2018
아이펀팩토리
데브데이

코드 수정 없이
리눅스 게임 서버 성능 평가

김진욱
<jinuk.kim@ifunfactory.com>

소개

아이펀팩토리 (2013-)

- 게임 서버 프레임워크 (C++, C#)
- 게임 서버 운영 플랫폼 (Python)

넥슨 코리아 (2012-2013)

- 게임 서비스를 위한 사설 클라우드 (Python)

엔씨소프트 (2007-2012)

- 게임 서버 라이브러리 (C++)
- 크래시 덤프 수집 및 분석 서비스 (C++, C#, Python)
- 게임 서버 배포 서비스 (C#)

Contents.

목표

01

목표

코드를 손대지 않고 성능 문제를 분석할 수 있을까?

실제 서비스를 직접 분석해도 문제 없을까?

Linux eBPF + BCC로 정말로 가능할지 확인

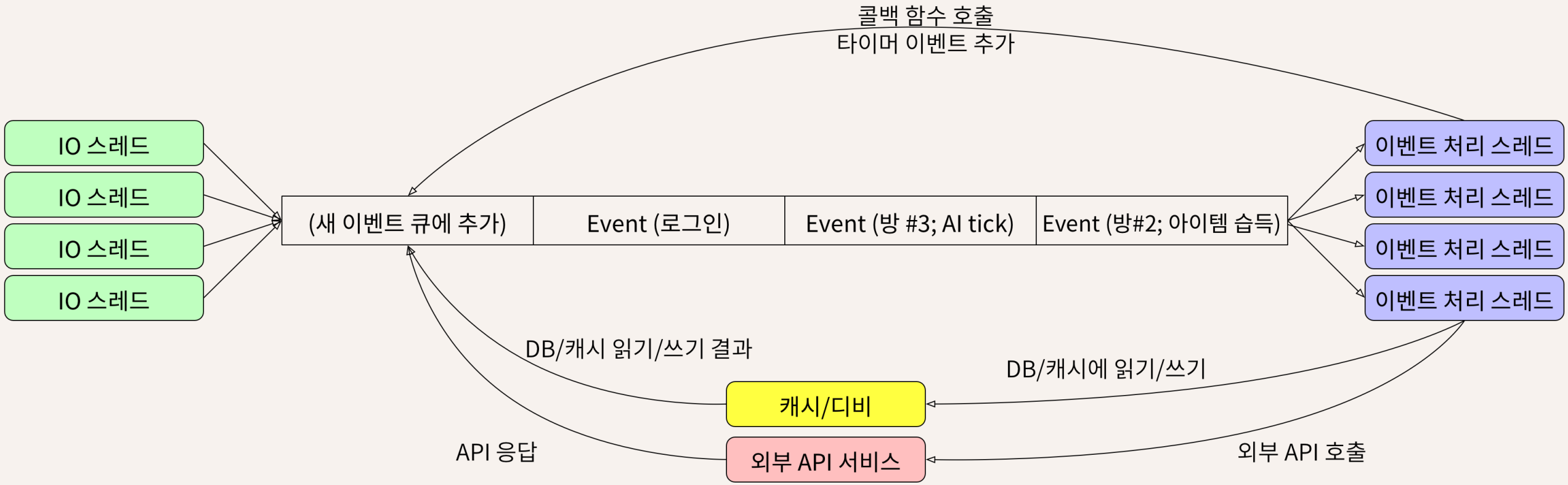
성능 분석: 코드 직접 수정

02

성능 분석: 코드 직접 수정

- 게임 서버 성능 분석하기 (2017 아이펀팩토리 데브데이)
- 게임 서버 코드에 성능 카운터 추가 + 이벤트 별로 분석
  - 대기열에서 기다리는 시간
  - DB 에서 가져오는 객체 수 / 캐시에서 가져오는 비율 추적
  - CPU 위에서 코드가 도는 시간 분석
  - 잠금 (lock) 획득에 걸리는 시간 분석
  - 외부 시스템 (DB, redis, 3rd party API) 호출 시간 분석
- 이에 대한 분포를 추적

분석 예시: 모델링


콜백 함수 호출
타이머 이벤트 추가
IO 스레드
IO 스레드
IO 스레드
IO 스레드
(새 이벤트 큐에 추가)
Event (로그인)
Event (방 #3; AI tick)
Event (방#2; 아이템 습득)
이벤트 처리 스레드
이벤트 처리 스레드
이벤트 처리 스레드
이벤트 처리 스레드
DB/캐시 읽기/쓰기 결과
DB/캐시에 읽기/쓰기
캐시/디비
API 응답
외부 API 서비스
외부 API 호출

분석 예시: 측정 결과

Name	Count	Queued	DB/Cache IO	mean exec	(max)	DB rows	Cache Hit (%)
StartLoading lambda	800	133.60	132.49	93.13	225.50	5.00	0.00
Match::_HandleTimer	400	89.84	0.00	9.85	82.18	0.00	0.00
StartGame lambda	400	86.78	46.23	20.47	95.65	12.00	0.00
UpdateQuest lambda	800	69.00	22.95	12.55	67.95	4.00	0.00
DoUpdateLeaderboard_SubmitScore_cb	1600	67.18	0.00	2.67	33.59	0.00	0.00
_RefreshKey_ExpireAsync_cb	3200	67.16	0.00	0.93	33.59	0.00	0.00
Match::_HandleTimer_Timer	38798	66.26	0.00	0.07	33.61	0.00	0.00
bnb::Match::_OnUpdateUserBEP()::__lambda101	19246	65.71	0.00	0.16	34.12	0.00	0.00
MessageHandler(cs_match_move)	172172	65.26	0.00	0.07	32.65	0.00	0.00

- 큐 대기 / DB 접근 / CPU 사용 시간을 측정 (최소/최대/평균/분산)
- DB 에서 가져오는 데이터 수와 캐시에서 가져오는 비율을 측정
- 해당 데이터를 바탕으로 시스템 설정 / 개별 이벤트를 최적화

장점

- 개별 함수 혹은 콜 스택 수준 대신에 개발자가 원하는 추상화 수준에서 성능을 분석한다.
- 실행 방식/특성에 따른 측정 방법의 차이를 반영하기 쉽다
  - 다중 프로세스 / 다중 스레드 / 파이버 (fiber) / 코루틴 등등
  - 블럭킹 방식
  - 락 획득 방식 / 락 획득 경쟁 정도
  - 재시도 로직
  - 외부 시스템의 지연 시간

단점

- 미리 실행 / 성능 모델을 만들어야 한다.

- 일부 코드 경로에만 적용하면 효과가 없거나 적다.

- 개발자가 작업할 부분이 많다
  - 성능 모델이나 부하 걸릴 부분에 대한 가정이 틀리면 쓸모 없다
  - 가설 변경 / 확인 코드 추가 / 검증 싸이클이 길다

툴 체인을 이용한 성능 분석

03

개발자가 직접 수정하는 걸 피한다

개발 툴체인을 이용한다

툴 체인을 이용한 성능 분석

개발자가 코드를 수정하지 않는다

컴파일러 옵션을 추가하거나 별도 라이브러리 링크

혹은 실행 방식을 바꾼다

- 네이티브 코드를 성능 분석/디버깅 인터프리터에서 실행

예: CPU 프로파일링

- **GNU Compiler Collection (GCC)**
  - `-pg` 컴파일러 플래그 **(gprof)**
- **LLVM/Clang**
  - `-fprofile-instr-generate`
- **Visual C++ instrumentation**
- **Valgrind Callgrind**

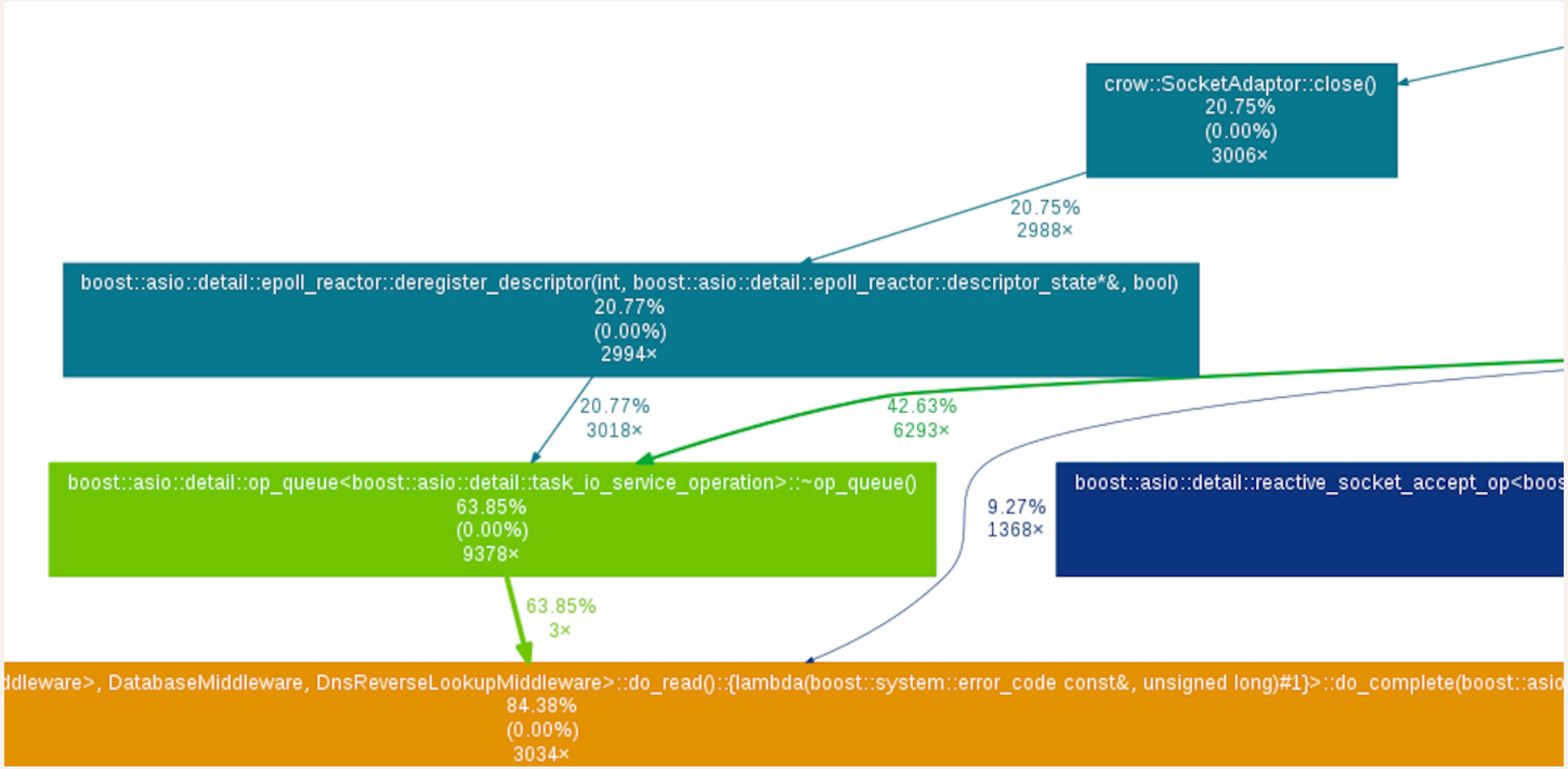
실제로 일어나는 일

- 컴파일러 수준에서 코드 삽입 / 혹은 링커 수준에서 동적 심볼 연결
- 실행 중 모든 순간 / 혹은 일정 비율의 샘플링으로 통계 데이터 생성
- 생성된 통계 데이터 저장
- 저장한 데이터를 각 툴에 맞는 방식으로 추가 처리 / 변환

GProf 실행 예: 실행 시간 테이블

```
 %   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call  name
14.29      0.03     0.03   207323     0.00     0.00  crow::json::escape
14.29      0.06     0.03     3059     0.01     0.02  crow::CrowApp::handle
 9.52      0.08     0.02     2638     0.01     0.02  crow::json::dump_internal
 4.76      0.09     0.01   307190     0.00     0.00  _Hashtable<…>::insert_unique
 4.76      0.10     0.01   208356     0.00     0.00  _Hashtable<…>::_M_rehash
 4.76      0.11     0.01    19129     0.00     0.00
                                vector<crow::json::wvalue>::_M_emplace_back_aux
 4.76      0.12     0.01    11961     0.00     0.00
     crow::Connection<crow::SocketAdaptor, crow::CrowApp>::cancel_deadline_timer
```

GProf 실행 예: 콜 그래프



툴 체인 코드 삽입방식의 문제

성능 오버헤드

- gprof: 30% - 300%
- Valgrind: 400% - 4,000%

측정할 수 없는 외부 코드

- 3rd 파티 라이브러리 (.dll, .so)
- 운영 체제 내부

장점

개발자가 코드를 관리하지 않아도 된다.

다양한 외부 툴 / 라이브러리가 있다

- 측정 대상 (CPU, 힙, 캐시 적중률, 락 경쟁 등)
- 구현 방식
- 시각화 방식

단점

측정 대상마다 툴 체인에 해당 기능이 필요

측정 대상에 대한 툴체인 지원이 필요

- 디스크 IO 시간이 10ms 이상 걸린 파일 목록이 알고 싶다면?

측정 대상에 대한 세부 제어 불가

- 특정 함수에서 할당된 메모리 양만 궁금하다면?
- 어떤 함수의 실행 시간 분포를 알고싶으면?

외부 관찰을 통한 분석

04

외부 툴을 써서 프로그램 관찰

CPU 프로파일링 다시보기

- 코드를 삽입하지 않고도 측정할 수 있다.
- Linux perf 커널 이벤트 카운터 (kernel v2.6+)
- Linux eBPF 트레이싱 프레임웍 (kernel v3.18+; v4.9+)
- Google Performance Tools
- Visual C++ 성능 도구: CPU 샘플링

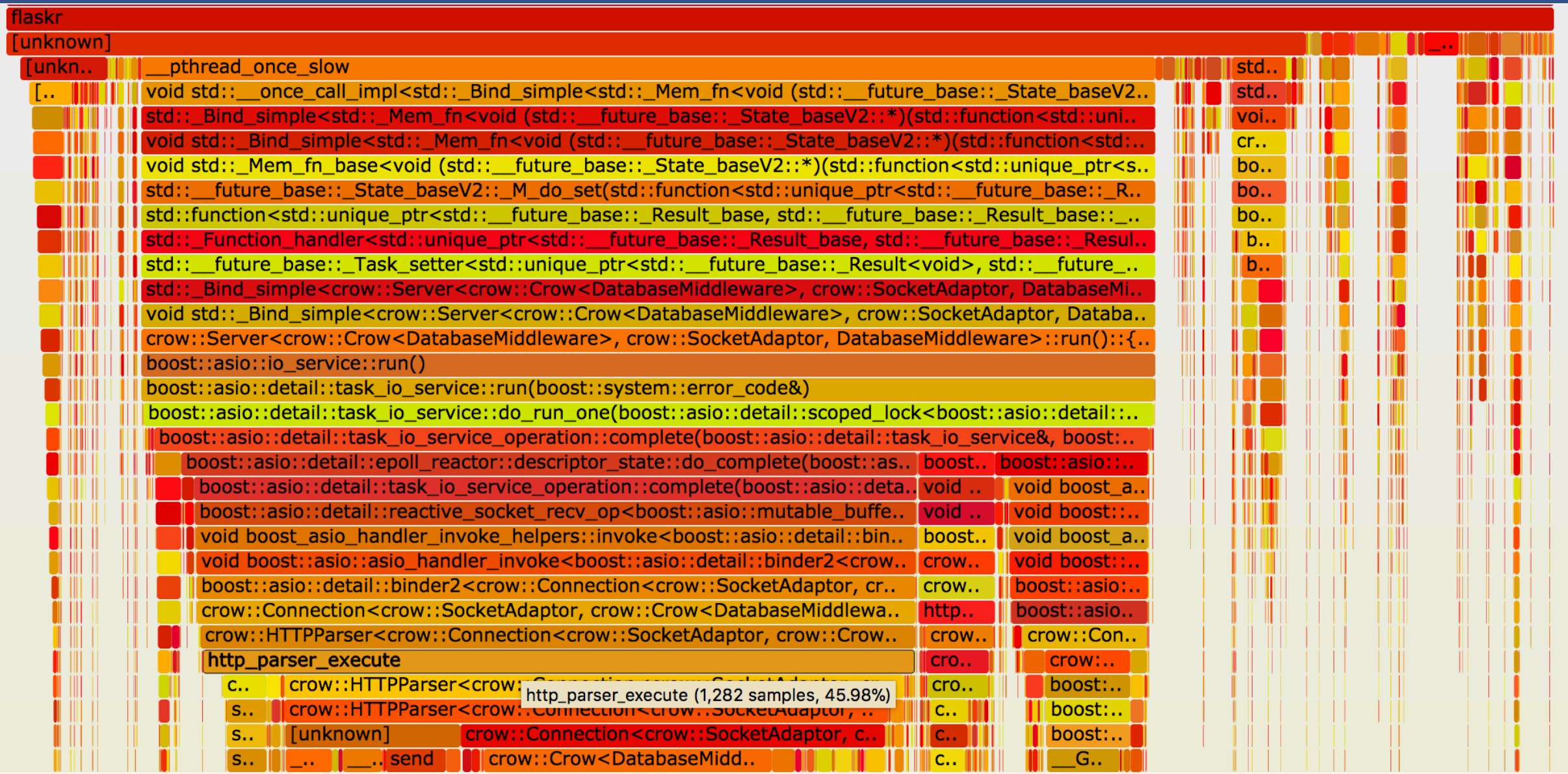
실제로 일어나는 일

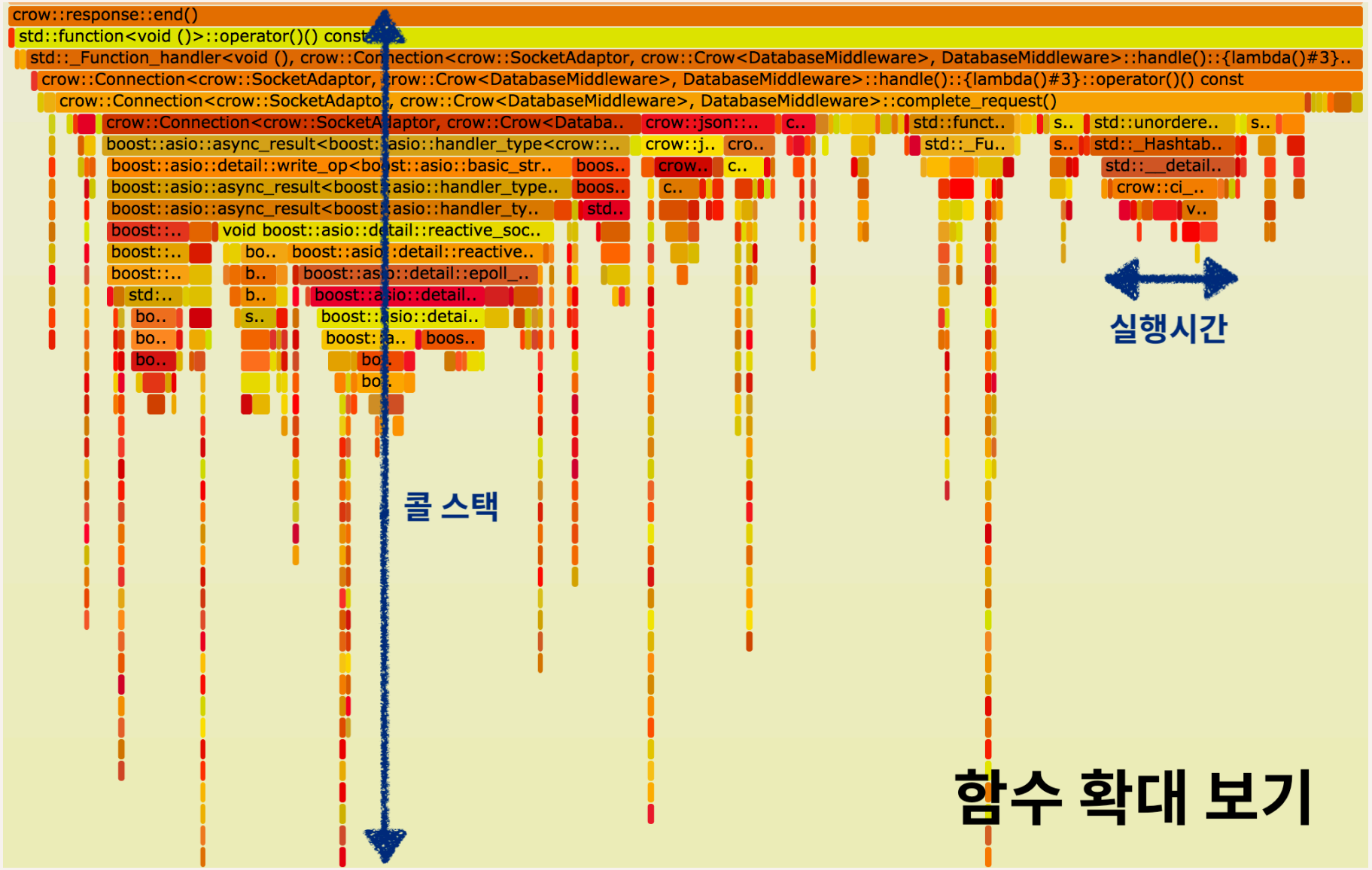
- 주기적으로 스레드를 멈춘다 (초당 50번 - 200번 정도)
- 해당 순간의 CPU 가 실행 중인 콜 스택의 빈도를 저장한다
- 저장한 데이터를 내보낸다
- 저장한 데이터를 각 툴에 맞는 방식으로 추가 처리 / 변환

예: CPU 프로파일링

Linux v4.13 eBPF + BCC profile: 콜스택 별 실행 시간 측정

Flamegraph: 실행 시간 / 콜 스택에 대한 시각화 도구
https://github.com/brendangregg/FlameGraph


flaskr
[unknown]
[unkn..
__pthread_once_slow
std..
[..
void std::__once_call_impl<std::_Bind_simple<std::_Mem_fn<void (std::__future_base::_State_baseV2..
std..
std::_Bind_simple<std::_Mem_fn<void (std::__future_base::_State_baseV2::*)(std::function<std::uni..
voi..
void std::_Bind_simple<std::_Mem_fn<void (std::__future_base::_State_baseV2::*)(std::function<std:..
cr..
void std::_Mem_fn_base<void (std::__future_base::_State_baseV2::*)(std::function<std::unique_ptr<s..
bo..
std::__future_base::_State_baseV2::_M_do_set(std::function<std::unique_ptr<std::__future_base::_R..
bo..
std::function<std::unique_ptr<std::__future_base::_Result_base, std::__future_base::_Result_base::_..
bo..
std::_Function_handler<std::unique_ptr<std::__future_base::_Result_base, std::__future_base::_Resul..
b..
std::__future_base::_Task_setter<std::unique_ptr<std::__future_base::_Result<void>, std::__future_..
b..
std::_Bind_simple<crow::Server<crow::Crow<DatabaseMiddleware>, crow::SocketAdaptor, DatabaseMi..
void std::_Bind_simple<crow::Server<crow::Crow<DatabaseMiddleware>, crow::SocketAdaptor, Databa..
crow::Server<crow::Crow<DatabaseMiddleware>, crow::SocketAdaptor, DatabaseMiddleware>::run()::{..
boost::asio::io_service::run()
boost::asio::detail::task_io_service::run(boost::system::error_code&)
boost::asio::detail::task_io_service::do_run_one(boost::asio::detail::scoped_lock<boost::asio::detail::..
boost::asio::detail::task_io_service_operation::complete(boost::asio::detail::task_io_service&, boost::..
boost::asio::detail::epoll_reactor::descriptor_state::do_complete(boost::as..
boost..
boost::asio::..
boost::asio::detail::task_io_service_operation::complete(boost::asio::deta..
void ..
void boost_a..
boost::asio::detail::reactive_socket_recv_op<boost::asio::mutable_buffe..
void ..
void boost::..
void boost_asio_handler_invoke_helpers::invoke<boost::asio::detail::bin..
boost..
void boost_a..
void boost::asio::asio_handler_invoke<boost::asio::detail::binder2<crow..
crow..
void boost::..
boost::asio::detail::binder2<crow::Connection<crow::SocketAdaptor, cr..
crow..
boost::asio::..
crow::Connection<crow::SocketAdaptor, crow::Crow<DatabaseMiddlewa..
http..
boost::asio..
crow::HTTPParser<crow::Connection<crow::SocketAdaptor, crow::Crow..
crow..
crow::Con..
http_parser_execute
cro..
crow::..
c..
crow::HTTPParser<crow::
cro..
boost::..
http_parser_execute (1,282 samples, 45.98%)
s..
crow::HTTPParser<crow::
c..
boost::..
s..
[unknown]
crow::Connection<crow::SocketAdaptor, c..
c..
boost::..
s..
_..
__..
send
crow::Crow<DatabaseMidd..
c..
__G..


crow::response::end()
std::function<void ()>::operator()() const
std::_Function_handler<void (), crow::Connection<crow::SocketAdaptor, crow::Crow<DatabaseMiddleware>, DatabaseMiddleware>::handle()::{lambda()#3}..
crow::Connection<crow::SocketAdaptor, crow::Crow<DatabaseMiddleware>, DatabaseMiddleware>::handle()::{lambda()#3}::operator()() const
crow::Connection<crow::SocketAdaptor, crow::Crow<DatabaseMiddleware>, DatabaseMiddleware>::complete_request()
crow::Connection<crow::SocketAdaptor, crow::Crow<Databa..
crow::json::..
c..
std::funct..
s..
std::unordere..
s..
boost::asio::async_result<boost::asio::handler_type<crow::..
crow::j..
cro..
std::_Fu..
s..
std::_Hashtab..
boost::asio::detail::write_op<boost::asio::basic_str..
boos..
crow..
c..
std::__detail..
boost::asio::async_result<boost::asio::handler_type..
boos..
c..
crow::ci_..
boost::asio::async_result<boost::asio::handler_ty..
std..
v..
boost::..
void boost::asio::detail::reactive_soc..
boost::..
bo..
boost::asio::detail::reactive..
boost::..
b..
boost::asio::detail::epoll_..
std::..
b..
boost::asio::detail..
bo..
s..
boost::asio::detai..
bo..
boost::a..
boos..
bo..
bo..
bo..
콜 스택
실행시간
함수 확대 보기

장점

툴 체인 기반 프로파일링에 비해서 가볍다
대부분 OS에서 지원하는 기능을 이용한다

측정 대상의 코드 수정 / 빌드가 필요하지 않다

필요한 정보가 있다면 (심볼), 라이브러리나 OS도 측정한다

단점

OS 지원이 필요하다

버전이 낮은 OS에서는 기능이 제한된다

OS 코드가 바뀌면 잘 돌던 측정 방법이 안돌수 있다!

- Linux v4.13 randstruct
- eBPF 모듈 선언에 추가 `#define` 필요

다음 문제들은 어떻게 측정할까?

- Disk I/O 가 느린 코드는 어디인가?
- CPU 수보다 활성 스레드 수가 많은가?
- 잠금 (lock) 경쟁이 심한가?
- 외부 API 호출이 느린가?
- 시스템 콜이 너무 많은가?
- 시스템 콜이 오래 걸리는가?

유연한 트레이싱 도구를 쓴다

Linux

- eBPF (kernel v3.18+; 가능하면 4.9+ 필요)
- perf (kernel v2.6+; 제한적)

Windows

- Event Tracing for Windows (ETW; Win 8+)
- Windows Event Logging (제한적; Win 2k+)

취합 값이 아닌 분포를 볼 수 있어야한다

```
rein@rein-dev ~> sudo iostat -x 10 10
Linux 4.13.0-37-generic (rein-dev)      03/27/2018      _x86_64_

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.38    0.00    0.13    0.02    0.00   99.47

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s
sda               0.08     1.68    0.32    1.97     7.07   176.28
sdb               0.00     0.00    0.00    0.00     0.47     0.61
dm-0              0.00     0.00    0.00    0.00     0.46     0.61
```

```
rein@rein-dev ~> sudo /usr/share/bcc/tools/biolatency
Tracing block device I/O... Hit Ctrl-C to end.
^C
     usecs               : count     distribution
         0 -> 1          : 0        |                                        |
         2 -> 3          : 0        |                                        |
         4 -> 7          : 0        |                                        |
         8 -> 15         : 0        |                                        |
        16 -> 31         : 0        |                                        |
        32 -> 63         : 10       |                                        |
        64 -> 127        : 271      |****                                    |
       128 -> 255        : 1207     |********************                    |
       256 -> 511        : 2496     |****************************************|
       512 -> 1023       : 640      |**********                              |
      1024 -> 2047       : 2        |                                        |
      2048 -> 4095       : 10       |                                        |
      4096 -> 8191       : 1        |                                        |
```

평균보다는 중앙값

가장 흔한 케이스는?

최단/최장 케이스?

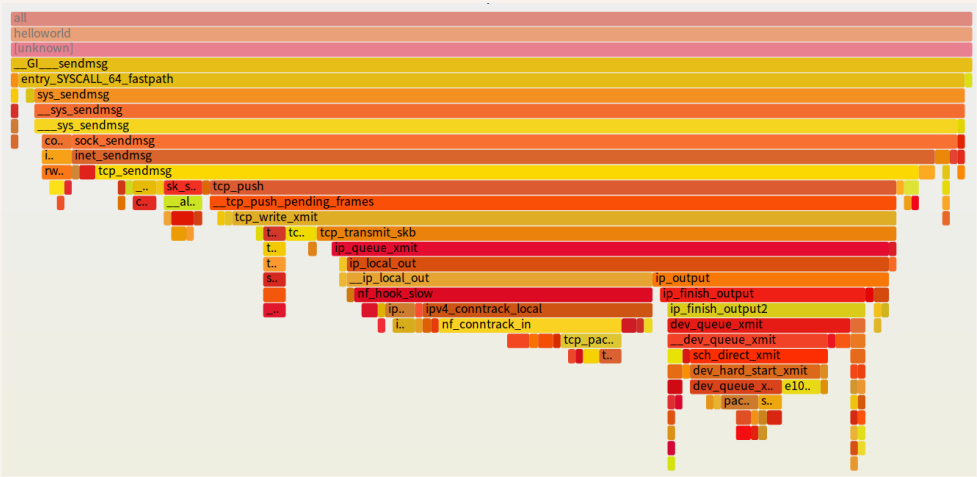
안전하고 분리된 분석 도구가 필요

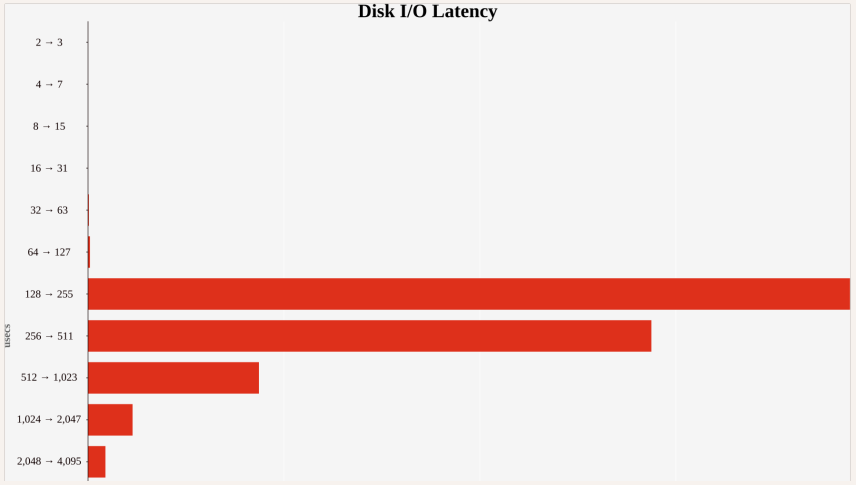
분석 도구와 분석 대상이 분리되어야 한다

분석 도구의 버그로 분석 대상이 크래시하면 안된다

분석 도구 수정 후에 분석 대상을 다시 빌드하거나
재실행하지 않고 진행 가능해야 한다

문제 분석에 적합한 시각화





시각화를 통해서 문제 파악이 쉬워야한다

필터링 방법 제공

직접 시각화 방법을 수정/추가 가능해야

I FUN FACTORY

Linux eBPF + BCC

- eBPF: 리눅스 커널에 들어있는 트레이싱/디버깅 도구

- BCC: Python / lua / go 등에서 eBPF를 쓰기 위한 툴체인

eBPF

OS 커널 수준에서 동작하는 별도의 어셈블리 언어
(x86-64/arm64 유사)

커널 내에서 컨텍스트 스위칭 없이 돌아서 부하가 작다

커널 내의 샌드박스 형태의 인터프리터 VM에서 실행한다

- 트레이싱/디버깅 중 크래싱할 가능성은 (거의) 없다
- 실행 전에 크래시할만한 부분을 검사한다
- 실행 하기 전에 JIT 컴파일해서 속도를 올린다

eBPF: 무엇을 할 수 있는가?

커널 수준에서 다양한 이벤트에서 데이터를 수집하게 해준다

- 시스템 콜 혹은 커널 함수 호출
- 커널 함수 호출
- (바이너리 수준에서 정의한) 트레이싱 지점

수집한 데이터를 커널 메모리에 모을 수 있다

- 단순 값이나 콜 스택을 해시맵 형태로 저장

외부(유저영역)에서 모은 데이터를 읽거나 업데이트할 수 있다

BCC

- eBFP Compiler Collection
- Python / lua / go 같은 고수준 언어로 성능 측정 초기화/후처리
- C 로 정의한 eBPF 모듈을 컴파일하고 커널에 로드
- 커널은 eBPF 모듈을 실행해서 필요한 정보 수집
- 모은 정보를 python / lua / go 코드로 돌아와서 데이터 후처리

eBPF + BCC: 할 수 있는 일

개발자가 필요로하는 CPU, 네트워크 스택, 디스크 I/O, 메모리 등등의 정보를 모두 얻을 수 있다

성능 테스트 대상을 재시작하지 않고도 eBPF + BCC 만 수정하고 재시작하면 (다시) 측정할 수 있다

아래 과정을 반복해서 성능 문제를 해결:

1. 가설을 세우고, 이를 확인할 수 있는 질문을 만든다
2. 답을 얻을 수 있는 eBPF + BCC 프로그램을 만든다
3. 돌고 있는 프로그램에서 이를 측정한다

BCC 구현 예: 함수 실행 속도 분석

BCC funclatency

특정 혹은 전체 프로세스에서 지정한 함수 실행 시간 분포를 분석

함수 진입 시간 / 함수에서 나오는 시간을 기록한다

해당 시간 차이를 함수 별로 정리

구현 예: eBPF 정의 (1)

```
92  # define BPF program
93  bpf_text = """
94  #include <uapi/linux/ptrace.h>
95
96  typedef struct ip_pid {
97      u64 ip;
98      u64 pid;
99  } ip_pid_t;
100
101 typedef struct hist_key {
102     ip_pid_t key;
103     u64 slot;
104 } hist_key_t;
105
106 BPF_HASH(start, u32);
107 STORAGE
108
109 int trace_func_entry(struct pt_regs *ctx)
110 {
111     u64 pid_tgid = bpf_get_current_pid_tgid();
112     u32 pid = pid_tgid;
113     u32 tgid = pid_tgid >> 32;
114     u64 ts = bpf_ktime_get_ns();
115
116     FILTER
117     ENTRYSTORE
118     start.update(&pid, &ts);
119
120     return 0;
121 }
```

함수 호출 정보 구분용

함수 진입 정보 기록

구현 예: eBPF 정의 (2)

```
123  int trace_func_return(struct pt_regs *ctx)
124  {
125      u64 *tsp, delta;
126      u64 pid_tgid = bpf_get_current_pid_tgid();
127      u32 pid = pid_tgid;
128      u32 tgid = pid_tgid >> 32;
129
130      // calculate delta time
131      tsp = start.lookup(&pid);
132      if (tsp == 0) {
133          return 0;   // missed start
134      }
135      delta = bpf_ktime_get_ns() - *tsp;
136      start.delete(&pid);
137      FACTOR
138
139      // store as histogram
140      STORE
141
142      return 0;
143  }
144  """
```

함수 종료 정보 기록

```
150 if args.pid:
151     bpf_text = bpf_text.replace('FILTER',
152         'if (tgid != %d) { return 0; }' % args.pid)
153 else:
154     bpf_text = bpf_text.replace('FILTER', '')
155 if args.milliseconds:
156     bpf_text = bpf_text.replace('FACTOR', 'delta /= 1000000;')
157     label = "msecs"
158 elif args.microseconds:
159     bpf_text = bpf_text.replace('FACTOR', 'delta /= 1000;')
160     label = "usecs"
161 else:
162     bpf_text = bpf_text.replace('FACTOR', '')
163     label = "nsecs"
164 if need_key:
165     bpf_text = bpf_text.replace('STORAGE', 'BPF_HASH(ipaddr, u32);\n' +
166         'BPF_HISTOGRAM(dist, hist_key_t);')
167     # stash the IP on entry, as on return it's kretprobe_trampoline:
168     bpf_text = bpf_text.replace('ENTRYSTORE',
169         'u64 ip = PT_REGS_IP(ctx); ipaddr.update(&pid, &ip);')
170     pid = '-1' if not library else 'tgid'
171     bpf_text = bpf_text.replace('STORE',
172         """
173     u64 ip, *ipp = ipaddr.lookup(&pid);
174     if (ipp) {
175         ip = *ipp;
176         hist_key_t key;
177         key.key.ip = ip;
178         key.key.pid = %s;
179         key.slot = bpf_log2l(delta);
180         dist.increment(key);
181         ipaddr.delete(&pid);
182     }
183         """ % pid)
184 else:
185     bpf_text = bpf_text.replace('STORAGE', 'BPF_HISTOGRAM(dist);')
186     bpf_text = bpf_text.replace('ENTRYSTORE', '')
187     bpf_text = bpf_text.replace('STORE',
188         'dist.increment(bpf_log2l(delta));')
```

한 프로세스만
관찰하는지 처리

측정 단위 결정

함수마다
그래프를
그릴지 결정

구현 예: BCC + BPF 프로그램 실행

```
198  # load BPF program
199  b = BPF(text=bpf_text)
200
201  # attach probes
202  if not library:
203      b.attach_kprobe(event_re=pattern, fn_name="trace_func_entry")
204      b.attach_kretprobe(event_re=pattern, fn_name="trace_func_return")
205      matched = b.num_open_kprobes()
206  else:
207      b.attach_uprobe(name=library, sym_re=pattern, fn_name="trace_func_entry",
208                      pid=args.pid or -1)
209      b.attach_uretprobe(name=library, sym_re=pattern,
210                         fn_name="trace_func_return", pid=args.pid or -1)
211      matched = b.num_open_uprobes()
212
213  if matched == 0:
214      print("0 functions matched by \"%s\". Exiting." % args.pattern)
215      exit()
216
```

BPF 프로그램 커널에 삽입

측정할 함수 후킹

구현 예: 결과 처리

```
221  # output
222  def print_section(key):
223      if not library:
224          return BPF.sym(key[0], -1)
225      else:
226          return "%s [%d]" % (BPF.sym(key[0], key[1]), key[1])
227
228  exiting = 0 if args.interval else 1
229  dist = b.get_table("dist")
230  while (1):
231      try:
232          sleep(int(args.interval))
233      except KeyboardInterrupt:
234          exiting = 1
235          # as cleanup can take many seconds, trap Ctrl-C:
236          signal.signal(signal.SIGINT, signal_ignore)
237
238      print()
239      if args.timestamp:
240          print("%-8s\n" % strftime("%H:%M:%S"), end="")
241
242      if need_key:
243          dist.print_log2_hist(label, "Function", section_print_fn=print_section,
244              bucket_fn=lambda k: (k.ip, k.pid))
245      else:
246          dist.print_log2_hist(label)
```

빈도수 통계 출력

예제코드에서 문제 찾기

05

간단한 HTTP API 서버 제작

서버 성능 문제 탐색

HTTP API 서버 만들기

네트워크 I/O, DB 처리, 그리고 (간단한) 응용 로직

코드는 짧지만 사용하는 라이브러리/OS 코드의 범위가 넓다

예제 구현: *`flaskr`*

- Crow HTTP 서버 프레임워크
- CrowDB + MariaDB 데이터베이스 처리
- 이를 이용해서 블로그를 위한 RESTful API 를 구현

HTTP 서버 구현하기: 서버 정의

```
std::unique_ptr<CrowApp> CreateApp() {
  std::unique_ptr<CrowApp> _app = std::make_unique<CrowApp>();
  CrowApp &app = *_app;

  CROW_ROUTE(app, "/") ([app=&app](const crow::request& req) {
      return HandleIndex(*app, req);
     });

  return _app;
}


int main(int, char**) {
  auto app = CreateApp();
  app->get_middleware<DatabaseMiddleware>().initialize(
      "mysql://crow:crow@10.10.0.255/crow");
  app->port(18080).concurrency(4).run();  // 4 스레드
  return 0;
}
```

URL / 을 처리하는 HTTP 앱 생성하는 함수

HTTP 앱을 생성하고, 4개의 작업 스레드를 붙여 서버 실행

HTTP 서버 구현하기: 핸들러 정의

```
std::string HandleIndex(CrowApp &app,
                        const crow::request &req) {
  auto &ctxt = app.get_context<DatabaseMiddleware>(req);
  auto result = ctxt.conn->execute<
      int, std::string, std::string>(
      "SELECT id, title, body FROM entries ORDER BY id DESC");

  std::vector<crow::json::wvalue> posts;
  for (auto row : result) {
    crow::json::wvalue post;
    post["id"] = crow::db::get<0>(row);
    post["title"] = crow::db::get<1>(row);
    post["text"] = crow::db::get<2>(row);
    posts.emplace_back(std::move(post));
  }

  crow::json::wvalue res;
  res["posts"] = std::move(posts);

  return crow::json::dump(res);
}
```

DB에서 모든 게시물 획득

JSON 배열로 변환

HTTP 응답 전송

HTTP 서버 구현하기: 미들웨어

```
struct DatabaseMiddleware {
  struct context { ConnPtr conn; };

  DatabaseMiddleware() : engine_() {}

  void initialize(const std::string &connection_string) {
    engine_ = std::make_unique<crow::db::mysql::Engine>(
        connection_string);
  }

  void before_handle(crow::request &, crow::response &, context &ctxt) {
    ctxt.conn = engine_->connect();
  }

  void after_handle(crow::request &, crow::response &, context &) {}

  std::unique_ptr<crow::db::mysql::Engine> engine_;
};


struct DnsReverseLookupMiddleware {
  struct context {};

  void before_handle(crow::request &req, crow::response &, context &ctxt) {
    boost::asio::ip::tcp::resolver resolver(*req.io_service);
    boost::system::error_code ec;
    boost::asio::ip::tcp::resolver::iterator ie, it =
        resolver.resolve(req.remote_addr, ec);
    if (it != ie)
      req.add_header("REMOTE_ADDR", it->host_name());
  }
};
```

MariaDB 용 DB 커넥터

HTTP 요청에 대한 DNS 호스트 주소 처리

구현 목표

4 코어 머신에서 초당 1000요청 이상 처리하기

⇨ 코어마다 초당 250건 이상 처리해야

⇨ 요청 당 허용된 시간 ≦ 4 ms

부하 테스트

siege: HTTP 부하테스트 도구

```
siege -c 32
      -r 16384
      -b
      http://example.com
```

지정한 URL에 32 개의 연결 (-c32)

요청 사이에 간격 없이 (-b)

총 16k개의 요청 전송 (-r 16384)

부하 테스트

초기 구현체로 테스트: 성능 목표 미달

110 요청 / 초
코어 당: 27.5 요청 / 초

요청 당 처리시간: 36 ms

문제의 원인은 무엇일까?

가설 #1: 어디선가 CPU를 많이 쓴다

질문: CPU를 많이 쓰는 코드는 어디인가?

eBPF + BCC: `profile` 명령

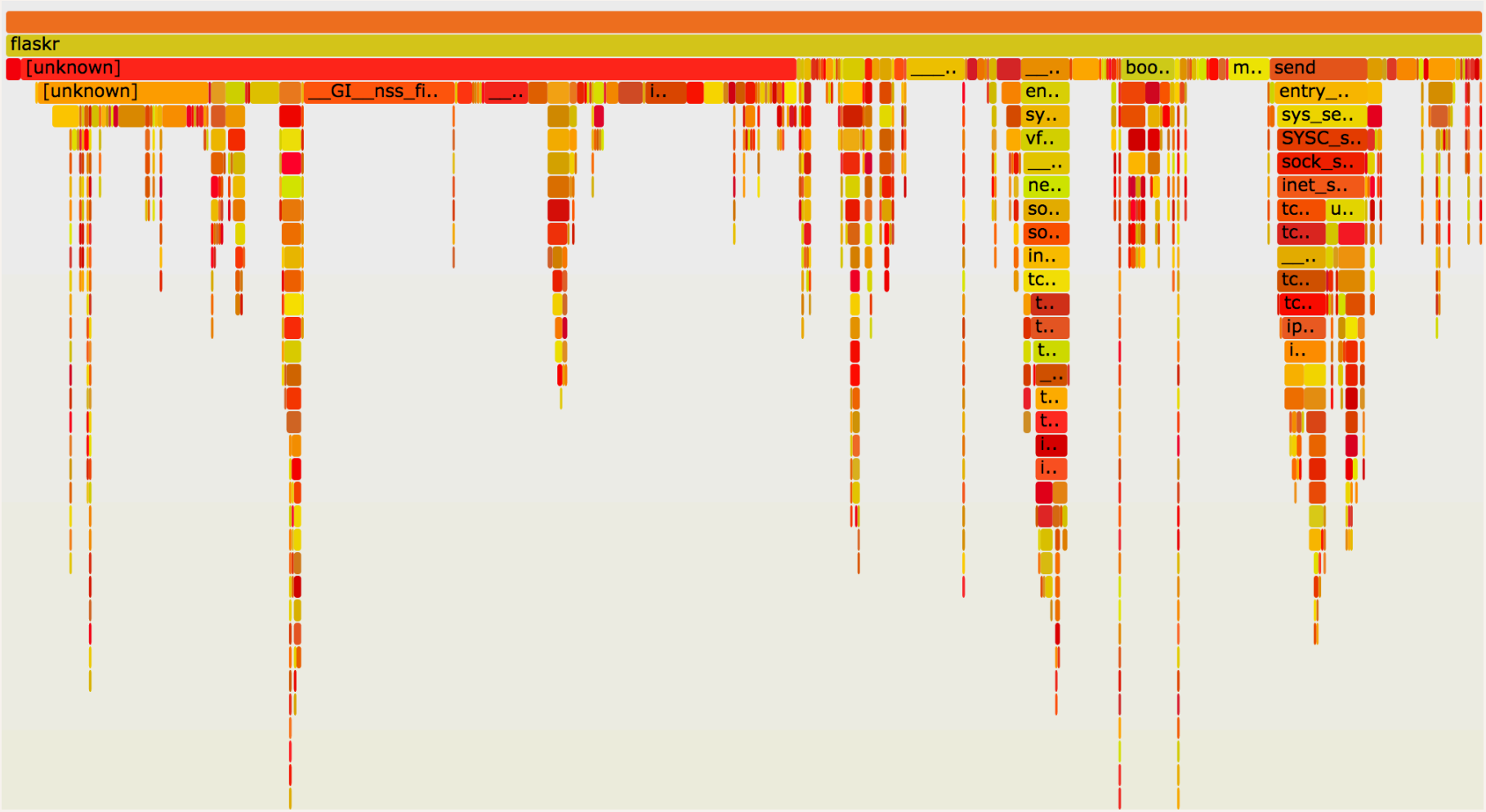
앞서 언급한 샘플링 방식의 프로파일러

- 주기적으로 스레드를 멈추고
- 해당 순간의 콜스택이 얼마나 자주 보이는지 확인

30초 간, 초당 199번 flaskr 프로파일링

```
sudo profile -p $(pgrep -nx flaskr)
             -f
             -F 199 30
```

측정 결과 (1): Flamegraph



측정 결과 (2): 오래 실행한 코드

DNS 설정을 읽는 C 함수 (10%)

- `__GI_nss_files_servent`

여러 시스템 콜 (OS 내부)

- TCP 전송 (6.3%)
- 소켓 닫기 (3%)
- 파일 읽기 (2%)

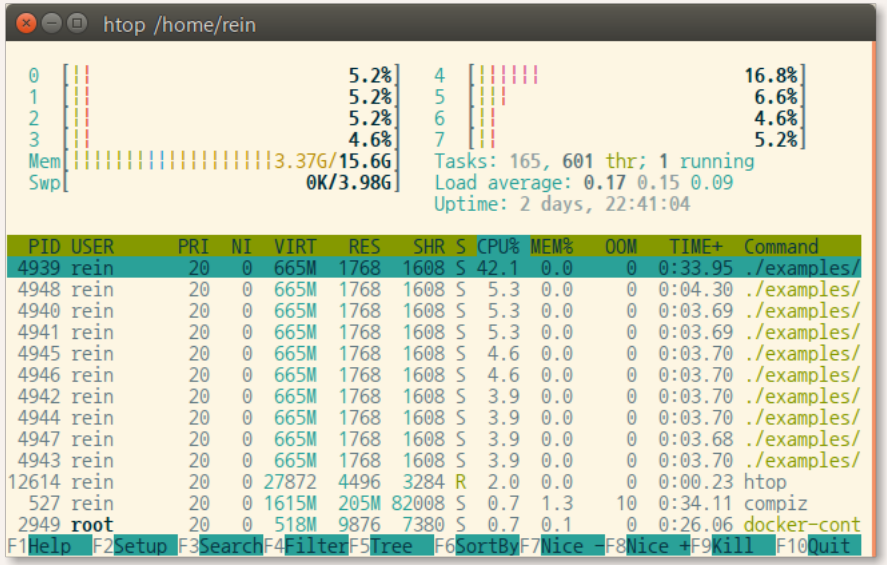
(앞서 보여준) 코드에 해당하는 부분이 없다!

정말 CPU 사용이 문제인가?

다른 방법으로 확인해보자

Linux `htop` 명령 (= 더 나은 `top` 명령)

- CPU 는 겨우 42% 사용 ⇨ CPU 문제가 아닌 것 같다.



가설 #2: 어딘가 대기하는 코드가 있다

#2 데이터베이스는 충분히 빠른가?

많은 경우 HTTP API 서버의 병목은 데이터베이스

문제가 될 수 있는 부분:

- DB 서버에 접속하는데 걸리는 시간
- SQL 쿼리 실행 속도
- SQL 쿼리 결과를 가져오는 지연 시간
- etc.

질문: DB처리가 느린 부분이 있나?

eBPF + BCC: `mysql_qslower`: 오래 걸린 쿼리를 찾아준다

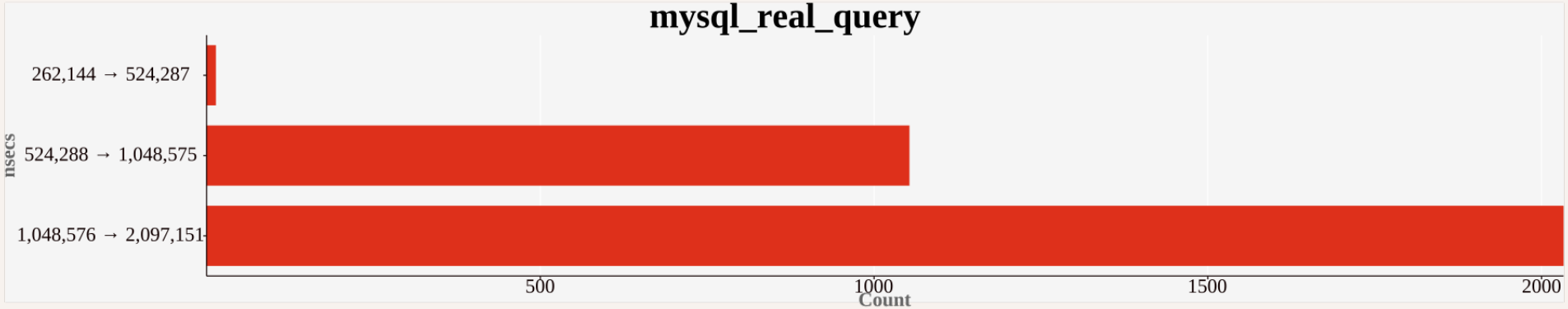
eBPF + BCC: `funclatency`

- **특정 함수의 실행 시간 분포를 측정한다**

MariaDB 의 관련 함수를 모두 측정 (mysql_ 로 시작하는 함수들)

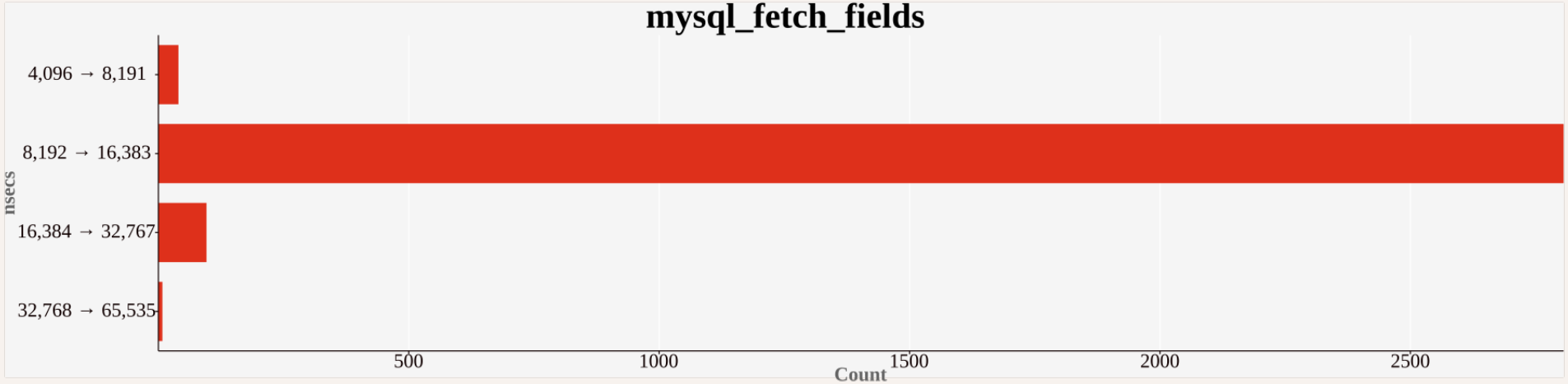
```
sudo funclatency.py -p $(pgrep -nx flaskr)
  -F /usr/lib/mariadb/libmariadb.so:mysql_*
```

측정 결과: SQL 쿼리 실행



- 실행 시간 분포: 0.5 ms - 2 ms 사이
- 2 ms ≪ 36 ms (측정값)

측정 결과: SQL 쿼리 결과 가져오기



- 실행 시간 분포: 0.008 ms - 0.032 ms 사이
- 0.032 ms ≪ 36 ms (측정값)

데이터베이스는 충분히 빠르다

가설 #3: DB와 관련없는대기코드가 있다

질문: 코드의 대기 시간을 측정하자

어떻게 하면 대기하는 코드를 찾을 수 있는가?

대기하게 되면 어떤 일이 일어나는가?

- 할당받은 CPU를 자발적으로 반납
- 반납하는 순간 OS로 진입해서 다른 스레드에게 CPU를 할당

측정 방식:

- OS 에 진입해서 할당받은 CPU를 반납한 시각을 기록
- 다시 CPU를 할당 받은 시간과의 차이 = 대기 시간

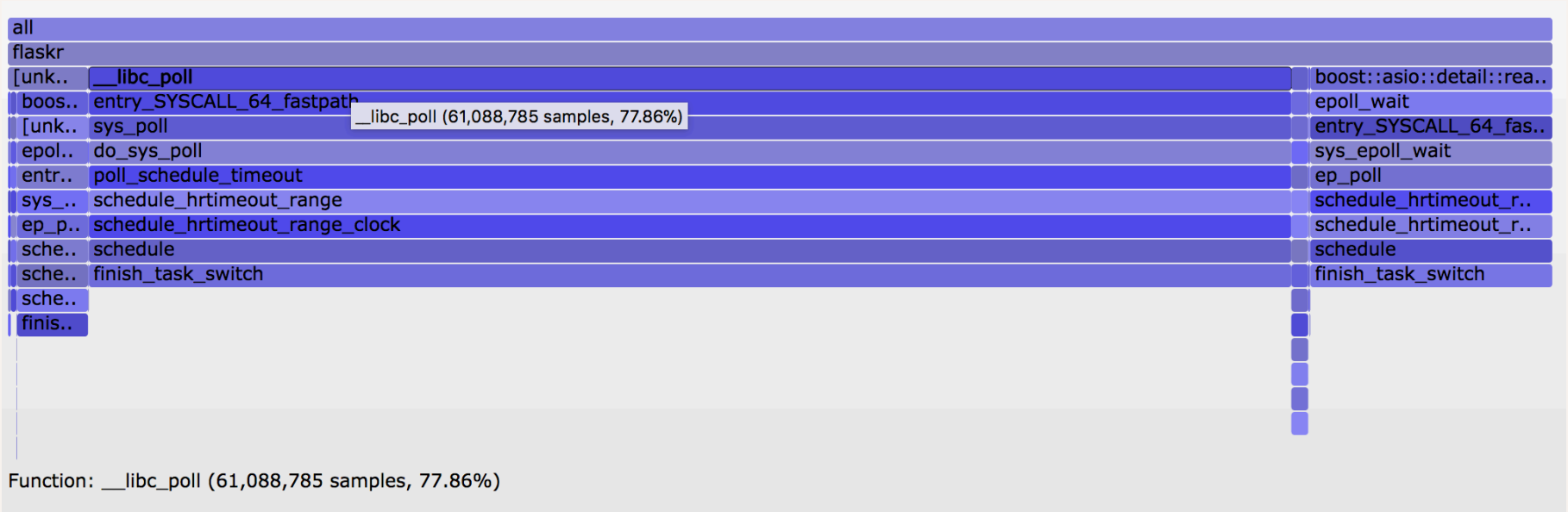
질문: 코드의 대기 시간을 측정하자

eBPF + BCC: offcputime

- 앞서 말한 방식으로 대기 시간을 측정
- 대기한 시점의 콜 스택 별로 대기 시간을 합산

제일 오래 대기하는 코드를 찾아서 분석한다

측정 결과: 대기 시간 Flamegraph

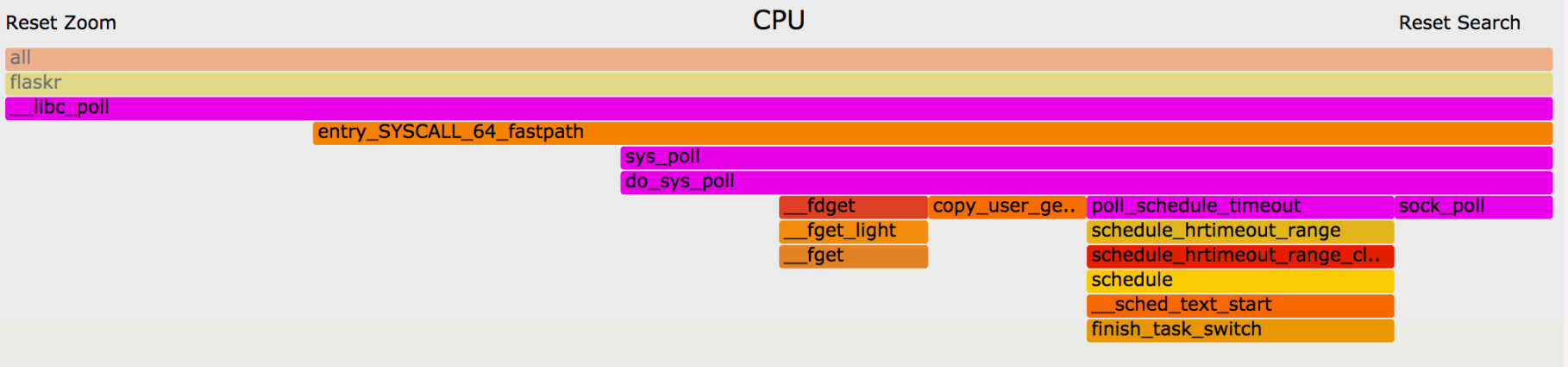


- **poll() : C 함수. 78%; 작성한 코드에 부르는 곳 없음**

- **epoll() : 시스템 콜. 16%; boost::asio 에서 사용**

질문: 누가 poll() 함수를 부르는가?

poll() 함수 호출하는 코드 찾기



프로파일링 결과에서 콜 스택을 확인해보자

C++ 릴리즈 빌드의 최적화

⇨ 인라인 처리 (혹은 샘플링 안되는 경우)

⇨ 콜 스택을 볼 수 없다!

반대 방향에서 콜 스택 추측하기

대기에서 깨어나려면 어떤 일이 벌어져야하는가?

- ⇨ 대기에서 깨어나는 조건을 누군가 충족시켜줬다
- 예: 파일 읽기로 대기 ⇨ 파일 읽기가 끝나서 실행 재개

다른 스레드의 대기 상태를 해제하는 코드를 찾아보자

- 다른 스레드를 깨우는 함수를 호출하는 곳을 찾는다
- 이 때 깨우는 스레드 / 깨어나는 스레드의 콜 스택 빈도를 합산

대기가 풀리는 코드 찾기

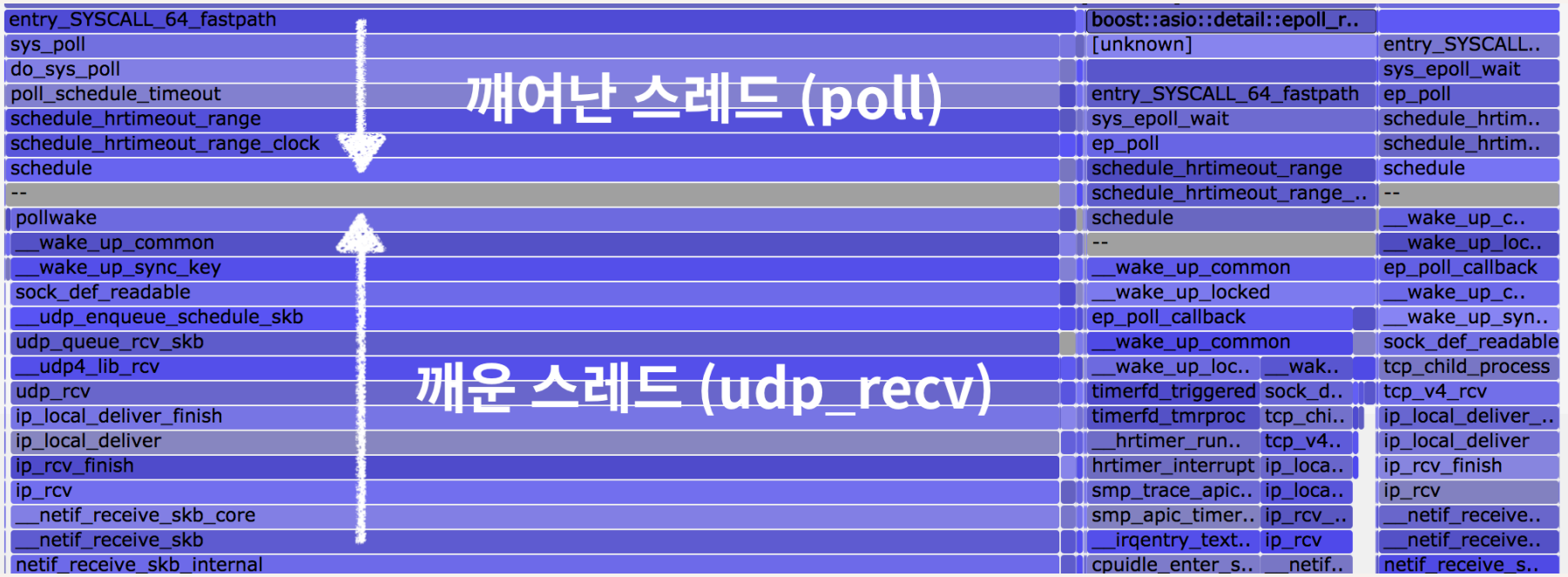
eBPF + BCC: `offwaketime`

다음을 모아 보여준다:

- 다른 스레드를 깨우는 콜 스택 빈도
- 대기에서 깨어난 스레드의 콜 스택 빈도

```
offwaketime -p $(pgrep -nx flaskr) -f
```

측정 결과



- **UDP/IP 메시지를 받고 poll한 스레드가 실행 재개**

질문: 누가 UDP를 쓰는가?

웹 서버에서 UDP의 용도?

웹 서버의 HTTP는 TCP/IP 기반으로 동작한다

UDP의 용도:

- DHCP
- DNS
- RTP
- etc…

DNS 호스트 주소 조회를 위해 사용하고 있다. (미들웨어)

질문: DNS 조회 소요 시간은 어떻게 확인할까?

DNS 조회 소요 시간 측정

DNS 조회를 실제로 처리하는 함수 (C API 에 구현)

- `getnameinfo`
- `getaddrinfo`
- `gethostname`
- etc...

이 함수들의 실행 시간 분포를 측정한다

eBPF + BCC: `funclatency`

측정 결과

- `getnameinfo` 실행 시간 분포
- 32ms - 134ms
- 측정치와 유사한 수준 (36ms)

해결책

문제: 대부분의 리눅스 배포판은 DNS 쿼리를 캐싱하지 않는다

DNS 쿼리를 캐싱하는 로컬 DNS 서버 dnsmasq 를 구동

처리량: 1,567 요청 / 초

- 코어 당 390 요청 / 초 수준, 요청 당 2.5 ms

결과: 14배 향상

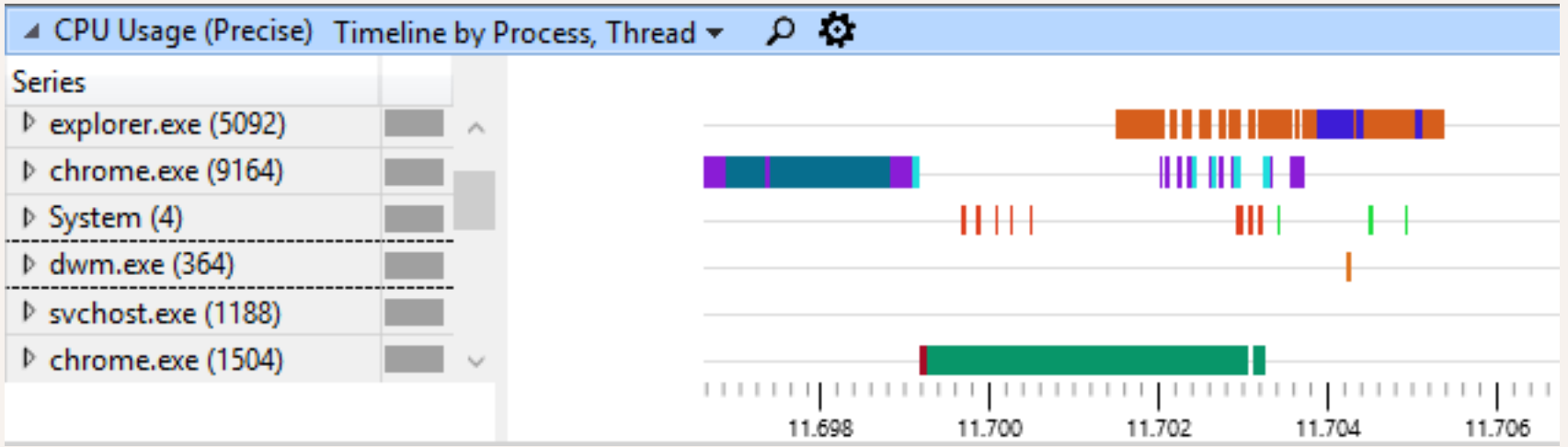
- 처리량: 110 요청 / 초 vs. 1,567 요청 / 초

덤: MS Windows + ETW

GUI를 지원 + 실행/대기시간 분석 가능

스케줄링 이외에서는 추가적인 트레이싱 지점 수는 적다

ETW / Xperf



참고: [The Lost Xperf Documentation on CPU Scheduling]

더 할 수 있는 일

06

시스템 레벨 분석 도구들

미래의 안드로이드
성능 분석 / 디버깅 도구

시스템 성능 분석

프로세스 수준을 벗어나는 문제 찾기 (runqlat)

- 스레드 수가 적절한가 (스레드를 너무 많이 생성하는가?)

Disk I/O 문제 찾기

- 어떤 파일에 대한 I/O가 가장 느린가 (fileslower)
- 혹은 가장 빈번한가 (filetop)

파일 시스템 접근이 느린 것 찾기

- 파일 시스템 종류 별로 ext4slower, btrfsslower, 등

디버깅 도구

할당한 후 해제하지 않은 메모리 찾기 (memleak)

- 할당한 후 30분째 해제하지 않는 메모리에 대한 콜 스택 목록은?

데드락 찾기 (deadlock_detector)

- mutex 획득 순서를 따져서 획득할 수 없는 순서를 찾는다

차세대 Android 성능 분석/디버깅

https://lwn.net/Articles/742363/

ANNOUNCE: bpfd - a remote proxy daemon for executing bpf code (with corres. bcc changes)

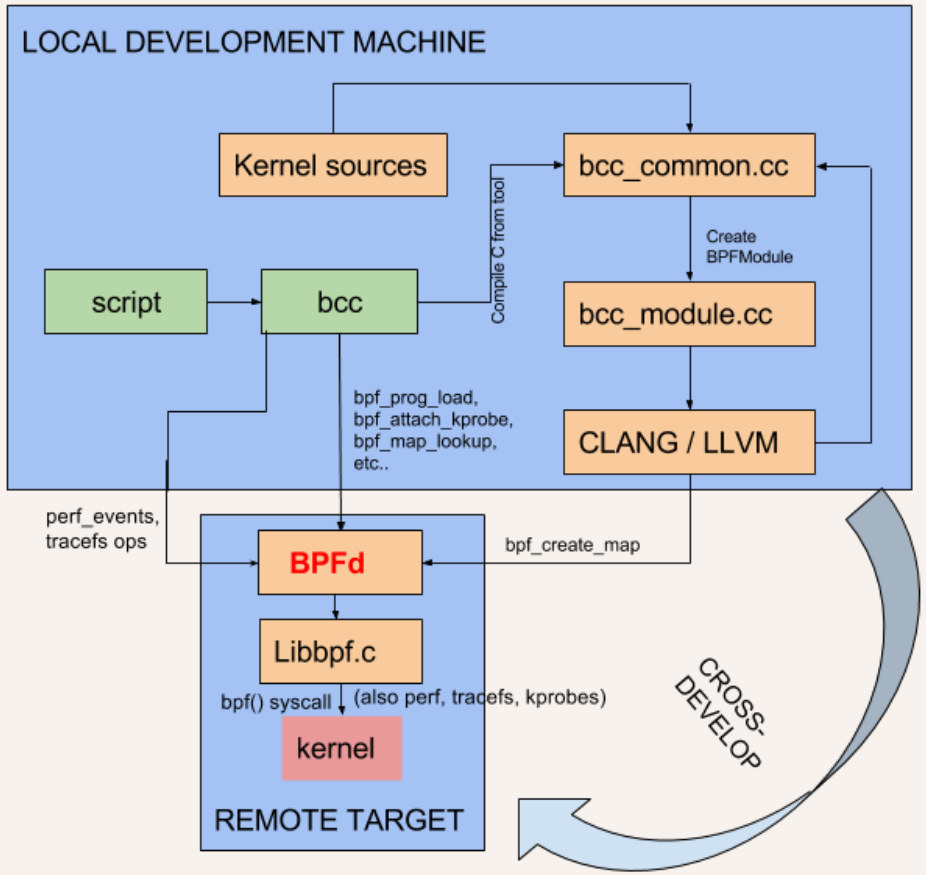
From: Joel Fernandes <joelaf-AT-google.com>
To: LKML <linux-kernel-AT-vger.kernel.org>, iovisor-dev-AT-lists.iovisor.org
Subject: ANNOUNCE: bpfd - a remote proxy daemon for executing bpf code (with corres. bcc changes)
Date: Fri, 29 Dec 2017 00:58:31 -0800
Message-ID: <CAJWu+orZ9Hh-qy+ofdULQXobwBqDZSLEMhqZga7m2iWVuncsYQ@mail.gmail.com>
Cc: Alexei Starovoitov <ast-AT-kernel.org>, bgregg-AT-netflix.com, bblanco-AT-plumgrid.com, Alison Chaiken <alison-AT-she-devel.com>, "Cc: Android Kernel" <kernel-team-AT-android.com> Mathieu Desnoyers <mathieu.desnoyers-AT-efficios.com>, Josef Bacik <jbacik-AT-fb.com>, Yonghong Song <yhs-AT-fb.com>, Steven Rostedt <rostedt-AT-goodmis.org>
Archive-link: Article

```
Hi Guys,

I've been working on an idea I discussed with other kernel developers
(Alexei, Josef etc) last LPC about how to make it easier to run bcc
tools on remote systems.

Use case
========
Run bcc tools on a remotely connected system without having to load
the entire LLVM infrastructure onto the remote target and have to sync
the kernel sources with it.
On architecture such as ARM64 especially, its a bit more work if you
were to run the tools directly on the target itself (local to the
target) because LLVM and Python have to be cross-compiled for it
```

차세대 Android 성능 분석/디버깅



https://github.com/joelagnel/bpfd

커널에서 데이터를 수집하는 eBPF 코드는 x86-64 / arm64 를 구분하지 않는다

1. 개발자 머신에서 성능 분석 코드를 작성 + LLVM 컴파일
2. Android 기기에 업로드 및 실행
3. 측정이 끝나면 데이터를 개발자 머신으로 가져온다
4. 개발자 머신에서 분석/통계처리/시각화 처리

차세대 Android 성능 분석/디버깅

BPFd 는 아직 개발 단계
(https://github.com/joelagnel/bpfd)

eBPF + BCC 는 최신 커널 필요 (v4.6+; 가능하면 v4.9+)

- Android 8.0 Oreo 의 최소 버전은 v4.4

다만 AOSP 8.0 는 v4.10; 분석 용으로 올려서 시험할 수 있다

eBPF + BCC를 사용한 성능 분석

운영체제 수준의
트레이싱을 사용한
유연한 성능 분석

크래시 걱정 없이
동작 중인
서비스를 분석

가설 검증을
반복해서
빠르게 문제 분석

Thank You!

Q&A

경기도 성남시 분당구 대왕판교로 660, 유스페이스1 B동 606호 아이펀팩토리

info@ifunfactory.com +82-70-4923-6566 www.ifunfactory.com