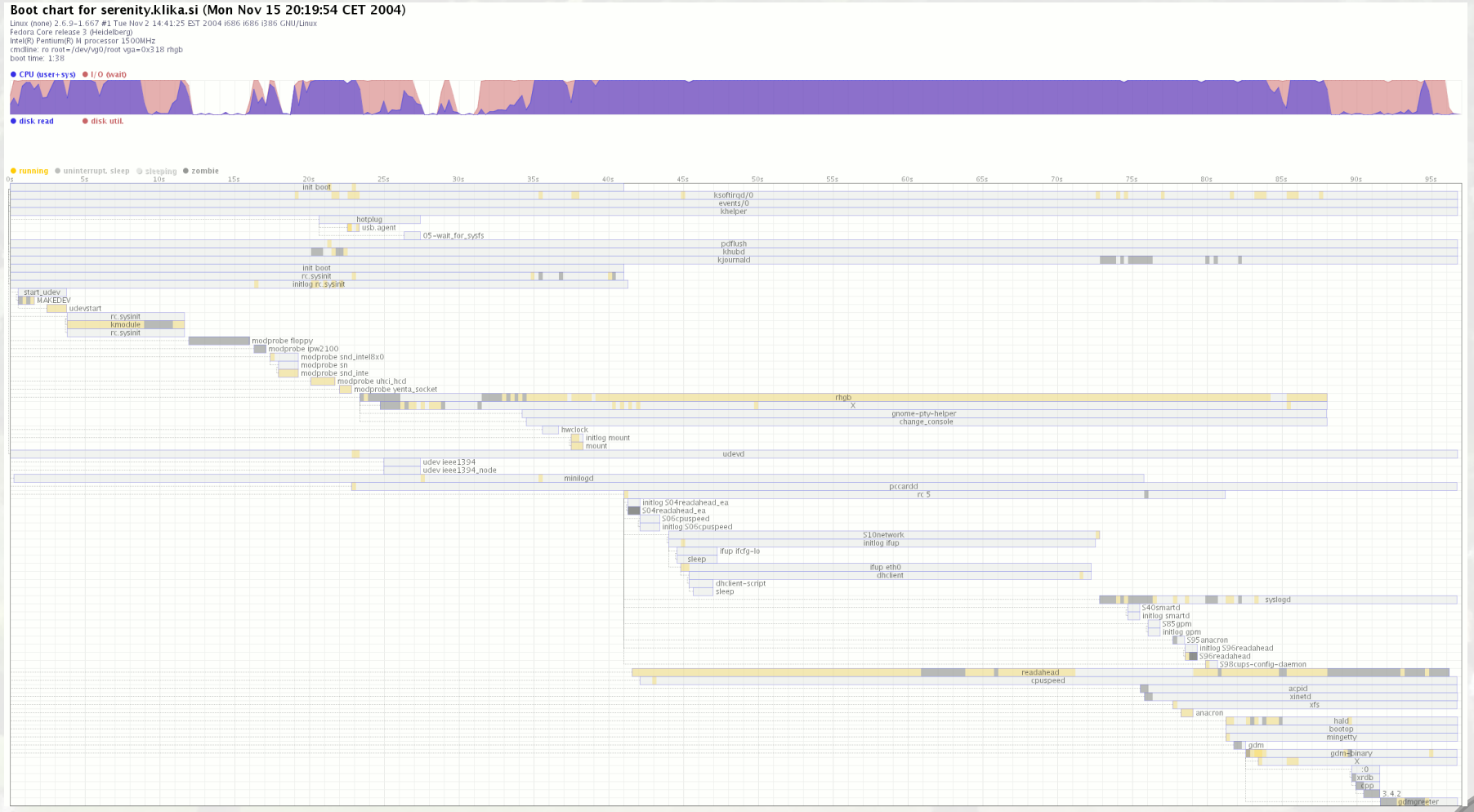
- Jak to działa?
  - Uruchamia /sbin/bootchartrd zamiast /sbin/init
  - Bootchart od razu uruchamia /sbin/init, ale sam zaczyna zbierać statystyki do tymczasowego systemu plików (tmpfs)
  - Skąd dane?
    - /proc/stat
    - /proc/diskstats
    - /proc/[PID]/stat
  - Warunek zakończenia działania – pojawienie się jednego z wyróżnionych procesów
  - Domyślne zbieranie statystyk – co 0.2 sekundy
  - Możliwość użycia process accounting

Przykładowe testy jądra - Bootchart

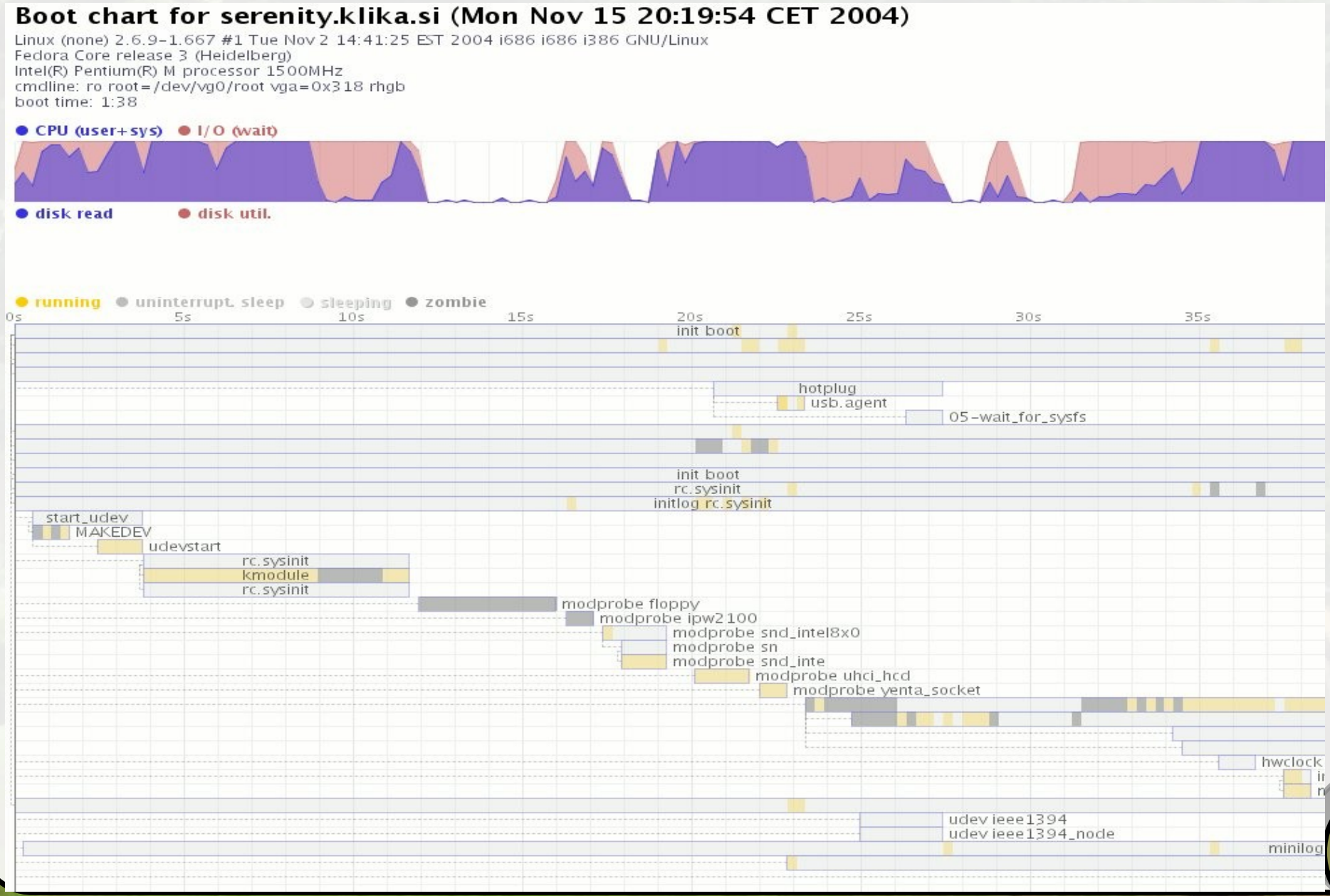
- Wizualizacja
  - Za pomocą aplikacji java lub formularza na stronie internetowej
  - Konieczność uproszczenia wykresu:
    - Łączenie “podobnych” procesów w jedną grupę procesów
    - Usuwanie krótko żyjących i bezczynnych procesów
- Graficzny sposób przedstawiania wyników (PNG, SVG, EPS)

Przykładowe testy jądra - Bootchart

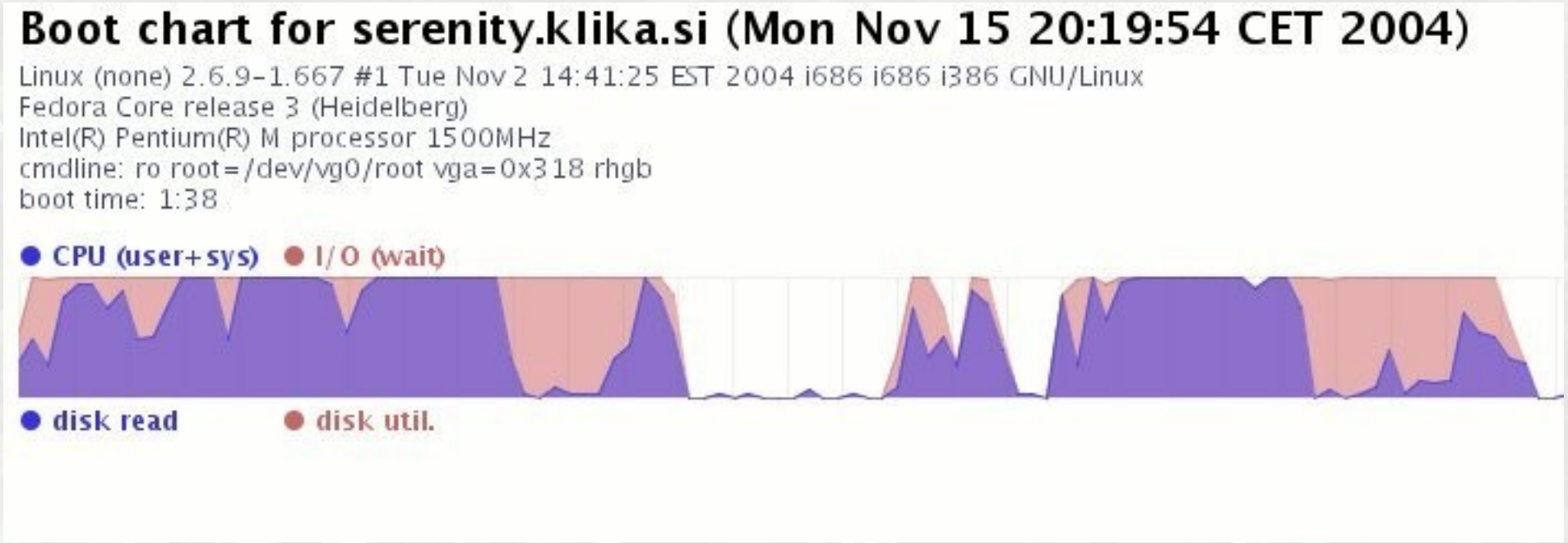
- Przykład – Fedora Core 3



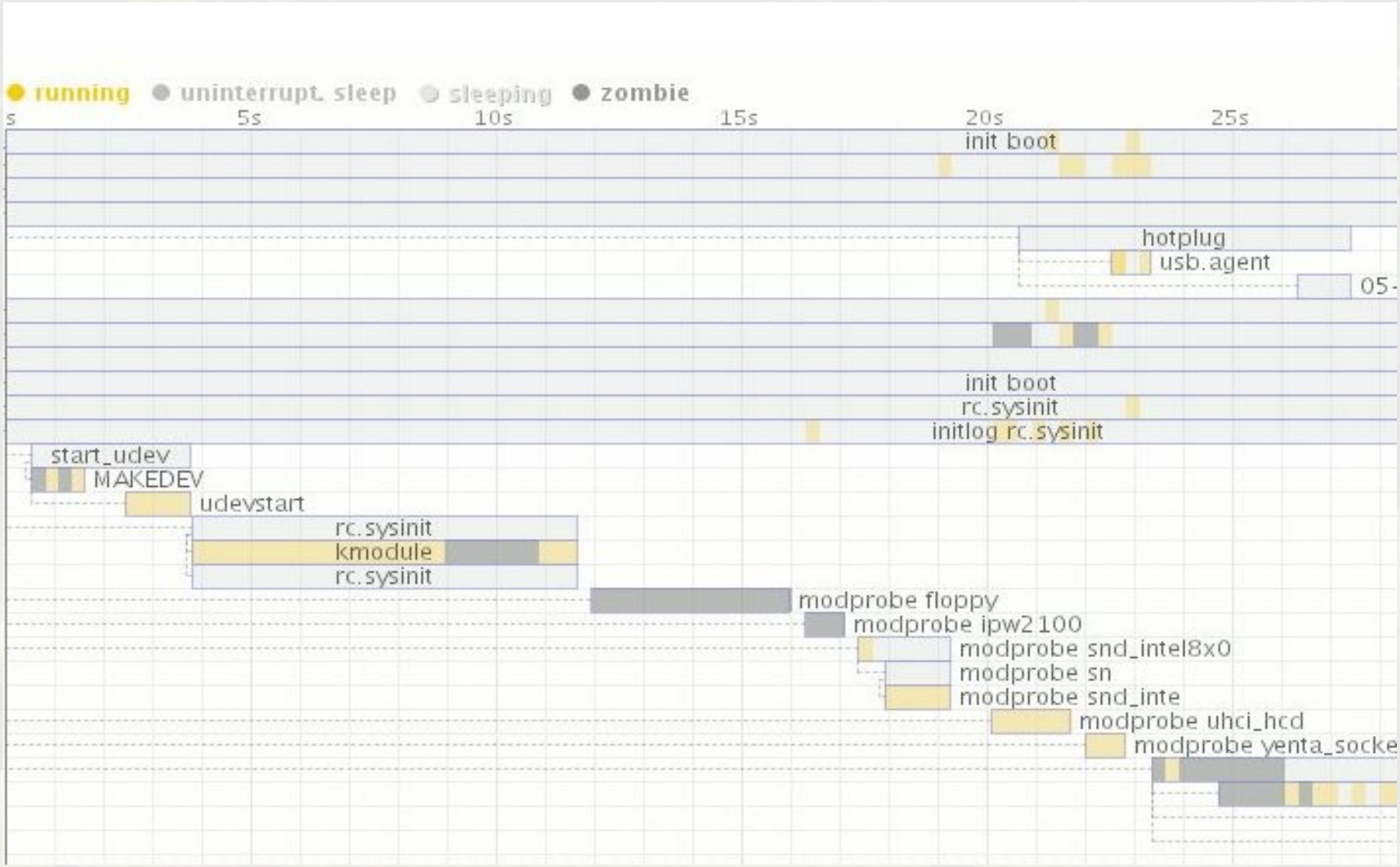
Przykładowe testy jądra - Bootchart



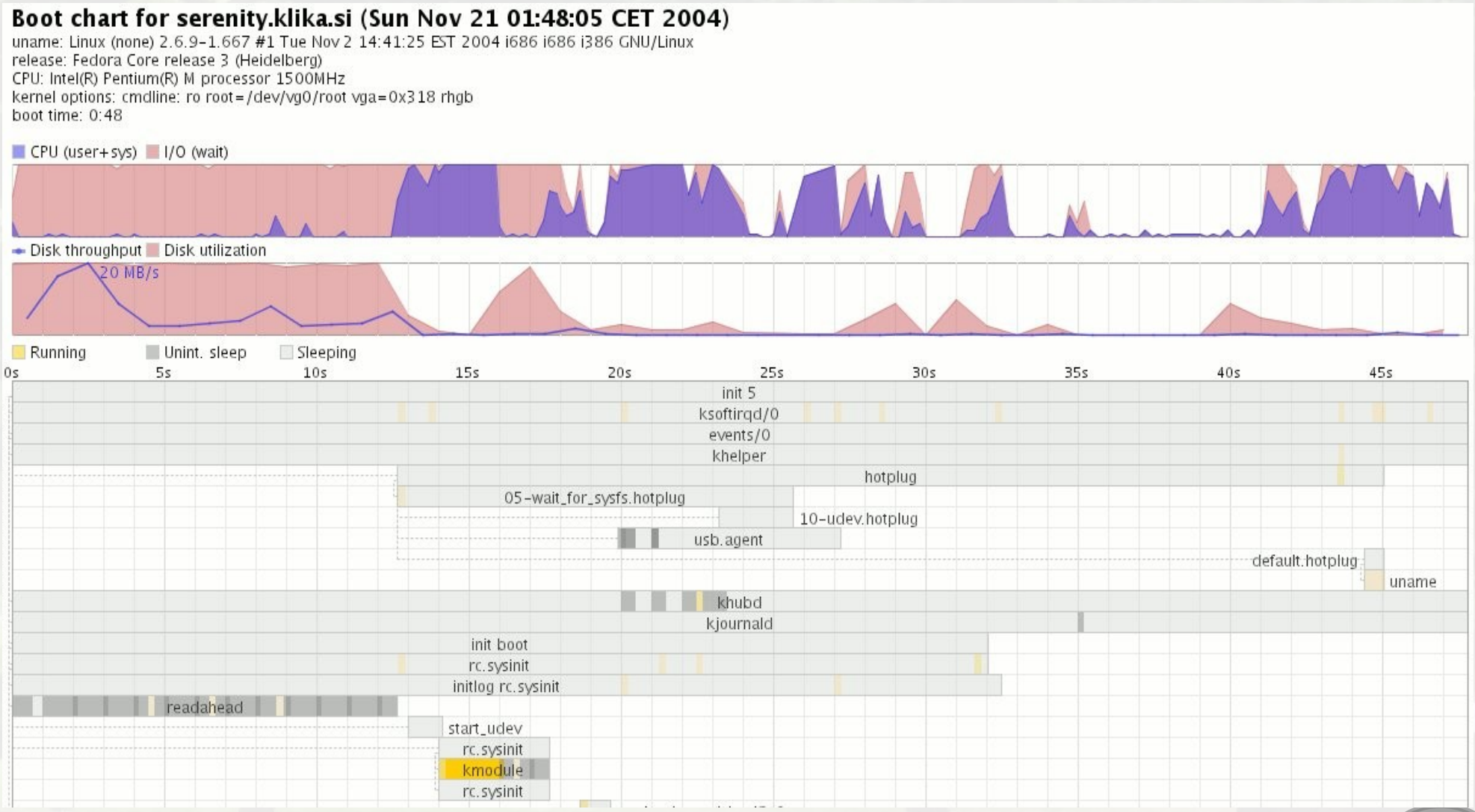
Przykładowe testy jądra - Bootchart



Przykładowe testy jądra - Bootchart



Przykładowe testy jądra - Bootchart

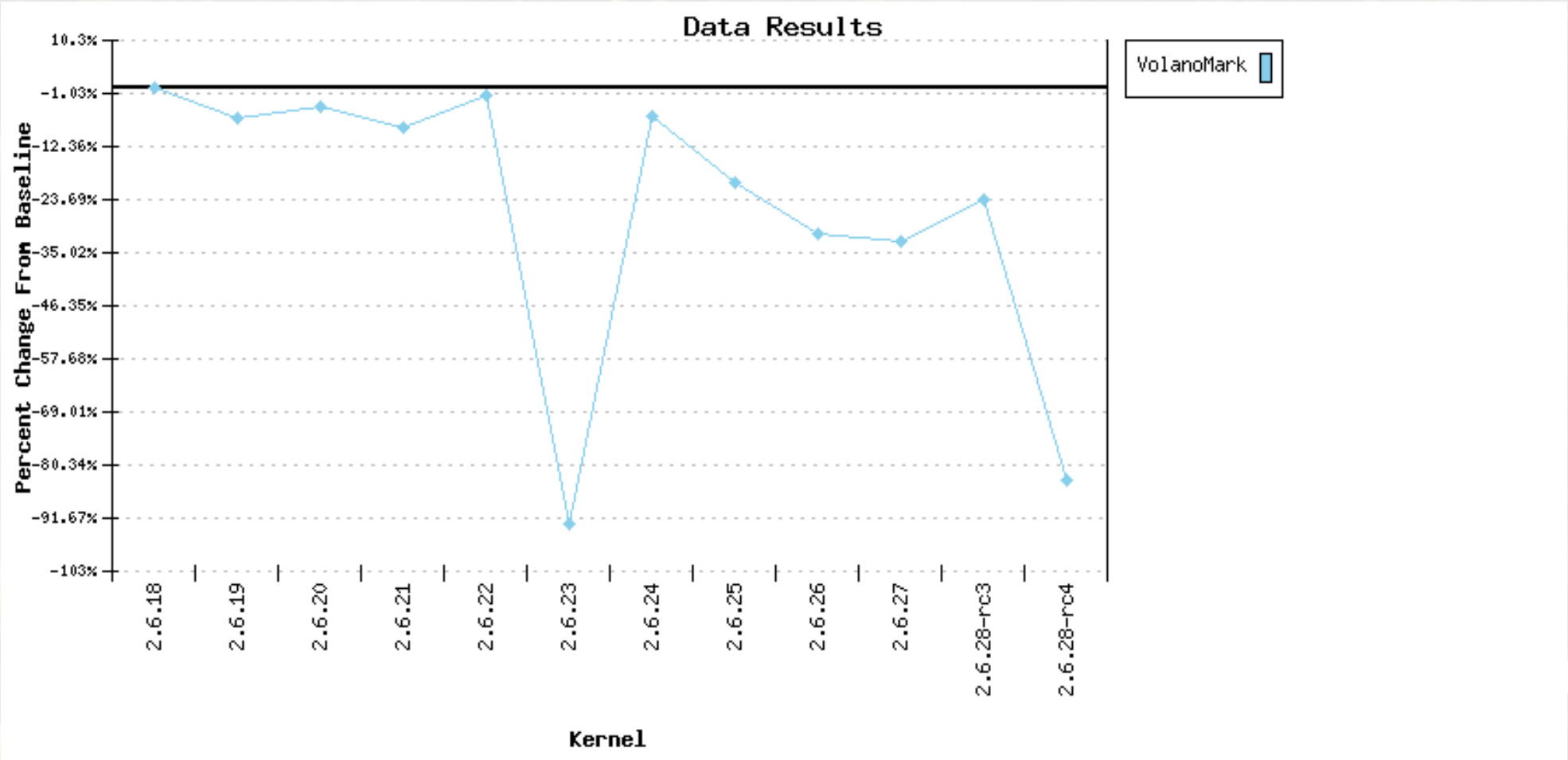


Przykładowe testy jądra - LKP

- Linux Kernel Performance Project – projekt mający na celu zwiększenie wydajności systemów Linux
- Częste testy wydajnościowe na tym samym sprzęcie różnych wersji jądra Linuxa
- Wydawałoby się, że każda kolejna wersja jądra powinna być lepsza od poprzedniej :)
- Spójrzmy zatem na przykładowe wyniki...

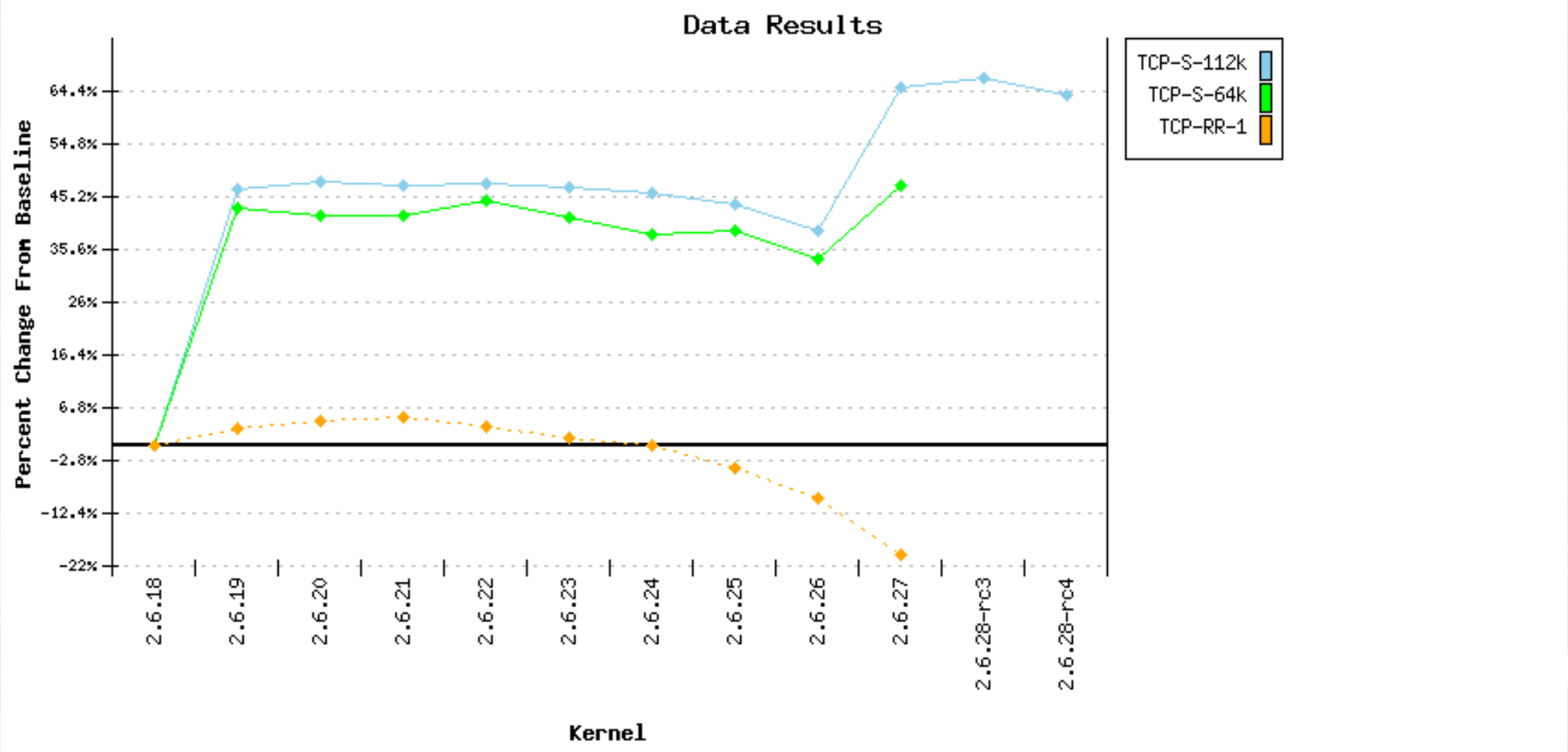
Przykładowe testy jądra

- VolanoMark



Przykładowe testy jądra

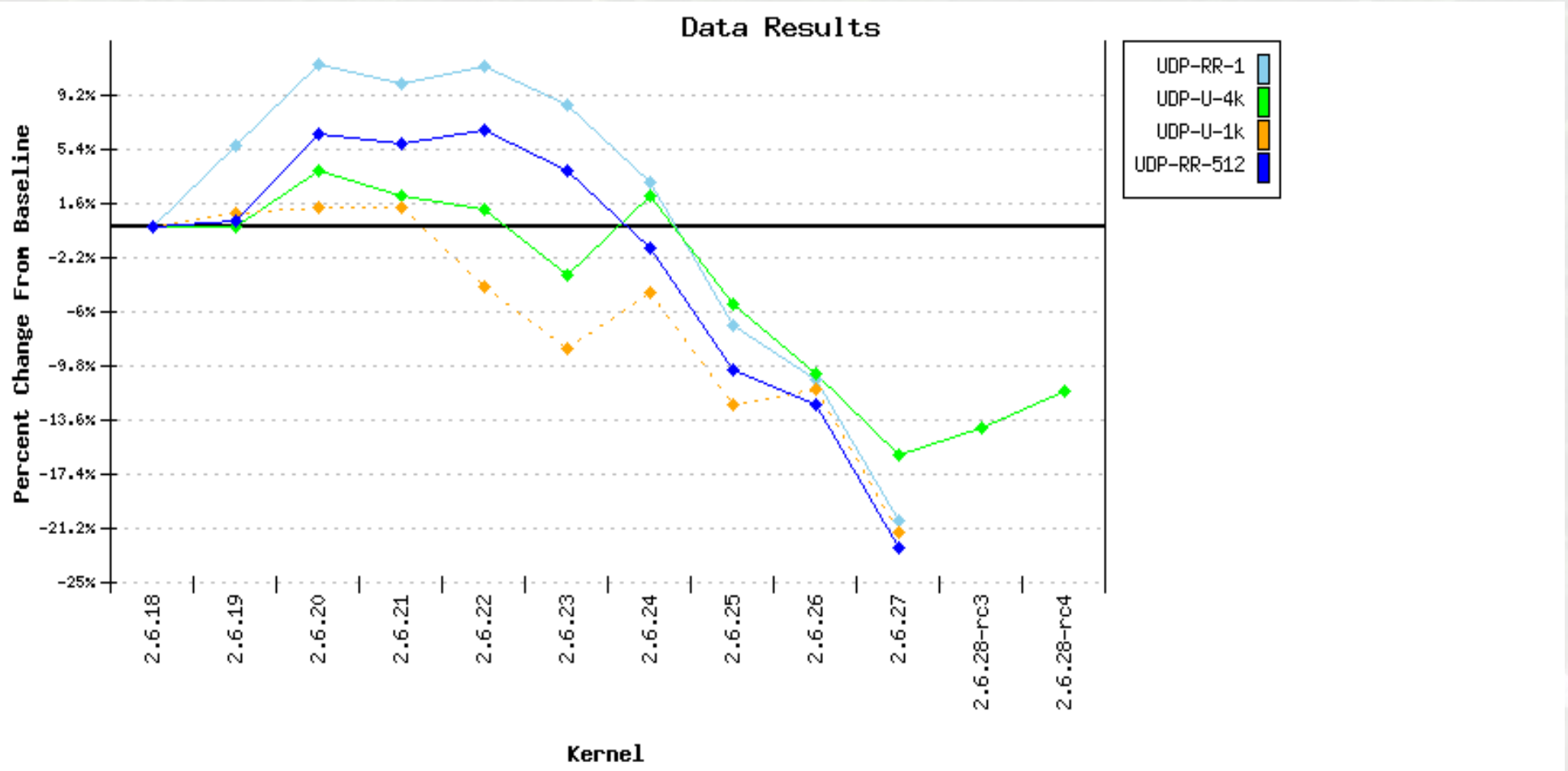
- Netperf – tcp



TCP-S-112k – test strumienia TCP z 112KB socket buffers i wiadmościami 4KB
TCP-S-64k – test strumienia TCP z 64KB socket buffers i wiadmościami 4KB
TCP-RR-1 – test TCP Request/Response z 1 byte'owymi zapytaniami i odpowiedziami

Przykładowe testy jądra

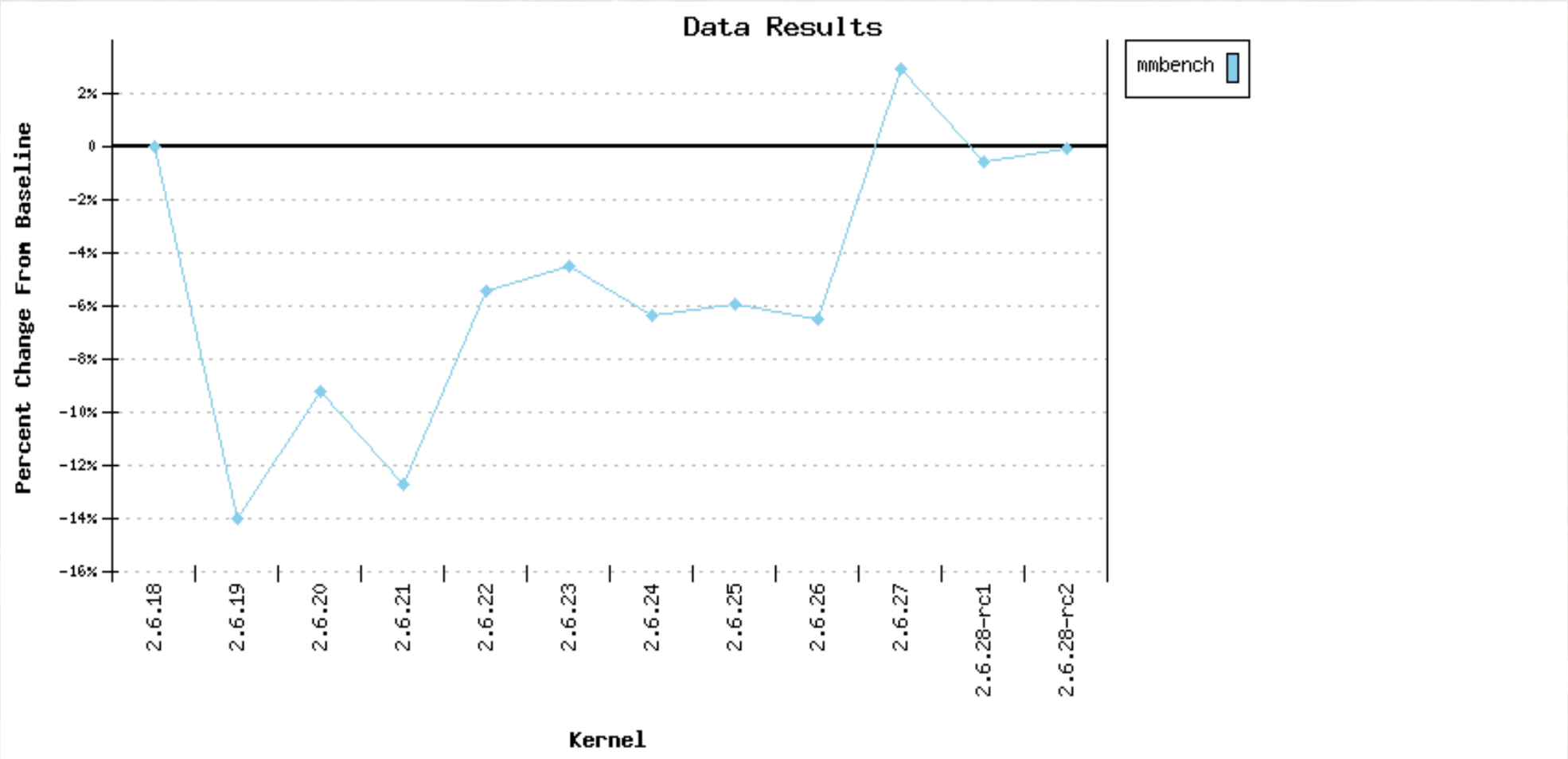
- Netperf – udp



UDP-RR-1 – test UDP Request/Response z 1 byte'owymi zapytaniami i odpowiedziami
UDP-RR-516 – test UDP Request/Response z 516 byte'owymi zapytaniami I 4 byte'owymi odpowiedziami
UDP-U-4k – test UDP Unidirectional z 64KB socket buffers i 4KB wiadomościami
UDP-U-1k – test UDP Unidirectional z 64KB socket buffers i 1KB wiadomościami

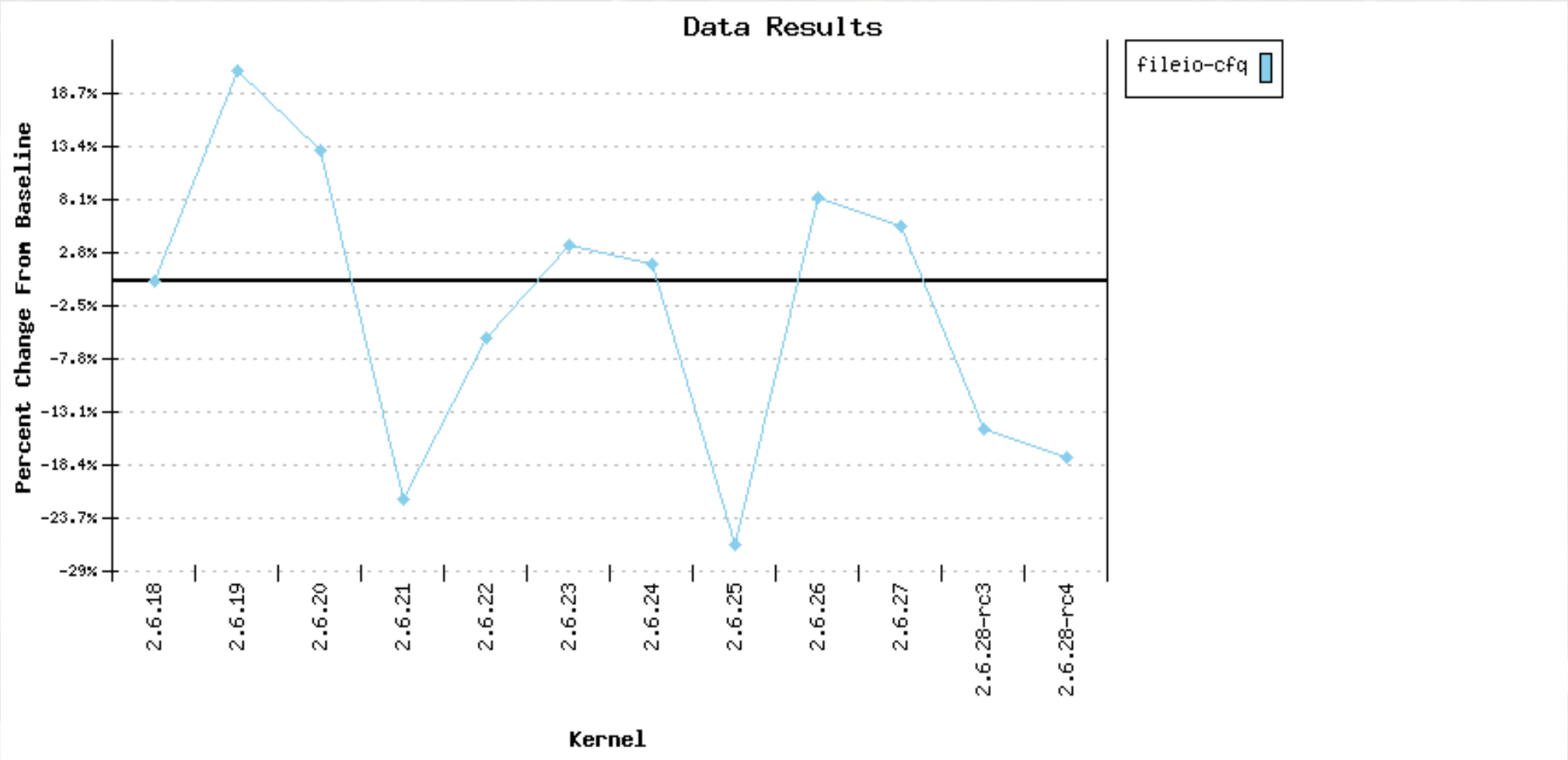
Przykładowe testy jądra

- Mmbench



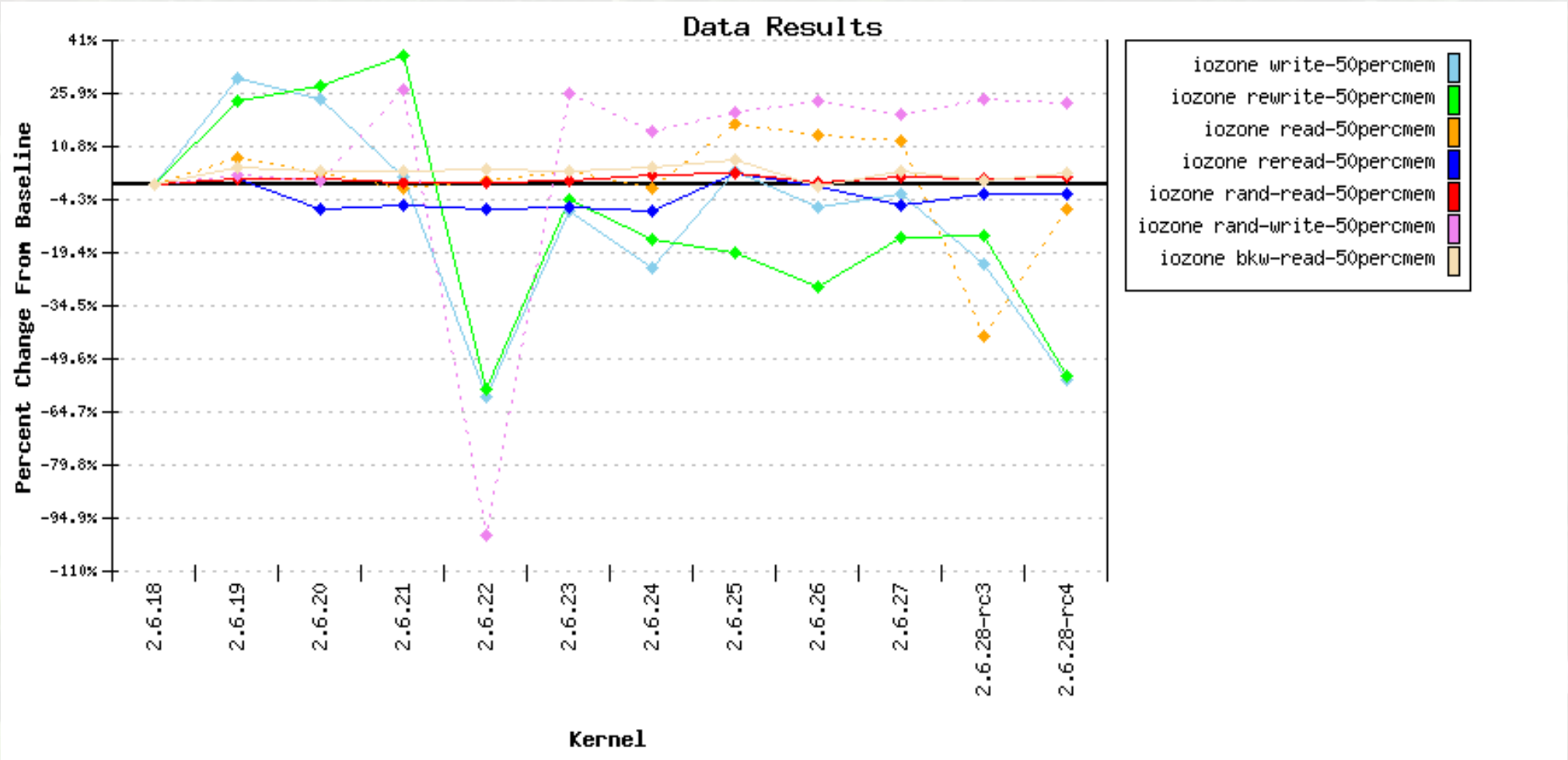
Przykładowe testy jądra

- Fileio



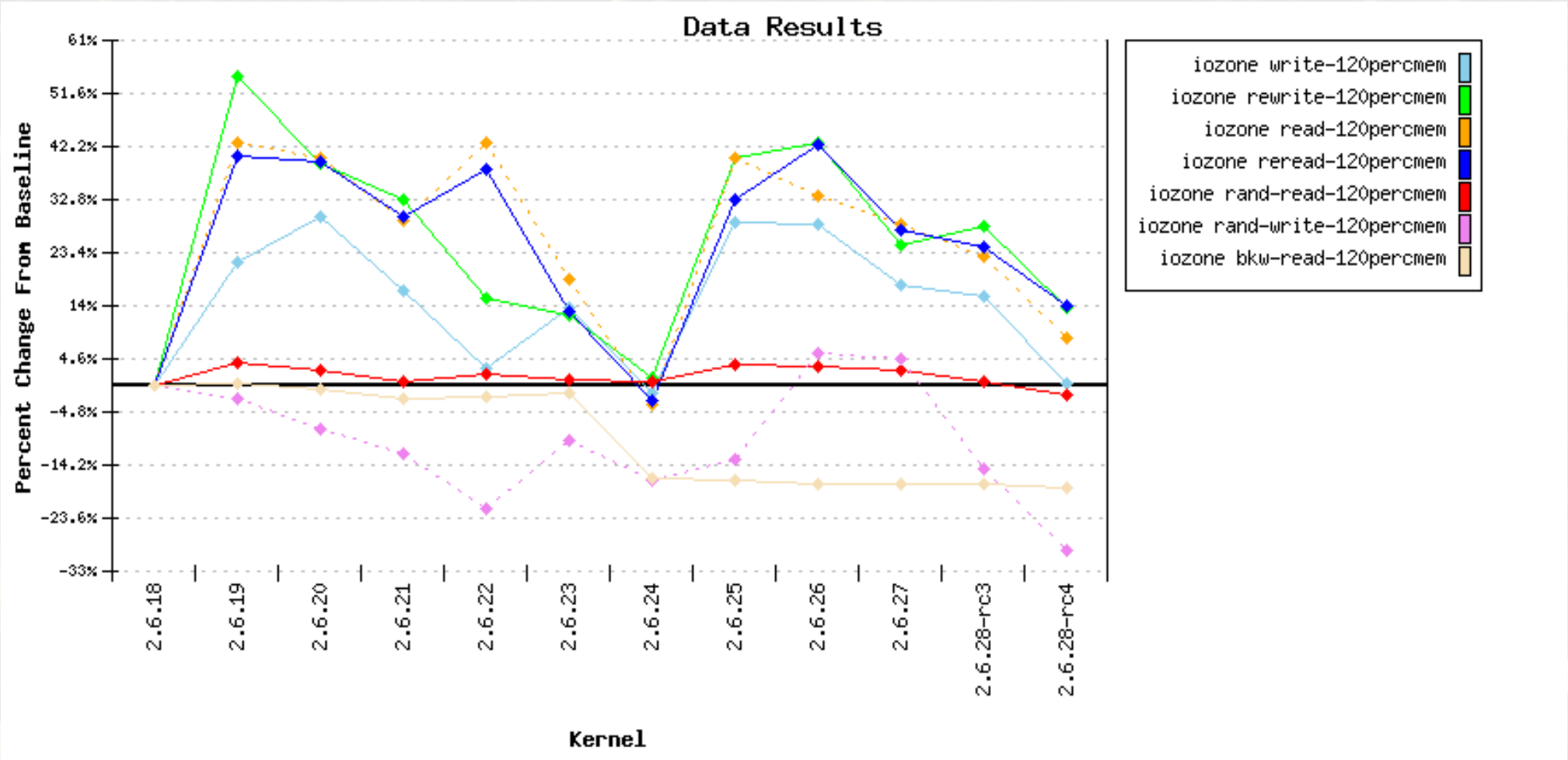
Przykładowe testy jądra

- Iozone – 50% obciążenie pamięci



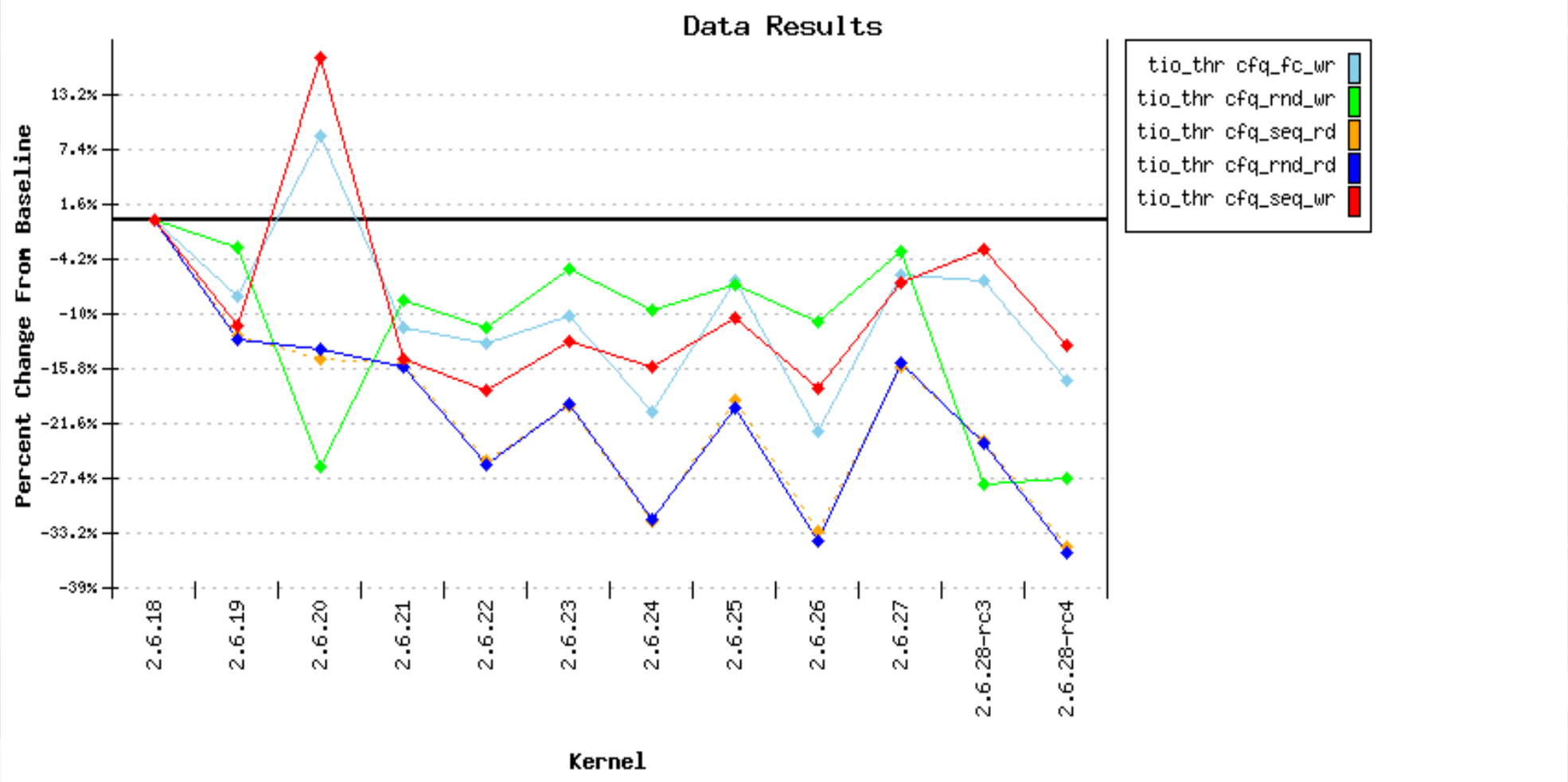
Przykładowe testy jądra

- Iozone – 120% obciążenie pamięci



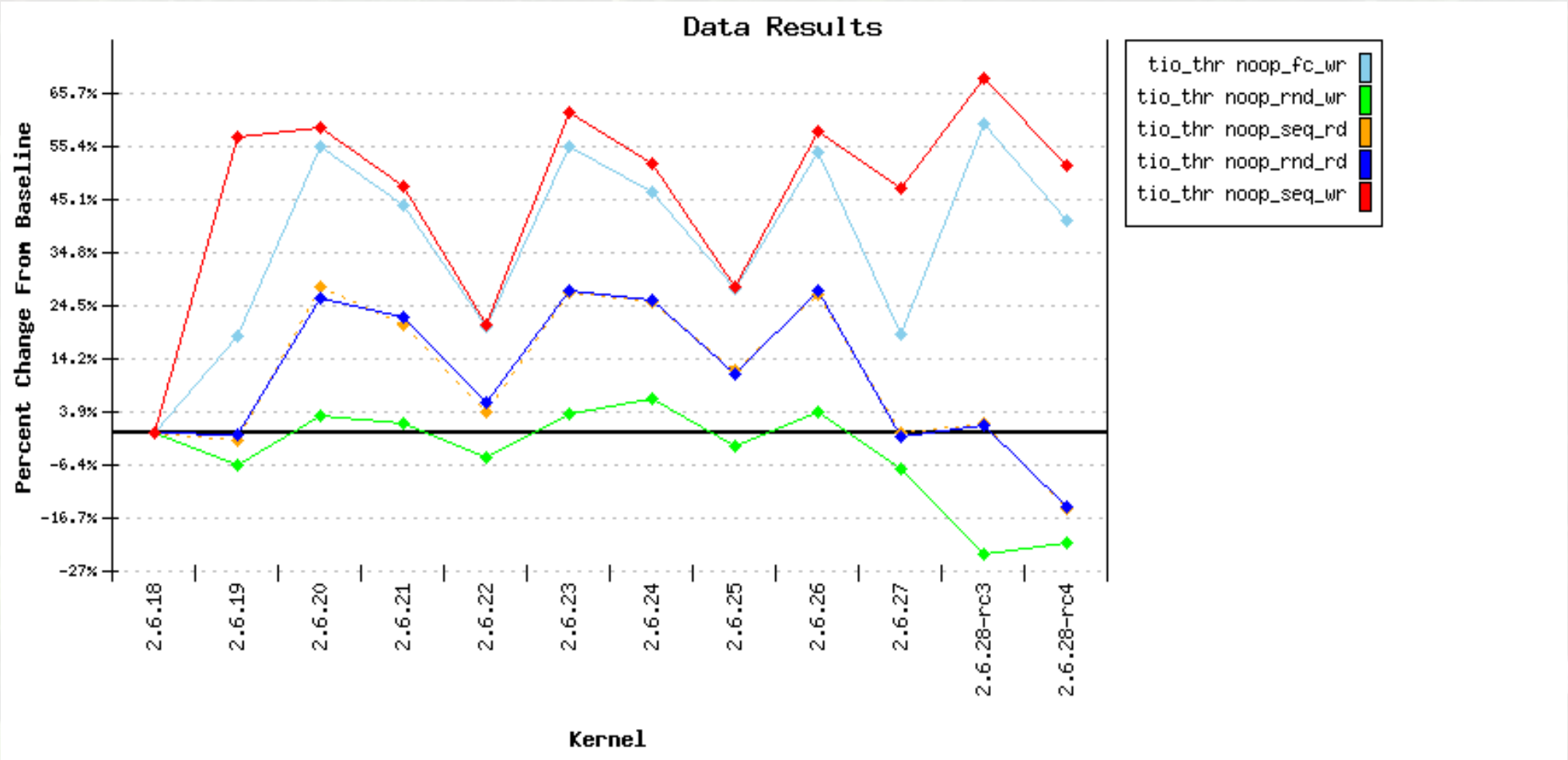
Przykładowe testy jądra

- Tiobench – różne operacje, algorytm cfq



Przykładowe testy jądra

- Tiobench – różne operacje, algorytm noop



Przykładowe testy jądra

- Tiobench – random read, różne algorytmy