

An introduction to last branch records

Let's say we have a large program that runs too slowly; its performance needs to be improved. The program is big and complicated and not completely understood. The code may or may not be a [big ball of mud](#). To improve performance we now need to find which parts are the performance bottlenecks. A standard profiler, like Linux perf, will help identify them, but often just knowing the bottlenecks is not good enough. It may be not obvious why they are slow, or what causes them to be executed.

March 23, 2016

This article was contributed by Andi Kleen

So, often it is not enough to look at a single location in the program, but we need to see how larger parts of the program behave together. For that, we need to look at the control flow of frequently executed "traces" of the program. A facility in Intel processors, last branch records (LBRs), provide a way to look at this control flow. These CPUs can log branch information to special registers in realtime.

Branches

At the most basic software level, a program consists of computation and control structures (branches). Looking at the branches is a good start, since branches can be mapped back to source lines and the control structures of the programming language. Of course it is nice to see the values of variables too, but those values change frequently, which generates lots of information. Control flow and branches are much easier to analyze and are quite useful to understand how the program behaves. So it's useful to have a way to record and analyze branches.

Let's look at a simple example:

```
for (i = 0; i < 10; i++)
    if (i % 2 != 0)
        foo(i);
```

translates to the following pseudocode:

```
i = 0
loop:
    tmp = i % 2
    if tmp == 0 goto l2          # jump 1
    call foo(i)                 # jump 2
l2:
    if i++ < 10 goto loop        # jump 4

function foo(i):
    ...
    return                      # jump 3
```

which then can be mapped to assembler instructions.

If we can trace the jumps, we can understand what the program does and, more importantly, how to make it faster. For example, the `foo()` function may be slow and we want to understand what exactly causes it to be called and if the number of calls can be reduced.

If we look at the execution of the above example we get:

```
jump 1    # if tmp == 0 goto l2 (branch not taken, but still traced)
jump 2    # call foo(i)
jump 3    # return
jump 4    # if i++ < 10 goto loop (branch taken)
jump 1    # if tmp == 0 goto l2 (branch taken, since i % 2 != 0)
```

```
jump 4      # if i++ < 10 goto loop (branch taken)
... repeat 8 more times ...
```

This execution pattern can then be analyzed. For example we can see that `foo()` is only called for half of the loop iterations. Such information can then be used for performance tuning.

Not all program control flow results in branches. In some cases the compiler can generate branch-less code (for example, using conditional move instructions), which will not show up as individual branches. But most interesting code usually has real branches.

Of course, for such a trivial example it's relatively straightforward to understand what is happening even without such a trace. But for complex programs, we often need all the help the tools can provide.

Logging branches

A compiler or binary analyzer can do static control-flow analysis over a program to determine branches. But without running the program — only being able to guess — it doesn't know which branches are hot and which are cold. When doing performance analysis, we only care about hot code (tuning cold code won't help anyone), so knowing what is hot is important.

Also, static control-flow graphs are incomplete; for example, they cannot follow indirect function calls where the target is only known at run time and typically do not know the loop iteration counts. To really understand these things, branches need to be collected at run time.

A modern CPU is likely running at multiple GHz and can execute many instructions in parallel. For typical code, there is a branch every five to ten instructions. This gives a large number of branches: if we log the 64-bit target address of every branch each ten instructions on a 3GHz CPU (about one branch every 3ns) we would generate 2.2GB of data each second per processor thread. For most purposes, this is far too much data and, in fact, may be beyond what the memory subsystem or the disk can store.

There is a way to record every branch using [Processor Trace](#) (PT) on newer Intel CPUs. PT addresses this problem by heavily compressing branches and avoiding redundant information. But it can still generate an overwhelming amount of information.

Sampling

For performance tuning, we can often use sampling instead. We are only interested in the most common (hot) code paths, which will eventually show up in sampling (at least most of the time, short of [systematic shadow](#) [PDF] effects). So by sampling branches, we can build up a histogram of interesting control-flow patterns.

The overhead of sampling can be adjusted by lowering the sampling rate, at a tradeoff to accuracy. Sampling generates far less data than full tracing, which makes any analysis later much easier.

The CPU has performance counters that can be programmed to count branches and raise an interrupt on every Nth branch. Linux `perf` can be configured to sample branches using performance counters. However, N cannot be too small because interrupts are expensive and doing them frequently would slow down the workload too much. Usually, we are interested in short sequences of branches (for example the loop body of a hot loop) where it is useful to see multiple consecutive branches.

Last branch records to the rescue

Intel CPUs have a feature called last branch records (LBR) where the CPU can continuously log branches to a set of model-specific registers (MSRs). The CPU hardware can do this in parallel while executing the program without causing any slowdown. There is some performance penalty for reading these registers, however.

The LBRs log the "from" and "to" address of each branch along with some additional metadata. The registers act like a ring buffer that is continuously overwritten and provides only the most recent entries. There is also a TOS (top of stack) register to provide a pointer to the most recent branch. With LBRs we can sample branches, but during each sample look at the previous 8-32 branches that were executed. This gives reasonable coverage of the control flow in the hot code paths, but does not overwhelm us with too much information, as only a smaller number of the total branches are examined.

The number of branch entries in the LBRs varies depending on the Intel CPU generation:

CPU generation	Branches in LBR
Netburst to Merom	4
Nehalem to Haswell	16

Skylake	32
Atom	8

Once we are able to sample LBRs it is possible to set up sampling of branch events at a frequency that does not slow down the workload unduly, and still create a useful histogram of hot branches.

It is important to keep in mind that this is still sampling, so not every executed branch can be examined. CPUs generally execute too fast for that to be feasible. That is one reason why it's not a good idea to try to use LBRs to detect specific patterns of security exploits, for example. Security checks require examining everything, but sampling cannot do that.

Basic block frequencies

Linux perf is a profiler integrated with Linux. [perf record](#) supports [sampling](#) the LBRs using the -b option.

```
% perf record -b workload
```

That gives us a list of hot branches, but what can we do with the data? One simple use is to show the frequency of basic blocks using a histogram. This gives us the frequency of every program block or, more importantly, how often every branch to a given target is executed.

This is supported in perf by [perf report](#), which generates a histogram from the sampling data collected earlier by `perf record`. Note that `div` is the test program in the listing below.

```
% perf record -b -e cycles:u ./div
% perf report --sort symbol_from,symbol_to --stdio
...
# Samples: 632K of event 'cycles'
# Event count (approx.): 632064
#
# Overhead  Source Symbol                                Target Symbol
# .....
#
# 32.71%  [.] main                                [.] main
# 22.90%  [.] main                                [.] compute_flag
# 22.41%  [.] compute_flag                       [.] main
# 21.60%  [.] compute_flag                       [.] compute_flag
```

This gives us a histogram of all the branches. We know 33% of branches are inside main, and 23% of branches are from main to compute flag, and another 23% back.

Currently perf can only report functions, not source lines, in this mode, which limits the usefulness somewhat, as functions often contain a large number of branches and it may not be clear which are the hot and cold ones.

Compiler profile feedback

One way to use this information is for profile feedback for compilers, like GCC or LLVM. Modern compilers support many optimizations, but normally they operate with one hand tied behind their back by not knowing which code is hot and which is cold. They can use the basic block frequencies and branch target information to guide their optimizations, such as function inlining, and to do de-virtualization (inlining hot indirect method calls). Often important optimizations, such as function splitting to optimize instruction cache use, are only available with profile feedback.

Traditionally such profiling information was collected with specially instrumented binaries, typically with counters for each basic block and other hooks to collect the targets of indirect jumps. Adding this instrumentation slows down the programs quite a bit, and it requires separate binaries, which can be hard to manage in a production environment.

Getting profile feedback directly out of a profiler using a hardware profiling mechanism is better because it can be done directly on the production binary, with only the impact for collecting the data. This allows continuous optimization by regularly feeding back dynamic profiling data to a rebuild.

Since the compiler primarily needs basic block frequencies, sampling LBRs is the best way to collect this information, as that gives good coverage of the branches with reasonable overhead. Google implemented such a scheme for GCC and LLVM with the [Automatic Feedback](#)

[Directed Optimizer](#) (AutoFDO) project. AutoFDO has been available in GCC since version 5.0 (or in a slightly improved form in the 4.9 Google branch), or in LLVM since 3.5.

We sample the branches in a workload with LBRs enabled:

```
% gcc -O2 -o program ...
% ocperf.py record -b -e BR_INST_RETIRED.ALL_BRANCHES:p program
% create_gcov --binary=program --profile=perf.data -gcov_version=1
% gcc -fauto-profile -O2 ...
```

(ocperf.py is a perf wrapper available in [pmu-tools](#) that allows resolving named performance monitoring unit events such as BR_INST_RETIRED.ALL_BRANCHES. Without it, using something like `-e cpu/event=0xc4,umask=0x0/` would be needed.)

AutoFDO has some capability to tolerate stale profiles — profile data that has been collected using an older version of the binary. The more out of date it gets, the less useful it is, but this often avoids reprofiling for smaller changes. The AutoFDO toolchain can also generate similar feedback profile data for LLVM/Clang with the `create_llvm_prof` tool. This is supported in LLVM since 3.6.

The performance increase using AutoFDO can be substantial. For example, when building GNU `awk` with profile feedback for a simple [benchmark](#), we see about 18% shorter run times for the feedback-optimized binary, with no significant slowdown occurring during the profiling phase. Of course, the results will vary depending on the workload; in particular, the gains may be less if the training workload is significantly different from the actual workload.

Hot-path analysis

Newer versions of perf can also use LBRs for profiling "super blocks", which are a combination of multiple basic blocks that frequently get executed in order. This allows finding the hot execution paths in the executable to examine them directly for optimization opportunities. In perf this is implemented by extending the call-stack display mechanism and adding the last basic blocks into the call stack, which is normally used to display the most common hierarchy of function calls.

One interesting case here is when something cheap (such as setting a flag) causes something expensive later. This is usually difficult to profile because the cheap operation will not show up in the profiler, and it may not be obvious from the expensive code what caused it. Let's look at an example:

```
1 /* div.c */
2 volatile int count;
3
4 __attribute__((noinline))
5 int compute_flag(int i)
6 {
7     if (i % 10 < 4)          /* ... in 40% of the iterations */
8         return i + 1;
9     return 0;
10 }
11
12 int main(void)
13 {
14     int i;
15     int flag;
16     volatile double x = 1212121212, y = 121212; /* Avoid compiler optimizing away */
17
18     for (i = 0; i < 2000000000; i++) {
19         flag = compute_flag(i);
20
21         /* Some other code */
22         count++;
23
24         if (flag)
25             x += x / y + y / x;      /* Execute expensive division if flag is set */
26     }
```

```

27         return 0;
28     }

```

Division can be expensive. In this example division is only executed in 40% of the loop iterations, but it is so expensive that it clearly shows up as the most expensive operation.

We want to focus on the path through the program that causes the division. This can be done by using the call graph (-g) and LBR (-b) options in `perf record` and the `--branch-history` option in `perf report`, which adds the last branch information to the call graph. Essentially it gives 8-32 branches extra context of why something happened:

```

% gcc -O2 -g -o div div.c
% perf record -e cycles:pp -c 1000000 -g -b ./div
% perf report --branch-history --stdio --no-children
...
# Overhead   Source:Line                               Symbol                               Shared Object
# .....    .....                               .....                               .....
#
...
  31.81%    div.c:25                               [...] main                           div
          |
          |--main div.c:19
          |   compute_flag div.c:10
          |   compute_flag div.c:10
          |   compute_flag div.c:8           <===== FLAG SETTING
          |   compute_flag div.c:6
          |   main div.c:19
          |   main div.c:19
          |   main div.c:18
          |   |
          |   |--17.50%--main div.c:19
          |   |       compute_flag div.c:10
          |   |       compute_flag div.c:10
          |   |       compute_flag div.c:8

```

We can examine the path through the source code through the line numbers. `div.c:25` is the division. The path through the program is printed reversed (last on the top). When looking at what runs before the division we see it is always `div.c:8`, which is the flag setting that causes the division later. If we want to avoid the division, we would need to optimize how flag is set.

The `--branch-history` option was added to `perf` 3.18. Right now, using this facility requires matching line numbers manually, which can be tedious. In future versions of `perf`, this will hopefully be improved.

Conclusion

Last branch records are a powerful facility, which can enable advanced performance-analysis techniques. They also can be used to generate better code from compilers. Linux `perf` has support for using LBRs to help improve performance. While some details could be improved in `perf`, the support is working well enough to be a useful tool in day-to-day performance tuning.

[\(Log in](#) to post comments)