출처 :
https://www.oss.kr/board_yaFS00/show/a48c8b4d-8e67-40ad-be88-73aadb2e2df4

perf 란 Linux 명령어 중의 하나이며 Linux Kernel 프로젝트에서 관리되는 성능측정 도구(profiler tool)이다. 리눅스 내의 특정 프로그램이나 시스템 전체를 분석할 수 있다. 예를 들면 내가 만든 프로그램의 어느 함수가 CPU 를 많이 사용하는지, 어느 코드 부분이 메모리 할당을 얼마나 하는지 등을 어셈블리 및 소스 코드 레벨로 파악할 수 있고 시스템 전체적으로도 page-fault, context-switch, cache-misses 등이 몇 번이나 일어났는지를 파악할 수 있다. 또한, 특정 커널 함수가 불리는지, 불린다면 얼마나 불리는지도 파악 할 수 있다.

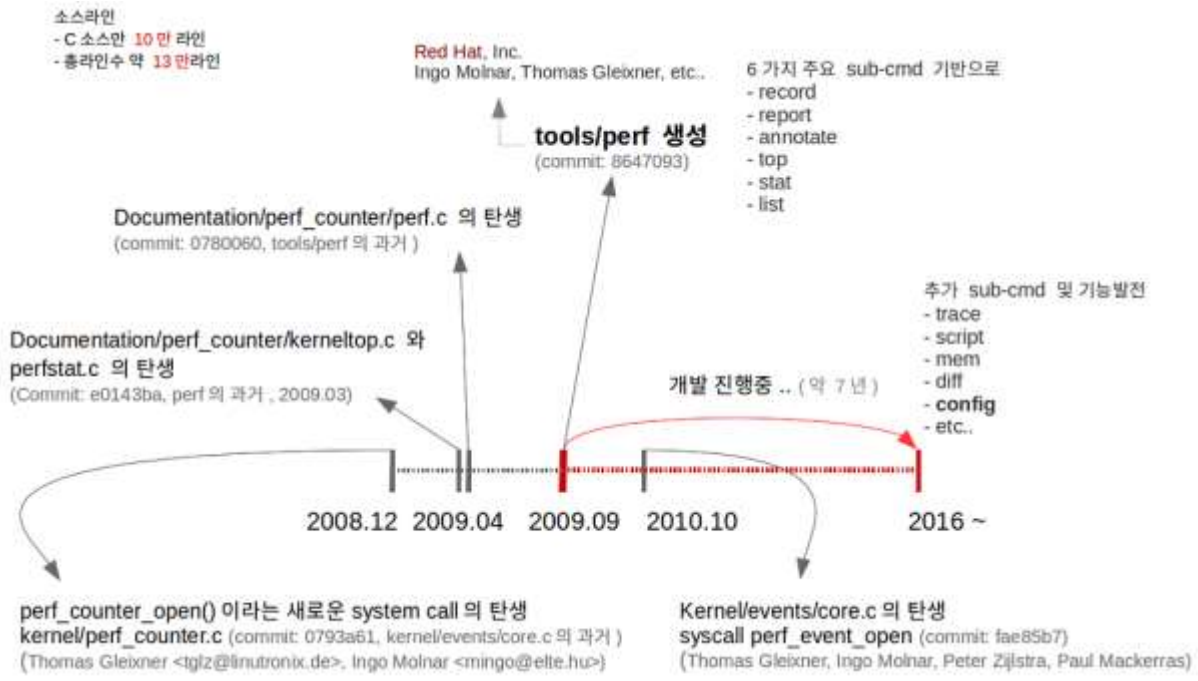
개요	Linux 기반 성능측정분석 도구
특징	- Linux kernel - performance events subsystem - Linux kernel 소스 내부 C 프로그램, Command line tool - 특정 프로그램 또는 시스템 전체 성능 분석 - 각종 CPU 에서 지원하는 PMU 기능 제어 - 각종 이벤트(cpu-cycles, cache-misses 등) 정보 수집 - 수집된 성능분석 정보 기반 통계 view 제공 (TUI, GUI 등) - Assembly 또는 source code 단위 overhead 분석 - Kernel API(무엇이 얼마나 불려지는지 등) Tracing 기능
목표	perf 프로젝트에 기여
기대효과	- 복잡, 다양해지는 각 커널 버전에 맞춰 성능분석 - 커널 또는 프로그램 속도 저하없이 성능 분석 - 각종 최신 CPU(x86, ARM, SPARC, etc.) 에도 이용 가능 - 커널, 시스템 과의 호환성 문제 및 성능저하 원인분석 용이
리퍼지토리	http://git.kernel.org/cgit/linux/kernel/git/tip/tip.git/ Branch : perf/core Source directroy : tools/perf, kernel/events 등

[목차]

1. Linux Kernel – perf 프로젝트 Time Line

Linux Kernel 은 수많은 sub project 들로 나뉘어서 관리된다. 커널 소스의 MAINTAINERS 파일을 열어보면 확인 가능한 것처럼 perf 프로젝트는 Performance Events Subsystem 이라는 이름으로 되어 있으며 공식적으로 https://git.kernel.org/pub/scm/linux/kernel/git/tip/tip.git 저장소에서 관리가 되고, perf/core 브랜치가 개발의 주가 되는 브랜치라고 볼 수 있다. 총 C 소스 라인은 10 만 라인 정도 되고, 모든 소스를 합하면 약 13 만 라인가량 된다.

아래 그림은 perf 프로젝트가 처음 만들어질 때부터 간략하게 역사 그래프를 만들어본 것이다. 본격적으로 perf 라는 이름으로 시작된 것은 2009 년이고 약 7 년 이상 개발되어오고 있다. 기간이 오래되긴 했으나 상당히 활발하게 개발이 진행되고 있으며 1 주에 약 20~50 개 정도의 새로운 PATCH 가 LKML 을 통해서 제안되고 적용되고 있다. 여러 가지 빌드 이슈나 기능문제, 사용법 등은 다음 메일링 리스트 linux-perf-users@vger.kernel.org 를 활용하면 도움받을 수 있다.



[그림 1] perf 프로젝트의 역사

2. Perf 사용 사례

perf 는 여러 가지 profiling/tracing 목적으로 이용할 수 있다. 다음 사례는 git 이라는 프로그램이 2009 년에 겪은 성능 이슈를 perf 로 분석한 사례이다.

http://osdir.com/ml/git/2009-08/msg00147.html 링크로 확인할 수 있듯이, 2009 년 8 월경에 git 프로그램 내부에서 Mozila SHA1 encryption 관련 성능문제가 발생했었고 perf 를 통해서 문제 상황 분석 및 확인을 할 수 있었다.

```
"perf report --sort comm,dso,symbol" profiling shows the following for
'git fsck --full' on the kernel repo, using the Mozilla SHA1:

    47.69%              git  /home/torvalds/git/git     [.] moz_SHA1_Update
    22.98%              git  /lib64/libz.so.1.2.3       [.] inflate_fast
     7.32%              git  /lib64/libc-2.10.1.so      [.] __GI_memcpy
     4.66%              git  /lib64/libz.so.1.2.3       [.] inflate
     3.76%              git  /lib64/libz.so.1.2.3       [.] adler32
     2.86%              git  /lib64/libz.so.1.2.3       [.] inflate_table
     2.41%              git  /home/torvalds/git/git     [.] lookup_object
     1.31%              git  /lib64/libc-2.10.1.so      [.] _int_malloc
     0.84%              git  /home/torvalds/git/git     [.] patch_delta
     0.78%              git  [kernel]                   [k] hpet_next_event
```

[그림 2] perf 사용 사례

3. Events

3.1. Event 란?

Kernel/App 이 실행됨에 따라 하드웨어/소프트웨어적으로 발생하는 모든 action 을 Event 라고 통칭한다. perf 에서 가장 중요한 것이 Event 이다. Event 에 대한 전반적인 이해가 동반되지 않으면 perf 를 정확하게 이용할 수 없으니, Event 에 대해서 알아보자.

위에서 설명했듯이 event 들은 4 종류로 구분할 수 있다. 하드웨어적으로 PMU(Performance Monitoring Unit) 또는 PMC(Performance Monitoring Counters)로 수집 가능한 이벤트 외에도 커널에서 제공하는 소프트웨어, tracepoint (ftrace 활용), 사용자에 의해서 정의될 수 있는 Event probepoint 가 있다.

Event 종류	예시	비고
하드웨어(PMU)	cpu-cycles, cache-misses, cache-references, instructions 등	해당 cpu 지원필요
소프트웨어	page-fault, context-switch, cpu-migrations 등	
Tracepoint	syscalls, irq, block, net, block, ext4, scsi, sock 등	
Probepoint	사용자정의에 의한 Event	

* Linux 내에서 perf list 로 수집가능한 전체 event 를 확인 할 수 있다.

[그림 3] Event 의 종류와 예시

a. Profiling 목적

perf 에서 profiling 이란 결국 sampling 을 통해서 각각의 Event 정보(언제/어디서/얼마나 등)를 수집하고 그 데이터를 분석, 통계 내는 기능을 의미한다. 그러한 목적에 쓰일 수 있는 Event 는 다음과 같이 2 가지 종류로 구분된다. 예를 들면 내가 만든 프로그램의 어떤 함수가 cpu-cycles 기준으로 overhead 가 가장 높은지 확인하고 성능분석을 할 수 있다.

- Hardware Event

아래와 같이 perf list 명령에 인자를 주면 하드웨어 이벤트들을 확인할 수 있다. 하지만 모든 환경에서 다음과 같은 결과가 출력되진 않는다. CPU 의 종류가 어떤 것이냐에 따라 지원되는 HW 이벤트는 달라질 수 있다.

```
$ perf list hw cache pmu

List of pre-defined events (to be used in -e):

  branch-instructions OR branches                    [Hardware event]
  branch-misses                                      [Hardware event]
  bus-cycles                                         [Hardware event]
  cache-misses                                       [Hardware event]
  cache-references                                   [Hardware event]
  cpu-cycles OR cycles                               [Hardware event]
  instructions                                       [Hardware event]
  ref-cycles                                         [Hardware event]

  L1-dcache-load-misses                              [Hardware cache event]
  L1-dcache-loads                                    [Hardware cache event]
  L1-dcache-stores                                   [Hardware cache event]
...
```

[그림 4] Hardware Event

- Software Event

```
$ perf list sw

List of pre-defined events (to be used in -e):

  alignment-faults                                   [Software event]
  bpf-output                                         [Software event]
  context-switches OR cs                             [Software event]
  cpu-clock                                          [Software event]
  cpu-migrations OR migrations                       [Software event]
  dummy                                              [Software event]
  emulation-faults                                   [Software event]
  major-faults                                       [Software event]
  minor-faults                                       [Software event]
  page-faults OR faults                              [Software event]
  task-clock                                         [Software event]
```

[그림 5] Software Event

b. Tracing 목적

perf 에서 tracing 은 각 tracepoint 가 될 수 있는 Event 들의 정보를 수집하여 특정 Event 들의 발자취, 즉 Event 발생과정을 추적할 수 있다. 예를 들면 perf 의 call-graph view 를 통해서 Event 가 발생하기까지 누가 누구를 호출했는지 모든 과정을 보면서 실행과정은 어떠한지, 문제 상황이 있는지 등을 사용자가 추적할 수 있다. 이런 목적을 위해서 쓰이는 Event 는 2 가지 종류로 구분된다.

- Tracepoint Event

말 그대로 추적 가능한 이벤트이다. block 이라고 하면 Block Device 레벨에서 벌어지는 I/O 등을 각각 이벤트라 하고 그와 관련된 모든 과정(호출과정, 언제, 어떻게, 얼마나 등)을 추적할 수 있다.

```
$ perf list hw cache pmu

List of pre-defined events (to be used in -e):

  block:block_bio_backmerge                          [Tracepoint event]
  block:block_bio_bounce                             [Tracepoint event]
  block:block_bio_complete                           [Tracepoint event]
  block:block_bio_frontmerge                         [Tracepoint event]
...
  cfg80211:cfg80211_cac_event                        [Tracepoint event]
  cfg80211:cfg80211_ch_switch_notify                 [Tracepoint event]
  cfg80211:cfg80211_ch_switch_started_notify         [Tracepoint event]

...
```

[그림 6] Tracepoint Event

- Probepoint Event

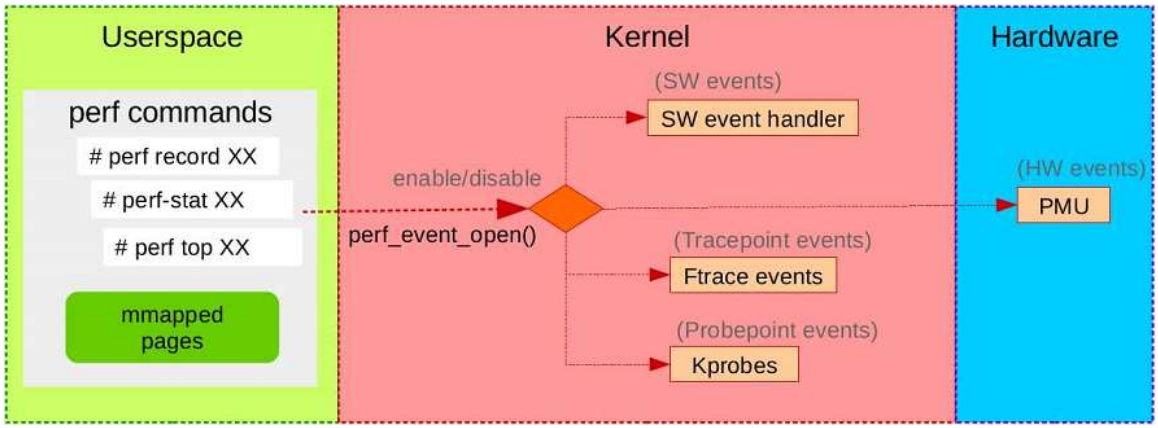
Probepoint 는 사용자 정의 Event 로 특정 함수명을 dynamic tracepoint 로 정의 내릴 수 있다. 다만 커널 debug info 를 가지고 있는 상태여야 한다. 만약 debug 정보가 없다면 커널을 빌드할 때 설정에서 debug info 를 포함하여 재빌드한 후 커널을 변경해서 부팅하여 perf probe 를 정상 이용할 수 있다. 아래 예는 tcp_sendmsg 라는 커널 함수를 사용자 정의 Event 로 선언하고 그것을 확인하는 과정이다. 다음과 같이 등록하여 이 함수의 argument 정보나 변수정보, 리턴 값은 무엇인지, 소스 몇 라인에서 특정 local 변수가 어떤 값을 가졌는지 등을 전부 동적 추적이 가능하다.

```
$ perf probe --add tcp_sendmsg
Added new event:
  probe:tcp_sendmsg    (on tcp_sendmsg)

You can now use it in all perf tools, such as:

    perf record -e probe:tcp_sendmsg -aR sleep 1

$ perf probe --list
  probe:tcp_sendmsg    (on tcp_sendmsg)
```

[그림 7] Probepoint Event

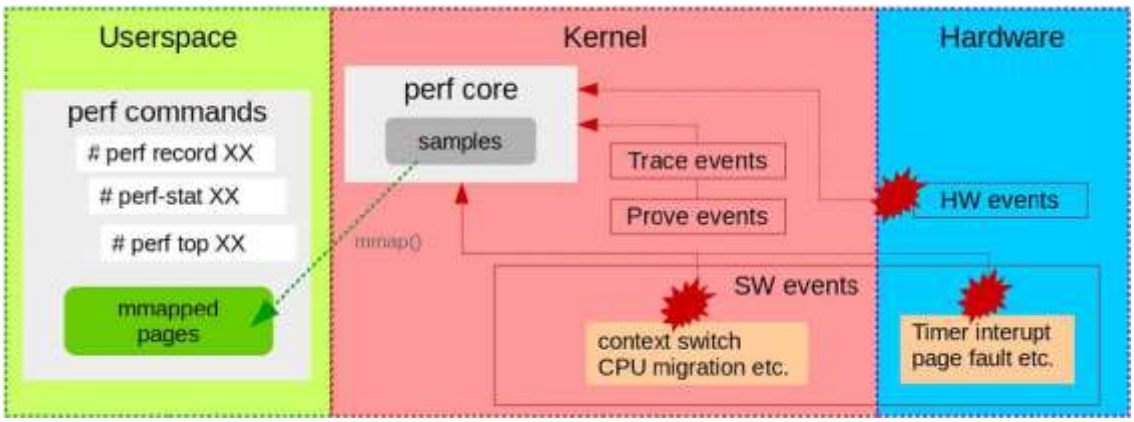
3.2. Event Sampling 과정

우선 perf_event_open 시스템 콜을 통해서 이벤트 샘플링이 가능하다. 이벤트 샘플링이라 하면 결국 해당 이벤트에 대한 정보(언제/어떻게/얼마나)를 수집한다는 뜻이다. 우선 perf_event_open 특정 이벤트 정보와 함께 시스템 콜을 하면 file descriptor 를 리턴 받게 된다. 성능정보를 측정할 수 있는 file descriptor 를 통해서 각각의 event 들과 연결할 수 있게 되는데 동시에 여러 이벤트들을 수집 가능하게 Grouping 도 가능하다. 그러면 각 event 들을 enable 하거나 disable 할 수 있다. 이는 ioctl() 나 prctl() 시스템 콜을 통해서 컨트롤이 가능하다. 이 과정을 아래 그림으로도 살펴볼 수 있다.



[그림 8] Event Sampling 과정 1

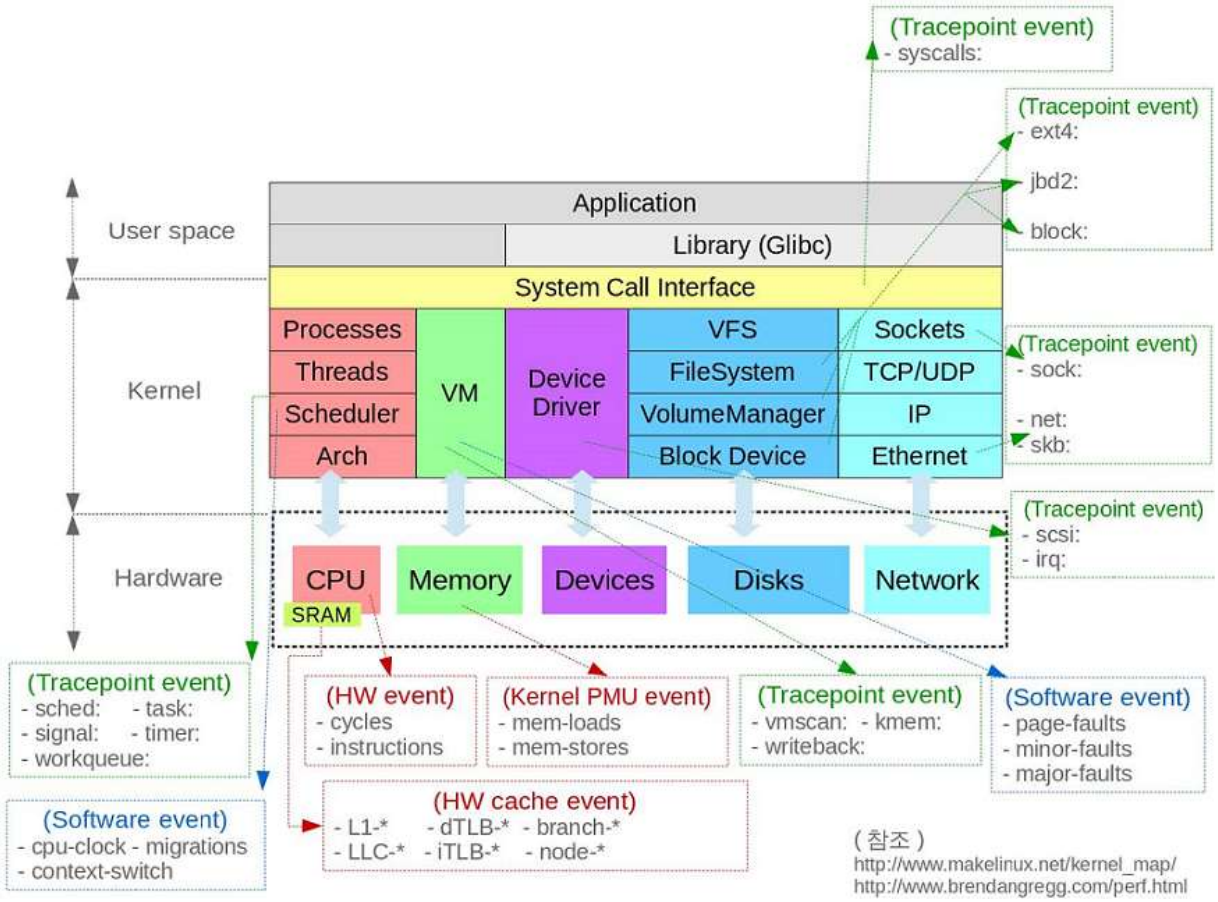
위의 과정이 끝나면 우선 특정 이벤트에 대해서 정보를 수집(sampling)하거나 단순히 이벤트 발생횟수를 세는 것(counting)이 가능해 진 것이다. sampling 을 한다고 하면 우선 mmap() 시스템 콜을 통해서 특정 메모리 영역(버퍼)에 정보를 받을 준비를 한 후에 특정 이벤트들이 발생할 때마다, 혹은 지정된 특정주기(ex. Page fault 10 번당 한번씩)마다 그와 관련된 정보들을 측정 정보를 버퍼에 쓴다. 버퍼에 쓰는 과정은 hw/sw 인터럽트(ex. PMU 활용)가 발생할 때마다 쓰게 된다. 그리고 perf command 가 mmap()를 통해서 성능정보 수집하기 때문에 kernel 에서 user 의 copying 없이 데이터 수집이 가능하다. 실제 수집준비가 끝난 뒤 수집하는 과정은 다음 그림으로도 확인할 수 있다.



[그림 9] Event Samping 과정 2

3.3. System Layer 기반에서 본 Event

아래 그림은 Linux kernel 의 주요 5 가지 subsystem 기준으로 system layer 를 나눠본 것이다. 아래와 같이 프로세스 관리(process management), 메모리 관리(memory management), 디바이스 드라이버(device driver), 파일 시스템(file system), 네트워킹(Networking) 5 가지로 나누어보았을 때 각각에 영역에 대한 이벤트는 다음 그림으로 살펴볼 수 있다.



[그림 10] Linux Kernel 의 주요 5 가지 subsystem 기준으로 나눈 system layer

4. Perf 기능소개

1. **4.1. 이벤트 발생횟수 세기(Counting)**

   특정 이벤트들의 발생 횟수가 몇 개인지 측정한다. 구체적으로 어떻게 카운팅을 할 수 있는지 예를 들어보겠다. pwd 라는 명령이 있다. 간단하게 perf 를 통해서 pwd 라는 프로그램이 대표적인 몇 개의 이벤트를 몇 번이나 발생시키는지를 알아보자. 'perf stat pwd'라고 명령어를 입력하면 pwd 라는 프로그램에 대해서 기본적인 이벤트들을 몇 번이나 발생시키는지 아래 표와 같이 확인할 수 있다.

```
$ perf stat pwd
/home/taeung

 Performance counter stats for 'pwd':

          0.585632      task-clock (msec)         #    0.729 CPUs utilized          (67.72%)
                 1      context-switches          #    0.002 M/sec                  (67.72%)
                 0      cpu-migrations            #    0.000 K/sec                  (67.72%)
                93      page-faults               #    0.159 M/sec                  (67.72%)
         1,644,012      cycles                    #    2.807 GHz                    (68.85%)
         1,079,596      instructions              #    0.66  insn per cycle         (68.85%)
           210,875      branches                  #  360.081 M/sec                  (68.85%)
             9,617      branch-misses             #    4.56% of all branches        (68.85%)

       0.000802816 seconds time elapsed
```

[그림 11] 기본 이벤트 Counting

하지만 위의 사용은 사용자가 원하는 특정 이벤트를 지정한 것은 아니다. 만약에 pwd 라는 프로그램에 대해서 CPU 사이클이 얼마나 도는지에 대해서 알고 싶다면 다음과 같이 측정할 수 있다.

```
$ perf stat -e cycles pwd
/home/taeung

 Performance counter stats for 'pwd':

         1,157,617      cycles                                                (73.30%)

       0.000495629 seconds time elapsed
```

[그림 12] 특정 이벤트 Counting

또한, 특정 이벤트들의 발생횟수를 살펴보고 싶다면 Modifier 를 이용할 수 있다. (중복사용도 가능하다)

a. 이벤트 수식어옵션(Modifiers)

Modifiers	설명	형식 및 예시
u	유저레벨(priv level 3,2,1)	event 명:u (ex. cycles:u)
k	커널레벨(priv level 0)	event 명:k (ex. cycles:k)
h	가상환경의 하이퍼바이저레벨 이벤트	event 명:h (ex. cycles:h)
H	가상환경의 Host 머신 대상	event 명:H (ex. cycles:H)
G	가상환경의 Guest 머신 대상	event 명:G (ex. cycles:G)

[그림 13] Modifiers

1. **4.2. 성능분석하기(Profiling)**

Hardware 또는 Software 이벤트 정보수집(sampling)을 통해서 분석하고, 통계 view 를 통해서 성능을 분석할 수 있다. Knapsack problem 알고리즘 풀이 프로그램이 있다. 본 프로그램의 성능에 대해서 perf 를 통해서 분석해보자. 우선 본 프로그램을 테스트 스크립트와 함께 실행시키면서 이벤트 관련 정보를 수집한다. 6623856 출력 값은 이 프로그램의 실행 결과이다.

```
$ perf record ./test.sh old_pack_knapsack
6623856
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.024 MB perf.data (238 samples) ]
```

[그림 14] Profiling 과정 1

특정 이벤트를 지정하지 않으면 기본적으로 cycles 이벤트를 기준으로 측정하게 된다.
측정이 끝나면 perf.data 라는 파일이 생성되게 되는데 이 파일이 현재 폴더에 존재하면 그 파일을 기준으로 perf report 기능이 다음과 같이 동작할 수 있다.

```
$ perf report --stdio
# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 238  of event 'cycles:ppp'
# Event count (approx.): 161333594
#
# Overhead  Command          Shared Object      Symbol
# ........  ...............  .................  ........................................
#
    91.54%  old_pack_knapsa  old_pack_knapsack  [.] pack_knapsack
     1.17%  test.sh          [kernel.kallsyms]  [k] cap_capable
     1.13%  test.sh          libc-2.21.so       [.] _IO_putc
     1.06%  test.sh          [kernel.kallsyms]  [k] __radix_tree_lookup
     0.98%  test.sh          libc-2.21.so       [.] __gconv_transform_utf8_internal
     0.97%  test.sh          [kernel.kallsyms]  [k] perf_event_aux_ctx
     0.79%  old_pack_knapsa  [kernel.kallsyms]  [k] sd_init_command
     0.79%  test.sh          [kernel.kallsyms]  [k] int_ret_from_sys_call
     0.64%  test.sh          [kernel.kallsyms]  [k] __wake_up_bit
     0.60%  test.sh          [kernel.kallsyms]  [k] handle_mm_fault
     0.08%  old_pack_knapsa  [kernel.kallsyms]  [k] handle_mm_fault
     0.07%  test.sh          [kernel.kallsyms]  [k] _raw_spin_lock
...
```

[그림 15] Profiling 과정 2

4.3. 이벤트 발생과정 추적하기(Tracing)

Tracepoint/Probepoint 이벤트에 대한 다양한 정보(누가/언제/어디서/얼마나/어떻게 등)를 수집하여 이벤트의 근원지부터 결과까지의 전 과정을 추적할 수 있다.

a. 정적 추적(Static Tracing)

정적 추적이라 하면 위의 성능분석(Profiling)과 유사한 명령을 통해서 이용할 수 있지만 목적이 다르다. 성능을 분석하는 부분은 Overhead 를 보면서 성능의 언밸런스를 찾는 게 초점이라면 Tracing 은 특정 이벤트가 발생의 근원부터 언제, 어떻게, 어떤 과정을 통해서 이벤트가 발생했는지 전 과정을 살펴볼 수 있고 그것을 통해서 실행과정에 문제가 있는지 혹은 어떠한 과정을 토대로 문제가 생겨나는지를 추적할 수 있다.

```
$ perf record -e sched:sched_process_exec -a
$ ./test.sh old_pack_knapsack (벤개로 실행)
$ perf report --stdio
# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 2  of event 'sched:sched_process_exec'
# Event count (approx.): 2
#
# Overhead  Trace output
# ........  ........................................................
#
    50.00%  filename=./old_pack_knapsack pid=15891 old_pid=15891
    50.00%  filename=./test.sh pid=15889 old_pid=15889


#
# (Tip: Profiling branch (mis)predictions with: perf record -b / perf report)
#
```

[그림 16] Static Tracing

b. 동적 추적(Dynamic Tracing)
정적 추적(Static Tracing)은 우선 event sampling 을 한 후에 수집이 완료된 perf.data 를 통해서 분석 결과를 펼쳐놓고 추적하는 거라면 동적 추적(Dynamic Tracing)은 현재 실행되고 있는 커널, 프로세스를 대상으로 특정함수, 변수 등에 대해서 추적이 가능하다. 예를 들면 tcp_sendmsg 라는 커널함수가 불렸는지, 불렸다면 인자는 무엇이었는지, 그 함수의 리턴 값은 무엇인지 언제 몇 번 불렸는지 등을 하나하나 추적 가능하다. 아래와 같이 추적하고 싶은 함수명 tcp_sendmsg(커널함수)을 probe 로 추가하고

```
$ perf probe --add tcp_sendmsg
Added new event:
  probe:tcp_sendmsg    (on tcp_sendmsg)

You can now use it in all perf tools, such as:

    perf record -e probe:tcp_sendmsg -aR sleep 1
```

[그림 17] Dynamic Tracing 1

아래와 같이 --list 옵션을 통해서 다음과 같이 확인할 수 있다.

```
  probe:tcp_sendmsg    (on tcp_sendmsg)
$ perf probe --list
```

[그림 18] Dynamic Tracing 2

지정해둔 사용자 정의 이벤트 'probe:tcp_sendmsg'를 4 초간 추적하고 상태를 본다.
-R 옵션은 --raw-samples 라는 옵션으로 열려있는 모든 카운터들로부터 sample 기록 모두를 수집한다는 내용이다. 아래와 같이 따로 추가하지 않아도 기본옵션으로 지정된다.

```
$ perf record -e probe:tcp_sendmsg -aRg sleep 4
$ perf report --stdio
# To display the perf.data header info, please use --header/--header-only options.
#
# Total Lost Samples: 0
#
# Samples: 6  of event 'probe:tcp_sendmsg'
# Event count (approx.): 6
#
# Children      Self  Trace output
# ........  ........  ................
#
   100.00%   100.00%  (ffffffff81752c40)
             |
             |--33.33%--0x732f796470732f74
             |          0x79d
             |          entry_SYSCALL_64_fastpath
             |          sys_write
             |          vfs_write
             |          __vfs_write
             |          sock_write_iter
             |          sock_sendmsg
             |          tcp_sendmsg
```

[그림 19] Dynamic Tracing 3