Perf for User Space Program Analysis

30 May, 2013

Tetsuo Takata,
Platform Solutions Business Unit, System Platforms Sector
NTT DATA CORPORATION

NTT DATA

Agenda

1. Introduction
2. What is perf?
3. Usage and usecases
4. A Use Case -- profiling a user space program
5. Conclusion and future work

Introduction

Background and goal

NTT DATA

Background:
OSS has been used in many mission-critical systems, where every single problem must be fixed fast and accounted for, and where tools assisting troubleshooting is more important than anywhere else.

Perf is becoming a de facto standard of performance analysis tools for Linux among many others. We think that perf is a very capable tool with very scarce documentation. Therefore,

Goal:

we are going to share our experience with perf with other user space developers and support engineers, by presenting

- **information necessary to make good use of perf without knowing much about the kernel, and**
- **a use case of perf where we analyzed a performance problem of a middleware.**

Troubleshooting Enterprise Systems

It follows the general troubleshooting process of diagnosis and solution, with some restrictions, mostly on data collection, to keep systems up and running.

Trouble!

Diagnosis

Problem definition	Elimination	Identification of root cause
Understand the problem through information gathered from operators, messages, etc.	Eliminate possible causes by collecting and examining information and data.	Identify where and how the problem occurred.

Solution

First aid	Solution of problem
Provide temporary measure(s) to avoid the problem and keep the system running.	Implement measure(s) to prevent the problem.

Most time-consuming part in majority of cases

The Diagnosis Phase

- What is it?
  - To eliminate candidate causes of a problem by collecting and examining information and data about the system under question.
- Why do we do that?
  - To implement right measure(s) to prevent the problem from recurring, and to avoid wasting bullets as in a local saying—even a poor shooter can hit the mark with many bullets.
- Techniques include
  - Overall analysis
    - message analysis
    - system statistics analysis
    - ...
  - Detailed analysis
    - tracing
    - profiling
    - probing
    - core dump analysis
    - ...

Profiling as a Diagnosis Tool

NTT DATA

- Profiling comes into play when there is a performance problem and responsible piece of software is known, and used to measure how much CPU time is spent where to narrow down the number of suspects.
- A profiler differs from a tracer, another performance analysis tool, in that the former gather samples at fixed intervals while the latter collects timestamps at specified places. A profiler, therefore, incurs less overhead while a tracer can obtain accurate timing information.
- There are several profiler implementations currently available for Linux.
  - perf: implemented in the kernel, actively developed.
  - oprofile: implemented in the kernel.
  - gprof: implemented in the user space, requiring a specific compile-time option.
  - sysprof: implemented in the kernel, does overall system profiling.

Profiler for Enterprise Troubleshooting

NTT DATA

A profiler has to satisfy the following to be used in enterprise settings:

- Little overhead
  - Additional overhead to systems under investigation that are usually under heavy load from performance problem(s) can lead to malfunctioning of the profiler, or worse, bring the systems down.
  - Controllable and Overhead, if any, must be under control and predictable.
- No additional installation
  - Any changes to a tested software configuration is not acceptable, without testing.
- Wealth of information gathered
  - There may not be a second or third chances.
- Presentation of information
  - We have to be able to drill into plentiful information.

What is perf?

What is perf?

NTT DATA

- perf(Performance Counters for Linux) is ...
  - an integrated performance analysis tool on Linux kernel
  - basically a profiler, but its tracer capabilities has been enhanced and becomes an all-round performance analysis tool.

- Events
  - for profiling
    - hardwre event
    - swevent
    - ...
  - for tracing
    - trace point
    - probe point

- Samples
  - Events related information
  - IP
  - CALLCHAIN
  - STACK
  - TIME
  - ...

perf events

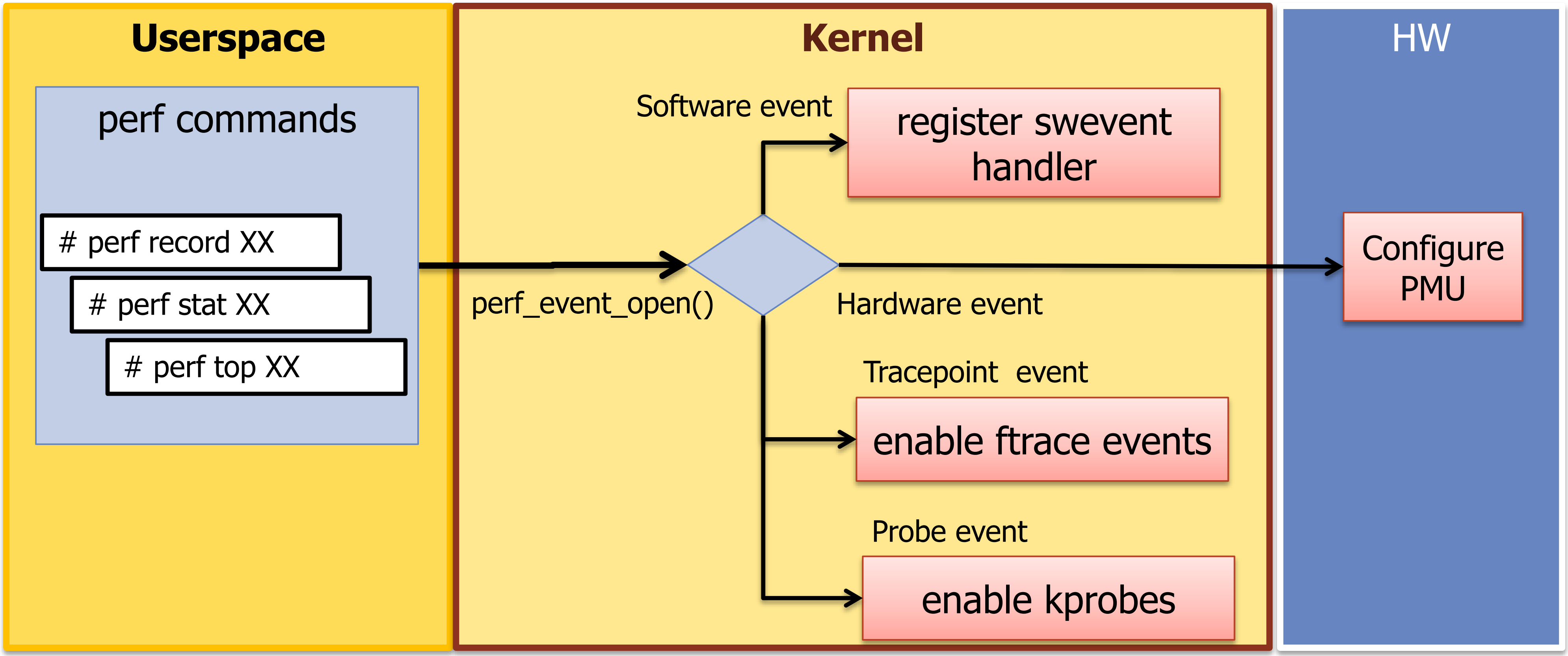
NTT DATA

Categories	Descriptions	Examples
Hardware events	Event measurable by PMU of processor. Data can be collected without the overhead though the contents is dependent on type of processors.	Cpu-cycles and cache-misses, etc.
Hardware cache events		L1-dcache-load-misses and branch-loads, etc.
Software events	Event measurable by kernel counter.s	Cpu-clock and page-faults, etc.
Tracepoint events	Code locations built into the kernel where trace information can be collected.	Sched:sched_stat_runtime and syscalls:sys_enter_socket, etc.
Probe events	User-defined events dynamically inserted into the kernel.	-

perf event registration

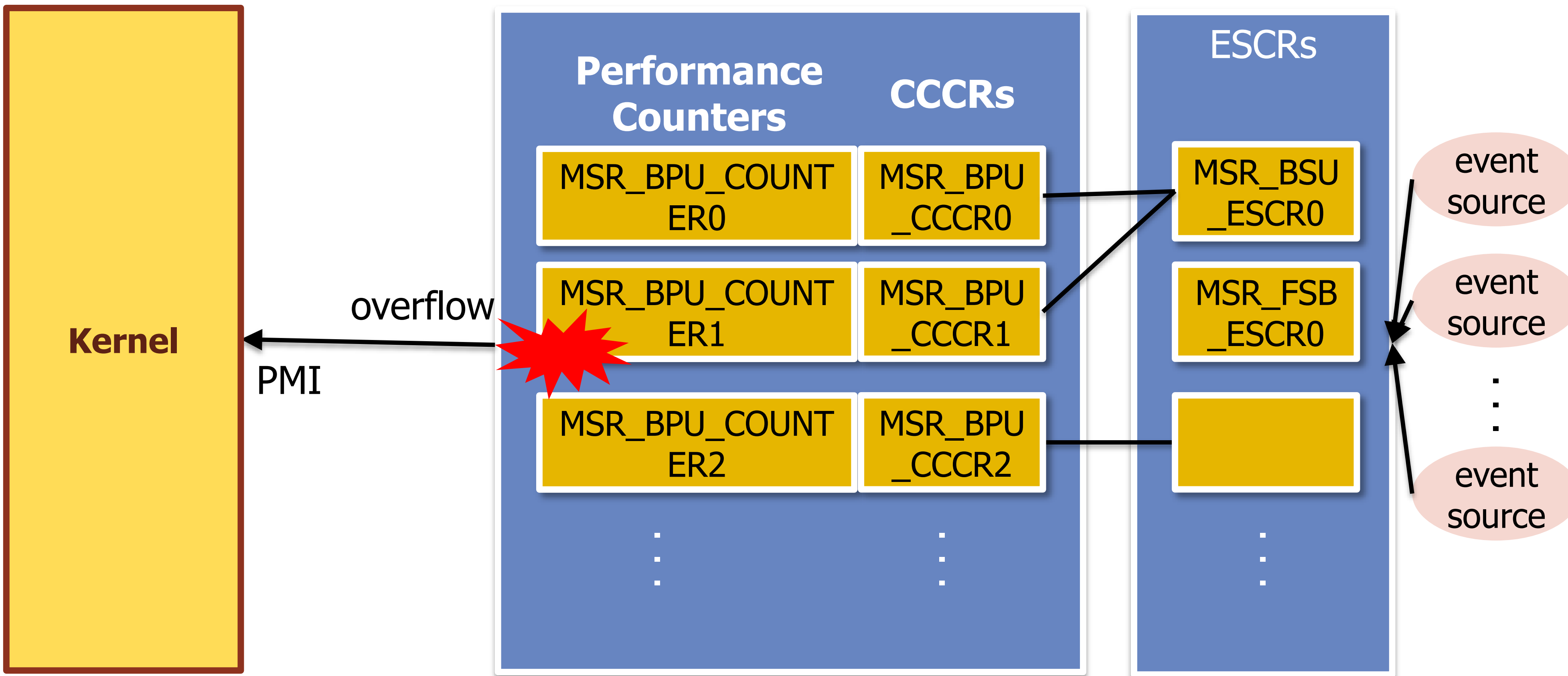
NTT DATA

perf commands register events by calling perf_event_open() system call, which, in turn, registers them in hardware or software according to their types.



Hardware Event Overview(Intel)

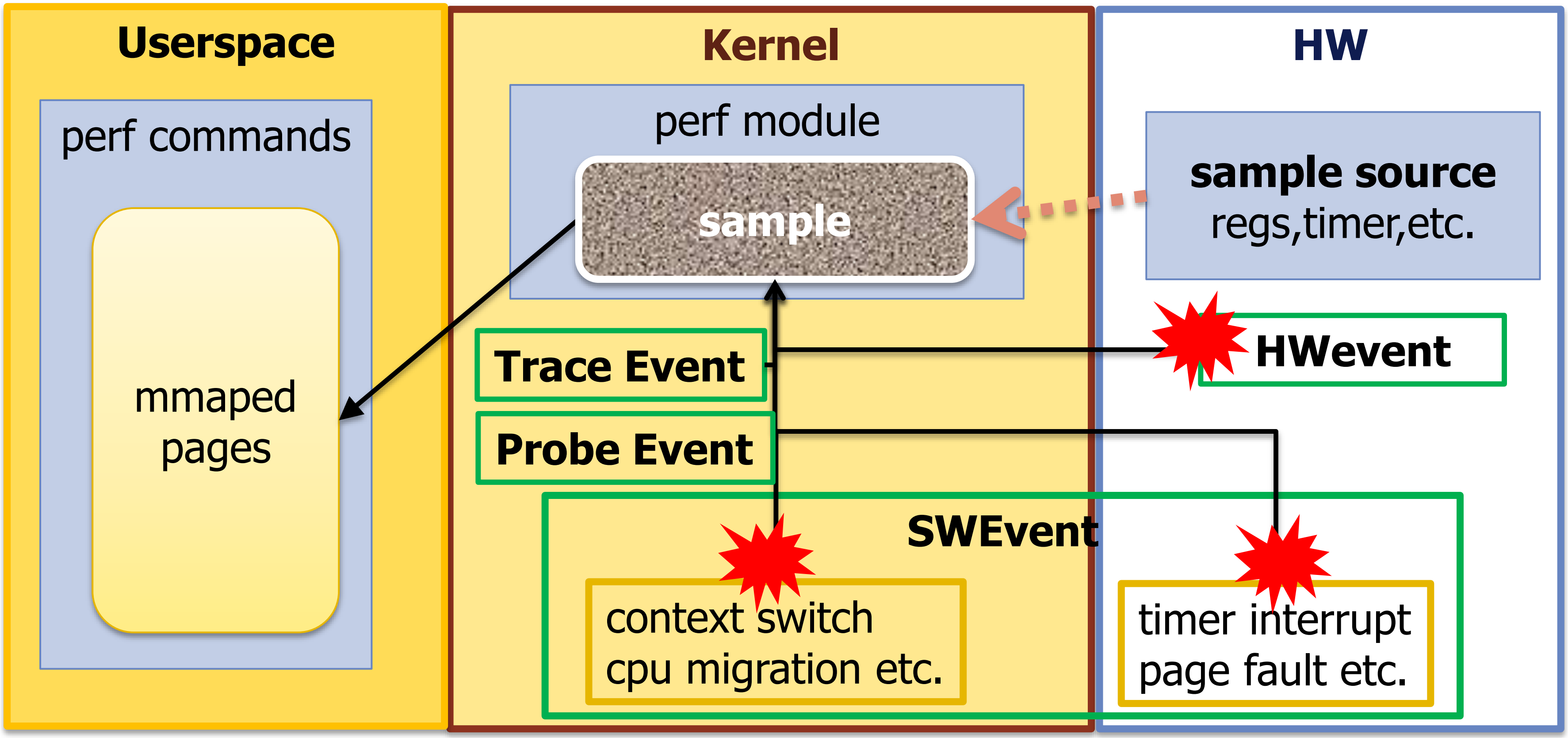
Collection of sample data on hardware events are mostly done by hardware. A pair of Performance counter and CCCR records data at events selected by an ESCR. Only when a Performance Counter overflows when the kernel receives a PMI interrupt and copies information from the registers.



Event handling and sampling

NTT DATA

The perf module collects samples when an event like HWevent occurs. Data to be collected is specified as sample types when a user invokes the perf command. They include IP (Instruction Pointer), user or kernel stack, timer and mostly taken from hardware. Samples collected are written to memory area mapped by the perf command so that it can retrieve them without kernel-to-user copying.



Usage and a use case

Usage of perf

▪Usage of perf command

```
# perf <command> [option]
```

Commands	Descriptions
annotate	Read perf.data* and display annotated code
diff	Read two perf.data* files and display the differential profile
probe	Define new dynamic trace-points
record	Run a command and record its profile into perf.data*
report	Read perf.data* and display the profile
script	Read perf.data* and display trace output
stat	Run a command and gather performance counter statistics
timechart	Tool to visualize total system behavior during a workload
top	Generate and displays a performance counter profile
trace	Show the events associated with syscalls, etc

Usage of perf (cont'd)

▪List of perf commands

perf.data* is created by **perf record**

Commands	Descriptions
archive	Create archive with object files with build-ids
bench	General framework for benchmark suites
buildid-cache	Manage build-id cache
buildid-list	List the build-ids in a perf.data* file
evlist	Displays the names of events sampled in a perf.data* file
inject	Filter to augment the events stream
kmem	Tool to trace/measure kernel memory properties
kvm	Tool to trace/measure KVM guest OS
list	List all symbolic event types
lock	Analyze lock events
sched	Tool to trace/measure scheduler properties (latencies)
test	Runs sanity tests

perf record & perf report

perf record records events. Recorded data is saved as perf.data by default. We can confirm this data with the **perf report**.

Use cases

- Record behavior of a specific command in detail
- Analyze a suspicious process in detail
- Determine a cause(s) of poor performance of a process

perf record - options -

Options	Descriptions
-e	Designate an event name
-o	Designate a output filename (perf.data by default)
-p	Designate a process ID
-t	Designate a thread ID
-a	Collect data from all of the processors
-C	Designate a core(s) from which the command collect data

perf record - example -

```
# perf record stress --cpu 4 --io 2 --vm 2  --timeout 10s
stress: info: [3765] dispatching hogs: 4 cpu, 2 io, 2 vm, 0 hdd
stress: info: [3765] successful run completed in 10s
[ perf record: Woken up 5 times to write data ]
[ perf record: Captured and wrote 1.008 MB perf.data (~44044 samples) ]

# ls
perf.data
```

perf report - options -

Options	Descriptions
-i	Designate a input file (perf.data by default when it is not designated)
-s	Sort data by given key such as pid

perf report - example -

```
# perf report | cat -n
…
22  # Overhead  Command      Shared Object                                      Symbol
    23  # ........  .......  .................  ...........................
    24  #
    25        35.40%   stress  libc-2.12.so        [.] __random_r
    26        15.25%   stress  libc-2.12.so        [.] __random
    27        14.70%   stress  stress              [.] 0x0000000000001bd1
    28         7.07%   stress  [kernel.kallsyms]   [k] acpi_pm_read
    29         5.16%   stress  [kernel.kallsyms]   [k] _spin_unlock_irqrestore
    30         5.06%   stress  [kernel.kallsyms]   [k] ioread32
    31         4.95%   stress  [kernel.kallsyms]   [k] finish_task_switch
    32         4.85%   stress  libc-2.12.so        [.] rand
    33         2.11%   stress  [kernel.kallsyms]   [k] sync_inodes_sb
    34         1.22%   stress  [kernel.kallsyms]   [k] iowrite32
    35         0.69%   stress  [kernel.kallsyms]   [k] clear_page_c
    36         0.33%   stress  [ahci]              [k] ahci_interrupt
    37         0.24%   stress  [kernel.kallsyms]   [k] __do_softirq
    38         0.14%   stress  [kernel.kallsyms]   [k] compact_zone
    39         0.11%   stress  [kernel.kallsyms]   [k] copy_page_c
```

perf report - example -

```
Samples: 26K of event 'cpu-clock', Event count (approx.): 6556250025
 35.40%  stress  libc-2.12.so        [.] __random_r
 15.25%  stress  libc-2.12.so        [.] __random
 14.70%  stress  stress              [.] 0x0000000000001bd1
  7.07%  stress  [kernel.kallsyms]   [k] acpi_pm_read
  5.16%  stress  [kernel.kallsyms]   [k] _spin_unlock_irqrestore
  5.06%  stress  [kernel.kallsyms]   [k] ioread32
  4.95%  stress  [kernel.kallsyms]   [k] finish_task_switch
  4.85%  stress  libc-2.12.so        [.] rand
  2.11%  stress  [kernel.kallsyms]   [k] sync_inodes_sb
  1.22%  stress  [kernel.kallsyms]   [k] iowrite32
  0.69%  stress  [kernel.kallsyms]   [k] clear_page_c
  0.33%  stress  [ahci]              [k] ahci_interrupt
  0.24%  stress  [kernel.kallsyms]   [k] __do_softirq
```

perf top

perf top can profile a system in real time. Just like the **top** command in Linux, it can dynamically conduct a system monitoring

Use cases

- Conduct a whole system profiling
- Conduct a system monitoring in real time

perf top - example -

NTT DATA

```
Samples: 26K of event 'cpu-clock', Event count (approx.): 21912
 24.82%  32bc-2.12.so          [.] __random_r            18032
 13.95%  [kernel]              [k] sync_inodes_sb
 16.94%  stress                [.] 0x0000000000000d18
 10.96%  libc-2.12.so          [.] __random
  6.67%  stress                [k] 0x0000000000000d18store
  7.35%  [kernel]              [k] _spin_unlock_irqrestore
  7.29%  [kernel]              [k] acpi_pm_read
  5.54%  libc-2.12.so          [.] rand
  2.41%  [kernel]              [k] finish_task_switch
  3.02%  [kernel]              [k] iowrite32
  0.72%  [kernel]              [k] wait_on_page_writeback_range
  0.47%  [kernel]              [k] sync_filesystems
  0.52%  [kernel]              [k] _spin_lock
  0.37%  [kernel]              [k] find_get_pages_tag
  0.23%  [ahci]                [k] ahci_interrupt
  0.32%  [kernel]              [k] find_get_pages_tag
  0.27%  [ahci]                [k] ahci_interrupt
```

perf anotate

perf anotate reads perf.data and display annotated decompiled code.

Use case

- Identify time-consuming part(s) in source code

perf annotate - options -

Options	Descriptions
-i	Specify input file name (perf.date by default)
-s	Set symbol to annotate
-v	Display result more verbosely
-l	Print matching source lines
-P	Make displayed pathnames as full-path
-k	Specify the vmlinux path

perf annotate - examples -

perf record ./a.out
perf annotate –s main

```
main
              Disassembly of section .text:

              0000000000400474 <main>:
              int main(void){
                push   %rbp
                mov    %rsp,%rbp
                int i;
                for(i=0; i<1000000000; i++){}
                movl   $0x0,-0x4(%rbp)
              ↓ jmp    11
14.78     d: ┌→addl   $0x1,-0x4(%rbp)
 3.78    11: │ cmpl   $0x3b9ac9ff,-0x4(%rbp)
81.45        └─jle    d
                return 0;
                mov    $0x0,%eax
              }
                leaveq
              ← retq
```

```
int main(void){
        int i;
        for( i=0; i<1000000000; i++ );
        return 0;}
```

perf diff

perf diff reads two perf.data files and display the differential profile.

Use case

- See differences between updated perf.data and older one.

perf diff - options -

Options	Descriptions
-S	Specify only consider symbols
-s	Sort by **key**(s): PID, comm, dso, symbol

perf diff - example -

NTT DATA

 # perf record -o perf1.data ls /boot
 # perf record -o perf2.data ls /etc/
 # perf diff perf1.data perf2.data

```
# Event 'cpu-clock'
#
# Baseline    Delta      Shared Object                          Symbol
# ........  .......   .................   ..............................
#
             +20.00%   ld-2.12.so          [.] dl_main
             +20.00%   ld-2.12.so          [.] do_lookup_x
             +20.00%   [kernel.kallsyms]   [k] up_read
             +20.00%   [kernel.kallsyms]   [k] __find_get_block
             +20.00%   [kernel.kallsyms]   [k] copy_from_user
    50.00%   -50.00%   [kernel.kallsyms]   [k] unmap_vmas
    50.00%   -50.00%   [kernel.kallsyms]   [k] mem_cgroup_charge_common
```

A Use Case -- profiling a userspace program

Profiling PostgreSQL -- bakground

Background

Most developers and support engineers of middleware feel uncomfortable with hardware and kernel level information returned by kernel profilers, and think rather in terms of functions or API implemented by themselves. For a profiler to be useful for them, it should be able to present information specific to an application while abstracting lower-level details away if possible.

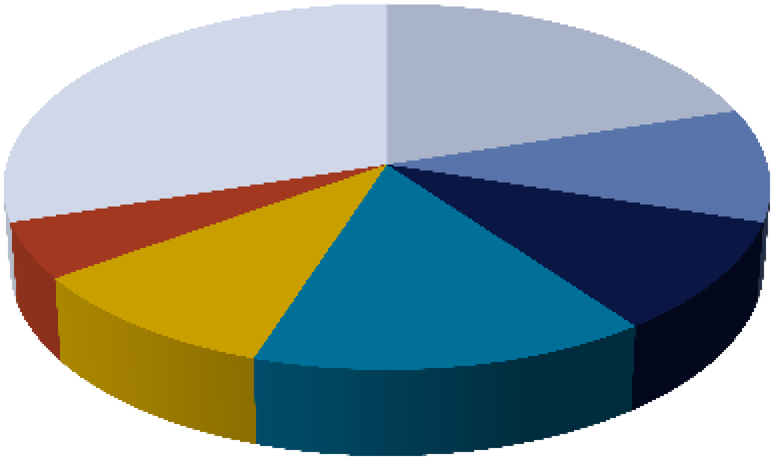
Userland profiling and PostgreSQL

A large userland program like a database management system would frequently benefit from a built-in tracing facility to find, for example, where performance regression comes from. Its implementation is not, however, always welcome by developer community because of maintenance burden, e.g. failed attempt at tracer in PostgreSQL (*).

*) http://www.postgresql.org/message-id/20090309125146.913C.52131E4D@oss.ntt.co.jp
http://www.postgresql.org/message-id/20090714183127.946A.52131E4D@oss.ntt.co.jp

The failed proposal tried to produce output like the figure. It is commonplace in commercial DBMSes and highly desirable for OSS ones like PostgreSQL as well. Our use case does not replicate it visually, but it still shows that perf can be used to do similar type of analysis.

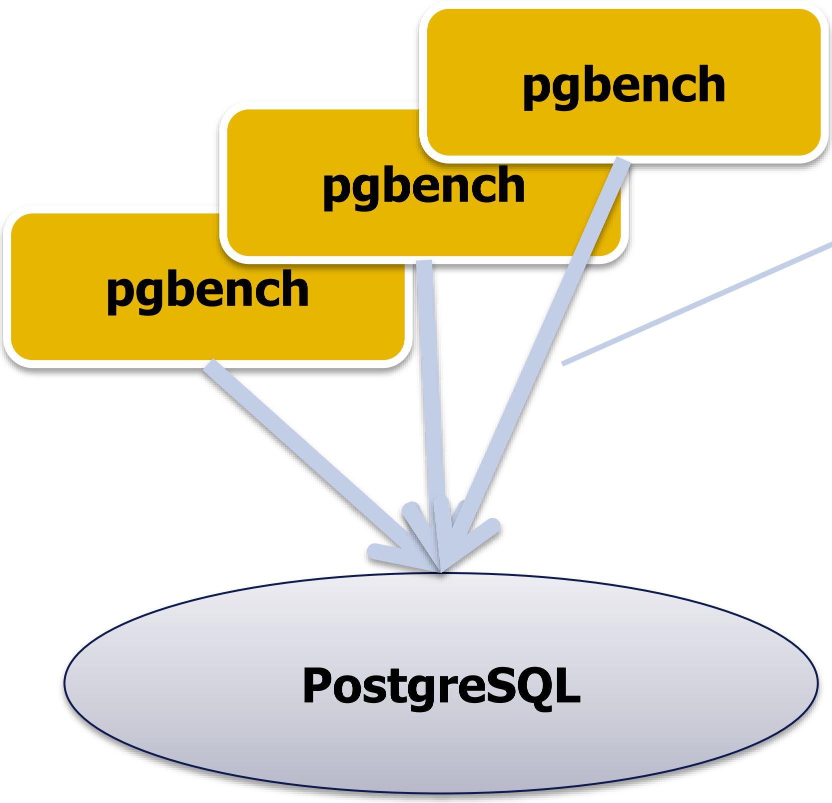
Performance Usage Graph (Image)



Types	Descriptions (typical activity)
CPU	Query parse/planning, Read/Write data from/to shared memory, Data sort on local memory
NETWORK	Receive Query from client, Send search results to client
IDLE	Just idle, sleep
XLOG	TXN file open/close, Write and Flush data to TXN file
DATA	File open/close, Read/Write from/to file via system call
LOCK	Acquire/Release/Waite rows/table level lock
LWLOCK	Acquire/Release/Waite light-weight lock for shared data

NTT DATA

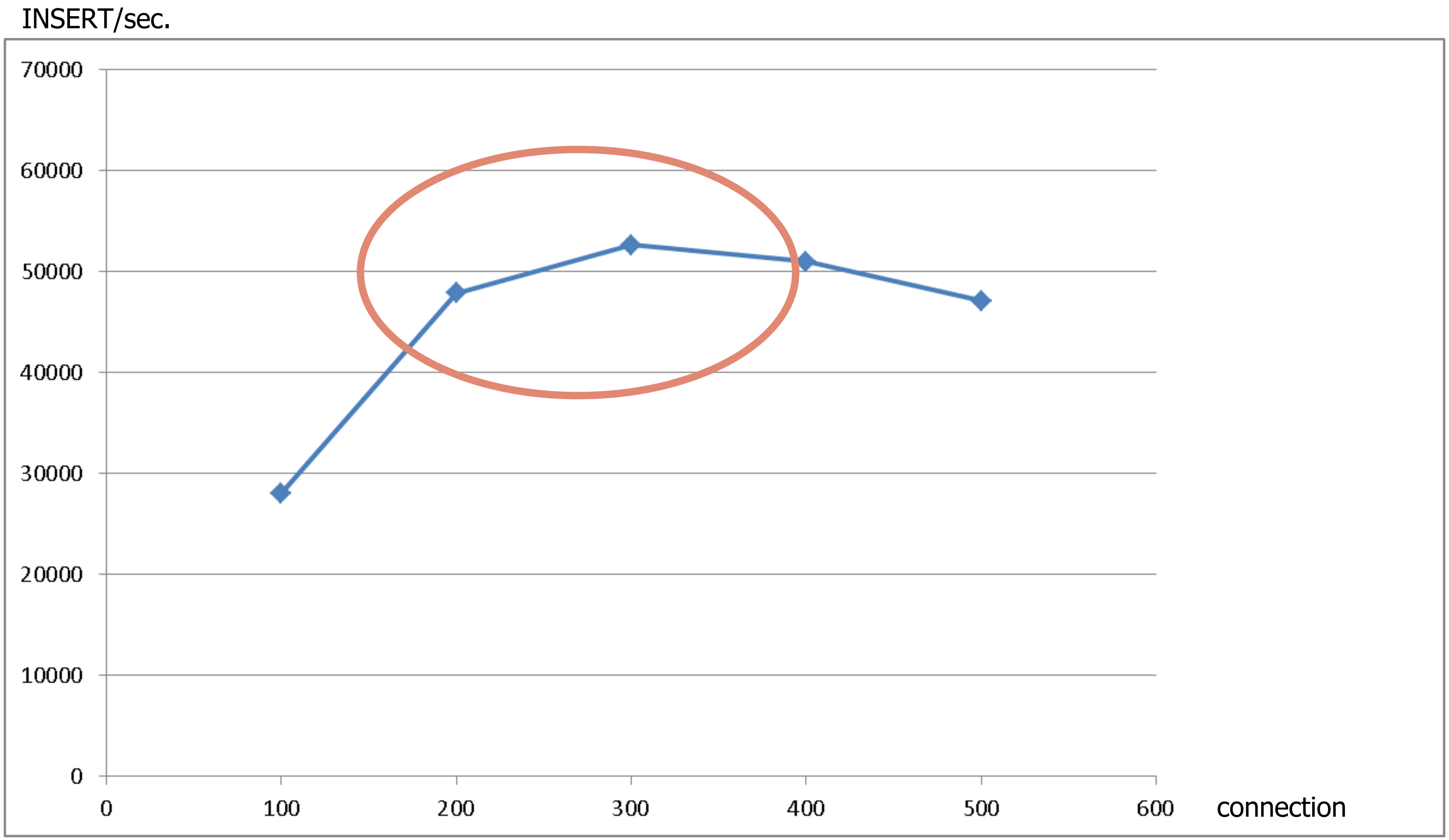
We ran a benchmark program to issue large amount of insert statements against PostgreSQL at the same time. It is expected to cause lock contention, which is hard to analyze without the help of a profiler like perf.




```
BEGIN;
INSERT INTO i_table VALUES (:user_id,  'c4ca4238a0b923820dcc509a6f75849b');
INSERT INTO i_table VALUES (:user_id,  'c81e728d9d4c2f636f067f89cc14862c');
INSERT INTO i_table VALUES (:user_id,  'eccbc87e4b5ce2fe28308fd9f2a7baf3');
INSERT INTO i_table VALUES (:user_id,  'a87ff679a2f3e71d9181a67b7542122c');
INSERT INTO i_table VALUES (:user_id,  'e4da3b7fbbce2345d7772b0674a318d5');
END;
```

- Environment(Server)
    - CPU: Intel(R) Xeon(R) CPU E5-2660 2.20GHz * 4 CPUs (32 Logical Cores)
    - Memory: 24GB
    - kernel: 3.9.2
    - PostgreSQL: 9.2.44

Number of inserts per second stopped grow linearly in 100--200 connections, suggesting existence of performance neck(s), and it went down with more than 300 connections, a tendency frequently observed when a lock is contented.



Analysis using perf call graph function

```
17.03%  postgres  postgres            [.] s_lock
        |
        --- s_lock
           |
           |--79.95%-- LWLockAcquire
           |          |
           |          |--56.91%-- GetSnapshotData
           |          |          GetTransactionSnapshot
           |          |          |
           |          |          |--94.56%-- exec_simple_
           |          |          |          PostgresMain
           |          |          |          ServerLoop
           |          |          |          PostmasterMain
           |          |          |          main
           |          |          |          __libc_start_main
           |          |          |          _start
```

The "--callgraph dwarf" option turns on the use of DWARF, the standard debugging information format on Linux.

The option enables sampling of user as well as kernel stack information and generation of callgraphs containing symbols in user programs.

The DWARF mode should be used with caution because the amount of data collected is far from a dwarf and an order of magnitude larger than the default mode.

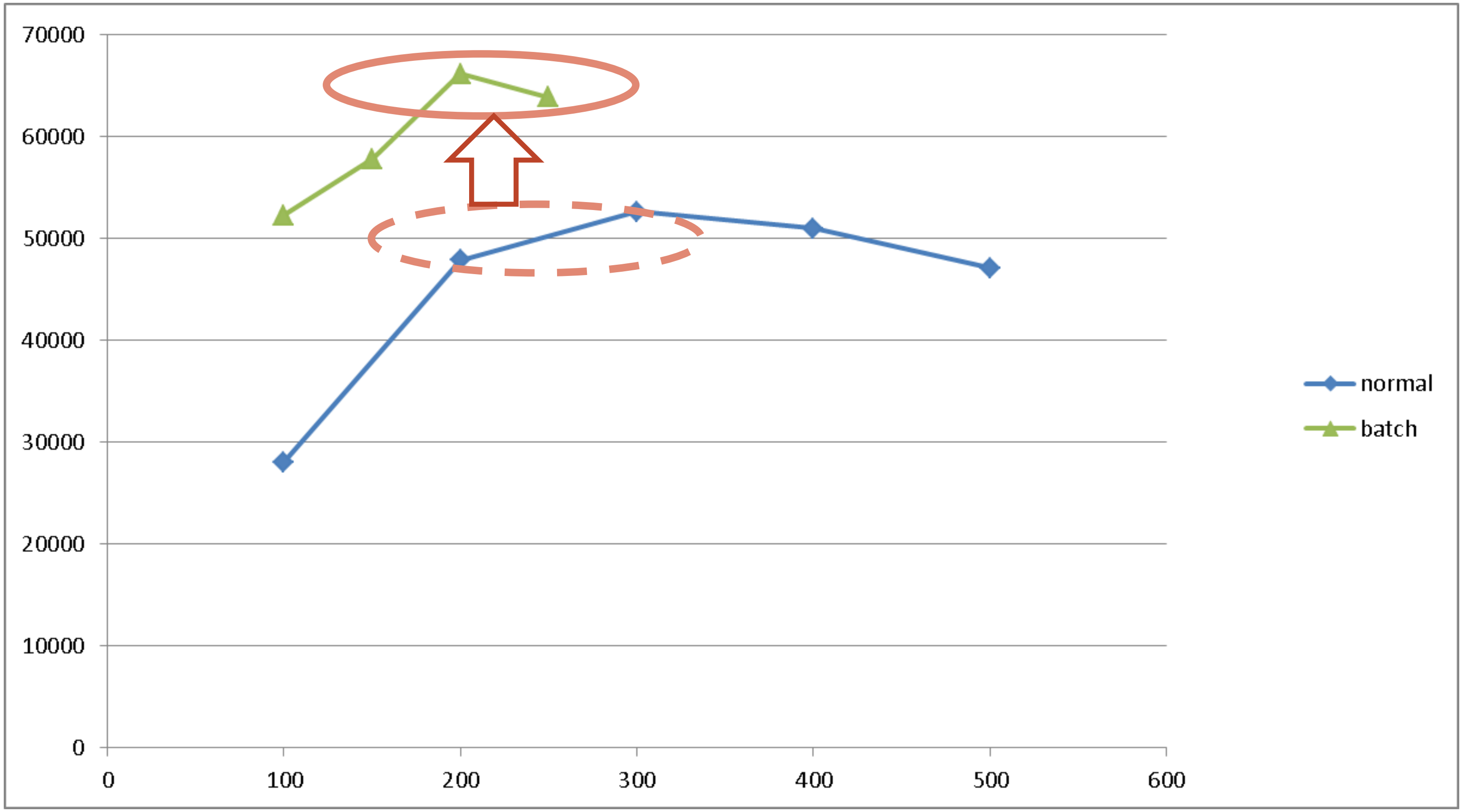
NTT DATA

```
Snapshot
GetSnapshotData(Snapshot snapshot)
{
...
 LWLockAcquire(ProcArrayLock, LW_SHARED);
...
numProcs = arrayP->numProcs;
    for (index = 0; index < numProcs; index++)
    {
...
    }
```

The culprit was a function where PostgreSQL found the oldest transaction from the list of active transaction id's. As it searched through an array containing entries for all prcesses, it took time propotional to number of process (O(N)). If there were more processes, it was more likely to cause lock contention.

As the benchmark program committed insert statements in batches, the lock is expected to be contended less often if the size of the batch was made larger, from 5 to 10 in our case.

The larger batch size successfully raised the maximum throughput, proving the analysis!!



Lock contention before and after the solution

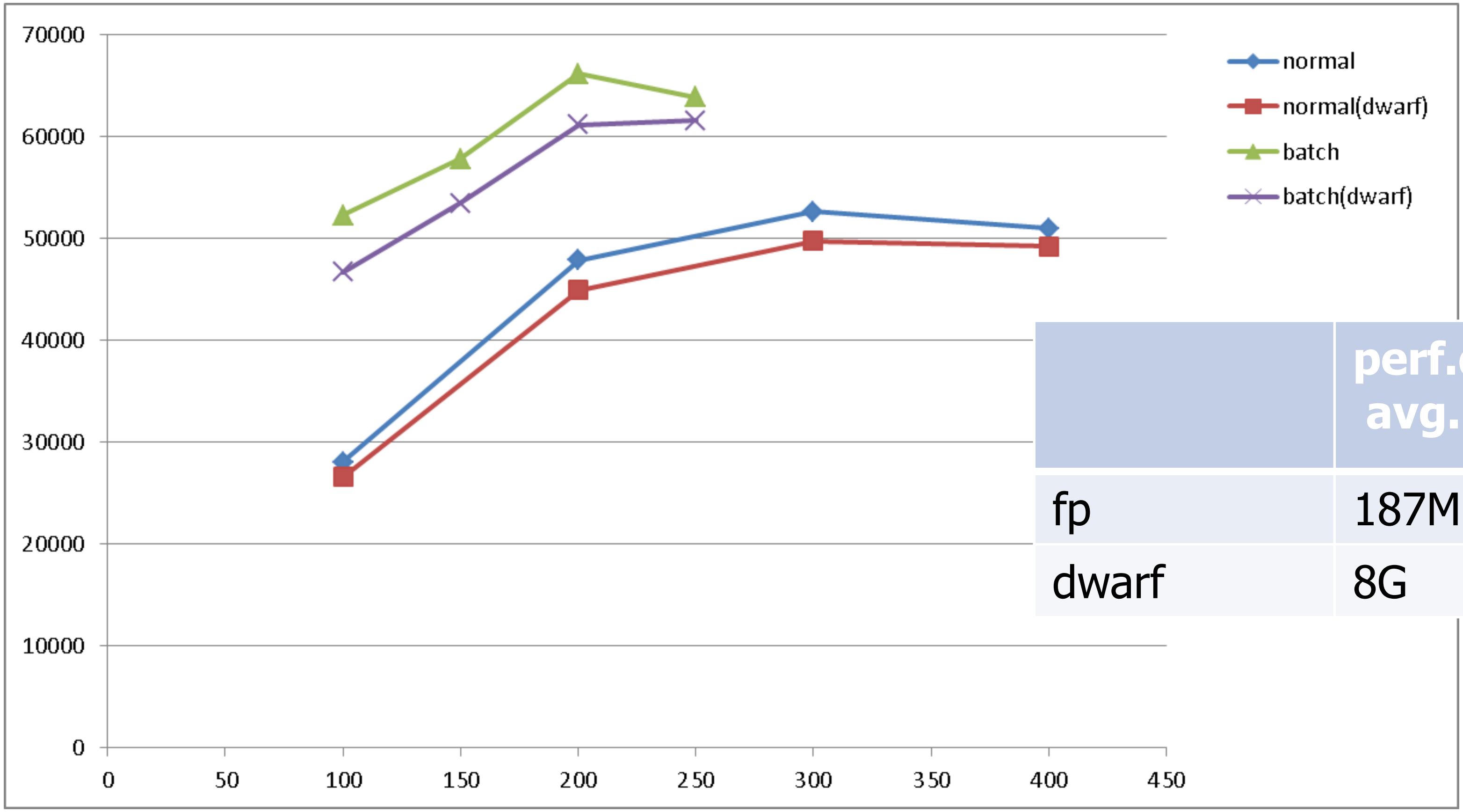
The perf diff was used to see any changes in time spent for the lock.

The 5.64% decrease in s_lock indicates that the solution eased the lock contention.

```
# Baseline    Delta          Shared Object                 Symbol
# ........  .......  ....................  ........................................
#
    29.77%   -5.64%  postgres              [.] s_lock
     2.58%   -0.25%  postgres              [.] GetSnapshotData
     1.54%   +0.64%  postgres              [.] base_yyparse
     2.00%   -0.04%  [kernel.kallsyms]     [k] update_cfs_rq_blocked_load
     2.38%   -0.54%  [kernel.kallsyms]     [k] update_blocked_averages
     1.67%   -0.32%  postgres              [.] AllocSetAlloc
     1.50%   -0.16%  postgres              [.] SearchCatCache
     0.97%   +0.08%  postgres              [.] hash_search_with_hash_value
     0.53%   +0.49%  postgres              [.] MemoryContextAlloc
     0.07%   +0.93%  postgres              [.] heap_fill_tuple
     2.45%   -1.51%  [kernel.kallsyms]     [k] tg_load_down
     1.24%   -0.41%  [kernel.kallsyms]     [k] _raw_spin_lock
```

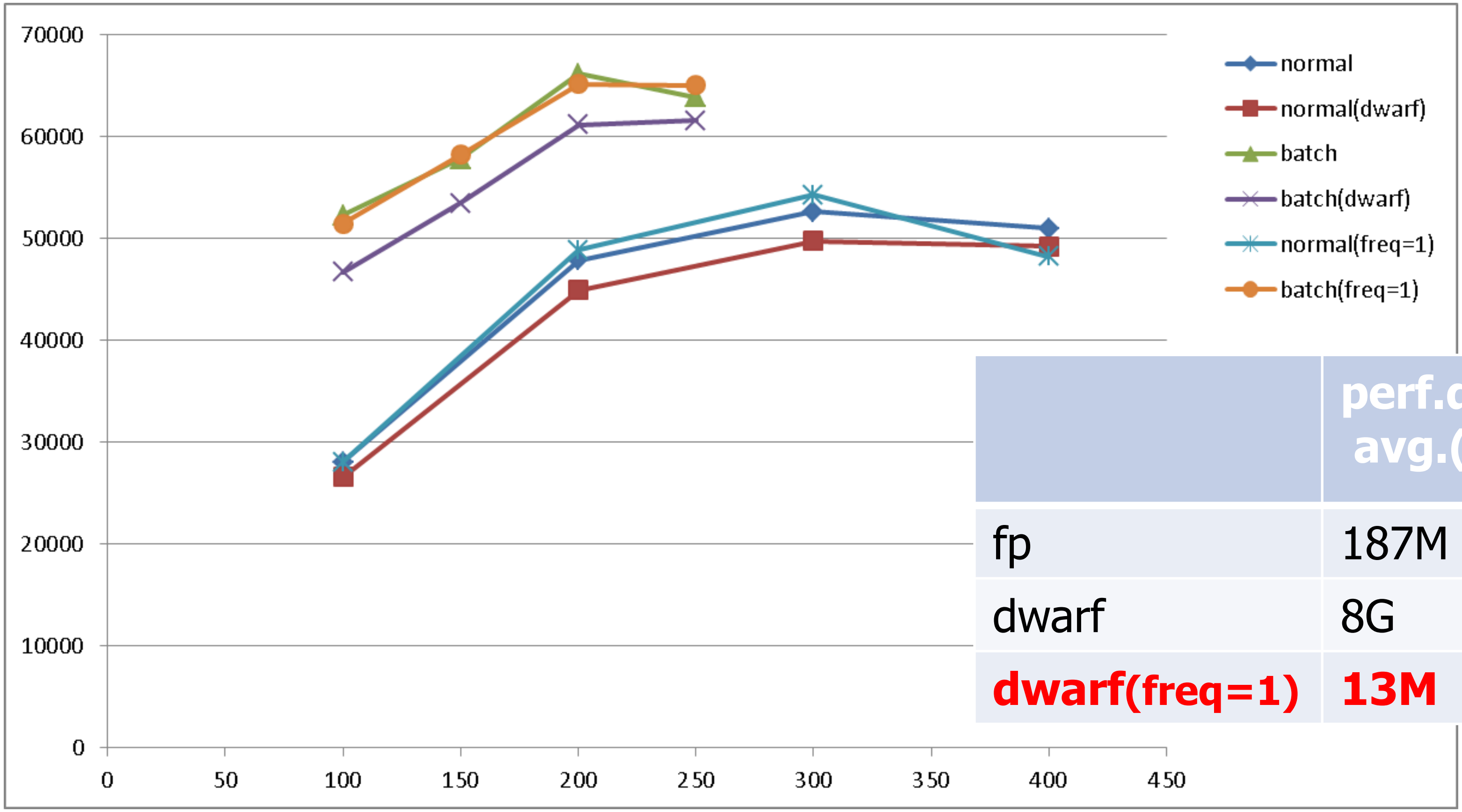
Side effect of the DWARF option

Turning the DWARF option on alone can cause significant decrease in performance. Though the exact figure depends on types of workload, we observed up to 10% performance penalty. This is due to much increased amount of I/O perf itself does to store sampled data, and can be controlled reducing sampling frequency or damp stack size.



	perf.data size avg.(60sec)
fp	187M
dwarf	8G

We ran the perf with the DWARF option enabled and sampling frequency reduced to 1 from the default value of 4000 by the "--freq" option, and found it can successfully reduce its impact on performance in the batch mode.



	perf.data size avg.(60sec)
fp	187M
dwarf	8G
dwarf(freq=1)	**13M**

Conclusion

Future works

- Tracing PostgreSQL by perf probe
- Efficient profiling data analysis of PostgreSQL by perf script
- Profiling other middleware than PostgreSQL