C++ Russia 2024

elbrus

Как работает профилирование

Виктор Шампаров
МЦСТ

viktor.shamparov@yandex.ru

vshamparov

Био

- Разработчик в МЦСТ
  - Универсальные оптимизации в составе компилятора
  - Некоторые профилировщики:
    - Специальное инструментирование в составе компилятора
    - Сэмплирующий со специальной поддержкой архитектуры «Эльбрус» (e2k)

Структура

- Два подхода к профилированию
- Немного про инструментирование
- Много про сэмплирование и аппаратную поддержку
  - `perf_event_open`
  - `ptrace`
  - Чуть про eBPF
  - Интерпретация instruction pointer

Два подхода к профилированию

Инструментирование

- Изменение программы
- Точный результат
- Большой оверхед

Два подхода к профилированию

Инструментирование

- Изменение программы
- Точный результат
- Большой оверхед

Сэмплирование

- Периодическое снятие характеристик программы
- Вероятностный (неточный) результат
- Небольшой оверхед

Два подхода к профилированию

Инструментирование

- Изменение программы
- Точный результат
- Большой оверхед

Сэмплирование

- Периодическое снятие характеристик программы
- Вероятностный (неточный) результат
- Небольшой оверхед

И никто не мешает их смешивать:)

Два подхода к профилированию

Как их можно смешать:

- Инструментировать код, а вместо сохранения поставить breakpoint и забирать данные 1 раз в N проходов этой точки
- Поднимать профиль для компилятора из сэмплирования:
  - Может понадобиться сохранить дополнительную информацию для упрощения интерпретации instruction pointer
- Микробенчмарки: инструментировать код вызовами подсчёта аппаратных событий

Инструментирование

Много и хорошо рассказал Павел Косов на C++Russia 2021:
«PGO: Как устроено и как использовать»

- Вставка сбора данных в код программы
- Чаще всего это делает компилятор
- Можно делать и вручную

Инструментирование PGO

Компилируем с инструментированием:

```
$ g++ -fprofile-generate <other options>
```

Выполняем на тренировочных данных.

Компилируем с PGO:

```
$ g++ -fprofile-use <same options>
```

Инструментирование PGO

Эффект PGO (e2k, компилятор LCC, `-O4 -fwhole`, SPEC CPU 2017r):

	Train данные	**Ref данные**
Min	+0%	-2%
Geomean	+20%	+18%
Max	+93%	+67%

Эффект критически зависит от качества тренировочных данных

Инструментирование PGO

Эффект PGO (e2k, компилятор LCC, `-O4 -fwhole`, отдельные задачи):

Набор	Задача	Ускорение
SPEC CPU	GCC 4.5.0	+27% (ref)
SPEC CPU	Xalan-C++, процессор языка преобразования XML	+41% (ref)
SPEC CPU	x264 (сжатие видео по стандарту H.264)	+10% (ref)
	SQLite (измерение провёл @zamazan4ik)	+28%

Инструментирование PGO

Оверхед на сбор PGO (e2k, компилятор LCC, `-O4 -fwhole`, SPEC CPU 2017r):

Min	+1%
Geomean	+60%
Max	x 5,1 раз

Инструментирование PGO

Базовый вариант PGO с инструментированием:

- Хотим счётчики всех дуг
- Ставим на минимальный набор дуг инкременты счётчиков
- На старте функции регистрируем массив счётчиков, если надо
- В библиотеке поддержки на выходе из приложения записываем файл с профилем

Инструментирование

Специальные виды инструментирования в компиляторе:

- Сбор покрытия
- Value profile
- Path profile
- Санитайзеры
- Пользовательское инструментирование
  - `gcc -finstrument-functions ...`

Инструментирование

Инструментировать можно и вручную:

- Дебаговая печать:)
- Самописные системы для сбора данных о программе
- Встроенные функции компилятора
  - Например, для GCC предложено в патенте **US 11,321,061 B2** для сбора пользовательского value profile.

Сэмплирование

Периодически снимаем интересующие нас данные во время исполнения программы.

- Период может измеряться в определённых событиях
- Чаще всего нас интересует IP (instruction pointer)
  - И иногда backtrace
- Результаты вероятностные
- Оверхед может быть намного меньше

Tracing

Снимаем данные по определённым событиям

- Сливается до неразличимости с сэмплированием по событиям с периодом 1 (то есть по каждому событию)

Поэтому далее будем рассматривать ещё и tracing, когда это возможно

Сэмплирование

Примеры сэмплирующих профилировщиков:

- Системный вызов `perf_event_open`
  - Perf
  - Android Simpleperf
- EBPF
- Strace
- Intel VTune

Сэмплирование

Рассмотрим два варианта обеспечения сэмплирования, оба доступны в Linux:

- `perf_event_open`
- `ptrace`

Perf для пользователя

Измерить количество тактов (e2k) на исполнение программы:

```
$ perf stat -e ticks ./a.out
```

Получить трассу исполнения программы — IP на 1/1 000 000 событий:

```
$ perf record -e ticks -c 1000000 ./a.out
```

Отчёт по трассе:

```
$ perf report
```

perf_event_open

```
PERF_EVENT_OPEN(2)                Linux Programmer's Manual                PERF_EVENT_OPEN(2)

NAME
       perf_event_open - set up performance monitoring

SYNOPSIS
       #include <linux/perf_event.h>
       #include <linux/hw_breakpoint.h>

       int perf_event_open(struct perf_event_attr *attr,
                           pid_t pid, int cpu, int group_fd,
                           unsigned long flags);

       Note: There is no glibc wrapper for this system call; see NOTES.

DESCRIPTION
       Given a list of parameters, perf_event_open() returns a file descriptor, for use in subsequent system
       calls (read(2), mmap(2), prctl(2), fcntl(2), etc.).

       A call to perf_event_open() creates a file descriptor that allows measuring performance  information.
       Each file descriptor corresponds to one event that is measured; these can be grouped together to mea-
       sure multiple events simultaneously.

       Events can be enabled and disabled in two ways: via ioctl(2) and via prctl(2).  When an event is dis-
       abled  it  does  not  count  or  generate overflows but does continue to exist and maintain its count
       value.

       Events come in two flavors: counting and sampled.  A counting event is one that is used for  counting
 Manual page perf_event_open(2) line 1 (press h for help or q to quit)
```

perf_event_open

1) Заполняем структуру `perf_event_attr`
2) Делаем `perf_event_open`, он возвращает файловый дескриптор
3) Выделяем с помощью `mmap` память под буфер с данными от профилировщика
4) Ждём по файловому дескриптору сигнала о готовности сэмпла
5) Читаем и обрабатываем сэмпл в памяти под `mmap`.

perf_event_open

Настройки в `perf_event_attr`:

1) Тип события: PERF_TYPE_HARDWARE / PERF_TYPE_RAW / PERF_TYPE_SOFTWARE / etc.
2) Событие
3) Период или частота сброса сэмплов
4) Данные в сэмпле: IP, TID, адрес данных, backtrace...
5) Прочее: флаги генерации сэмплов на каждый `mmap`, `exec`, собирать ли события ядра и т.п.

perf_event_open

Пример задания структуры perf_event_attr:

```
perf_event_attr
init_sampling( std::uint64_t period ) {
    perf_event_attr attr;
    /* Заполняем свойства профилирования. */
    attr.type          = PERF_TYPE_HARDWARE;
    attr.sample_period = period;
    attr.sample_type   = PERF_SAMPLE_IP;
    attr.config        = PERF_COUNT_HW_CPU_CYCLES;
    /* Заставляем дёргать каждый раз, когда есть данные в трассе. */
    attr.wakeup_watermark = 1;
    attr.watermark        = 1;
    return attr;
}
```

perf_event_open

Пример для perf_event_open и mmap:

```
perf_event_attr attr = init_sampling( period );
int fd = syscall( __NR_perf_event_open, &attr, tid, -1, -1, 0 );

/* Размер для mmap = 1 + 2^n страниц (n задаётся программистом):
 * первая страница с метаданными, остальное
 * под циклический буфер сэмплов */
int header_buffer_size_pages = 1 + ( 1 << n );
int header_buffer_size = header_buffer_size_pages * PAGE_SIZE;

perf_event_mmap_page *header_p = mmap( NULL,
                                       header_buffer_size,
                                       PROT_READ|PROT_WRITE,
                                       MAP_SHARED,
                                       fd, 0 );
```

perf_event_open

Пример ожидания и чтения сэмпла:

```
pollfd pollfds[1]; pollfds[0].fd = fd; pollfds[0].events = POLLIN;
int poll_ret = poll( pollfds, 1, timeout ); // Ожидаем
if ( pollfds[0].revents & POLLIN ) {
    __sync_synchronize();
    char *sample_buffer_p = (char*)header_p + header_p->data_offset;
    perf_event_header *event_header_p = sample_buffer_p
        + ( header_p->data_tail % header_p->data_size );
    /* Наконец читаем сэмпл на основе его заголовка */
    switch ( event_header_p->type ) {
        case PERF_RECORD_SAMPLE: …
        ...
    }
    __sync_synchronize();
    header_p->data_tail = header_p->data_head;
}
```

perf_event_open: под капотом

- Использует PMU (Performance Measurement Unit)
  - Очевидно, аппаратно-зависим
- Работает в контексте профилируемого процесса
- При возникновении сэмпла тратит время самого процесса на получение данных и позволяет профилировщику забрать сэмпл
  - Данные могут теряться — рассмотрим это позже

PMU

- Примеры:
  - Intel PMU, LBR (Last Branch Records)
  - ARM PMU (иногда с LBR)
  - PMU в архитектуре e2k
- Обычно позволяют:
  - Собирать счётчики аппаратных событий
  - Вызывать перехватываемые ОС исключения на аппаратные события с периодом N

Особые режимы PMU или `perf_event_open`

- Last Branch Records в PMU — позволяет собирать аналоги трасс исполнения
  - Есть в Intel, некоторых ARM, в Эльбрусах новых версий
- Буферизация сэмплов в `perf_event_open`
- Аппаратная буферизация в PMU

Оверхед `perf_event_open`

- Сбор сэмпла ненадолго ставит саму программу на паузу
- Может набирать большие файлы с трассами
- Пример:
  - Программа `a.out` на 240 млрд тактов
  - `perf record -e ticks -c 10000 ./a.out`
  - Замедлилась на 25%
  - Собрала трассу примерно на 400 Мб

Потери `perf_event_open`

`perf_event_open` организует сэмплы в циклический буфер и не ждёт, пока профилировщик заберёт данные
=> сэмплы могут теряться

Решения:

- Буферизация передачи сэмплов от `perf_event_open`
- Аппаратная буферизация в PMU

ptrace

Системный вызов для трассирования одного процесса другим.

- Соединяем два процесса по `PTRACE_TRACEME` или `PTRACE_ATTACH`
- Задаём в процессе-*tracee* нужные регистры PMU и ждём
- При возникновении исключения, связанного с PMU, ОС останавливает процесс-*tracee* сигналом SIGPROF и сообщает об этом процессу-*tracer*'у
- Добываем значения нужных регистров и отправляем процессу-*tracee* сигнал продолжить работу

ptrace

Пример старта профилируемой программы с PTRACE_TRACEME:

```
pid_t child_pid = fork();
switch ( child_pid )
{
     case -1: /* сломался fork */
     case 0:
          /* процесс-ребёнок */
          ptrace( PTRACE_TRACEME, 0, 0, 0);
          execv( ... );
     default:
          /* процесс-родитель */
          ...
}
```

ptrace

Пример получения и задания регистров:

```
user_regs_struct *regs;
/* Получаем значения в регистрах */
ret = ptrace( PTRACE_GETREGS, pid, 0, regs );

/* Читаем или записываем в регистры что нам надо */
std::printf( «rax = 0x%016lx\n», regs->rax );

/* Записываем значения в регистры программы */
ret = ptrace( PTRACE_SETREGS, pid, 0, regs );
```

ptrace

Пример обработки событий программы:

```
int status;
pid = waitpid( -1, &status, WUNTRACED );
if ( WIFEXITED( status ) || WIFSIGNALED( status ) ) ...
if ( !WIFSTOPPED( status ) ) ...

// Здесь известно, что процесс остановлен
int sig = WSTOPSIG(status);
switch( sig ) {
    /* Обрабатываем, каким сигналом остановлен процесс */
}
ptrace( PTRACE_CONT, pid, 1, sig );
```

ptrace

Пример обработки событий программы:

```
sig = WSTOPSIG( status );
switch( sig ) {
    case SIGSTOP:
    /* получаем SIGSTOP, когда появляется новый процесс */
    case SIGTRAP:
    {   int event = status >> 16;
        if ( event == PTRACE_EVENT_CLONE )
        /* Так приходят события о fork, clone, mmap... */
    }
    case SIGPROF:
    /* сэмпл от PMU */
}
```

Оверхед `ptrace`

В отличие от `perf_event_open`, `ptrace` тормозит профилируемую программу для получения данных профилировщиком

Оверхед побольше, тем более что нет буферизации. Пример:

- Та же программа
- Такой же период сброса сэмплов
- Замедлилась на 60%
- Собрала трассу примерно на 500 Мб

eBPF

Специальное средство ядра Linux для исполнения программ, в частности для мониторинга и профилирования

- Использует собственный байткод и выполняется в «песочнице»
- Есть много утилит на базе eBPF, см.:
  - BCC: https://github.com/iovisor/bcc
  - BpfTrace: https://github.com/bpftrace/bpftrace

Интерпретация IP

Так ли просто понять, к какой функции относится instruction pointer?

Интерпретация IP

Так ли просто понять, к какой функции относится instruction pointer? Хотя бы к функции…

Интерпретация IP

Так ли просто понять, к какой функции относится instruction pointer? Хотя бы к функции…

- Для самого приложения — да, если нет специальных режимов исполнения
- Например, «защищённого режима» в e2k
- Но динамические библиотеки попадают на случайные адреса — это защита от хакеров => нет

Интерпретация IP

Решение — отслеживать `mmap`:

- Загрузка динамической библиотеки выполняется через `mmap`
- И `perf_event_open` и `ptrace` это позволяют
  - У `perf_event_open` это специальный вид сэмпла, который можно включить
  - У `ptrace` это специальное событие
- На старте программы надо получить уже загруженные библиотеки либо отследить процесс загрузки

Интерпретация IP

Из-за inline получается, что IP в одной функции может быть в вызываемой — сложно разбирать. Неполный список доступных действий:

- Разбирать DWARF — во многих случаях дебаговая информация позволит достаточно точно получить место в исходном коде
- Разбирать по дампам компилятора

Что почитать и посмотреть

- Доклад Павла Косова на C++Russia 2021: *«PGO: Как устроено и как использовать»*
- Отличный сайт с большим количеством примеров perf и eBPF: https://www.brendangregg.com/overview.html
- Книга Дениса Бахвалова *«Performance Analysis and Tuning on Modern CPUs»*
- Репозиторий с большим количеством успешных примеров применения PGO: https://github.com/zamazan4ik/awesome-pgo

Вопросы?