Instrumentation based Profile Guided Optimization (PGO) is a compiler technique that leverages important program runtime information, such as precise edge counts and frequent value information, to make frequently executed code run faster. It's proven to be one of the most effective ways to improve program performance.

An important design point of PGO is to decide where to place the instrumentation. In current LLVM PGO, the instrumentation is done in the Clang Front-End (will be referred as FE based instrumentation). Doing this early in the front-end gives better coverage information as the there are more precise line number information. The resulted profile also has relatively high tolerance to compiler changes. All the compiler change after the instrumentation point will not lead to mismatched IR that invalidates the profile.

On the other hand, doing this too early gives the compiler fewer opportunities to optimize the code before instrumentation. It has significant impact on instrumentation runtime performance. In addition, it tends to produce larger binary and profile data file. Our internal C++ benchmarks shows that FE based instrumentation degrades the performance (compared to non-instrumented version) by 58.3%, and in some extreme cases, the application speed/throughput decreases by 95% (21x runtime slowdown).

Running the instrumented binary too slow is not desirable in PGO for the following reasons:

- This slows down already lengthy build time. In the worst case, the instrumented binary is so slow that it fails to run a representative workload, because slow execution can leads to more time-outs in many server programs. Other typical issues include: text size too big, failure to link instrumented binaries, memory usage exceeding system limits.
- Slow runtime affects the program behavior. Real applications sometimes monitor the program runtime and have different execution path when the program run too slow. This would defeat the underlying assumption of PGO and make it less effective.

This work proposes an option to turn on middle-end based instrumentation (new) that aims to speed up instrumentation runtime. The new instrumentation is referred to as ME based instrumentation in this document. Our experimental results show that ME instrumentation can speed up the instrumentation by 80% on average for typical C++ programs. Here are the two main design objectives:

- **Co-existing with FE Instrumenter:** We do not propose to replace the FE based instrumentation because FE based instrumentation has its advantages and applications. User can choose which phase to do instrumentation via command line options.
- **PGO Runtime Support Sharing:** The ME instrumenter will completely re-use the existing PGO’s runtime support.
1. FE Based Instrumentation Runtime Overhead Analysis

Instrumented binaries are expected to run slower due to instrumentation code inserted. With FE based instrumentation, the overhead is especially high and runtime slowdown can be unacceptable in many cases. Further analysis shows that there are 3 important factors contributing to FE instrumentation slowdown:

- [Main] Redundant counter updates of inlined functions. C++ programs can introduce large abstraction penalties by using lots of small inline functions (assignment operators, getters, setters, ctors/dtors etc). Overhead of instrumenting those small functions can be very large, making training runs too slow and in some cases to usable;
- Non-optimal placement of the count updates;
- A third factor is related value profiling (to be turned on in the future). Small and hot callee functions taking function pointer (callbacks) can incur overhead due to indirect call target profiling.

1.1 Redundant Counter Update

If checking the assembly of the instrumented binary generated by current LLVM implementation, we can find many sequence of consecutive 'incq' instructions that updating difference counters in the same basic block. As an example that extracted from real binary:

```assembly
... incq 0xa91d80(%rip)       # 14df4b8 <__llvm_profile_counters__ZN13LowLevelAlloc5ArenaC2Ev+0x1b8>
incq 0xa79011(%rip)        # 14c6750 <__llvm_profile_counters__ZN10MallocHook13InvokeNewHookEPKvm>
incq 0xa79442(%rip)        # 14c6b88 <__llvm_profile_counters__ZNK4base8internal8HookListIPFvPKvmEE5emptyEv>
incq 0x9c288b(%rip)        # 140ffd8 <__llvm_profile_counters__ZN4base6subtle12Acquire_LoadEPVL>
...```

From profile use point of view, many of these counter update are redundant.

Considering the following example:

```c
void bar()
{
    sum++;
}

void foo() {
    bar();
}
```

FE based instrumentation needs to insert counter update for the only BB of the bar().

```assembly
bar:   # @bar
    # BB#0:     # %entry
    incq .L__llvm_profile_counters_bar(%rip)
```
Incl  sum(%rip)
retq

It also need to insert the update the BB in function foo(). After inlining bar to foo(), the code is:

```
foo:                       # @foo
# BB#0:                   # %entry
  incl .L__llvm_profile_counters_foo(%rip)
  incl .L__llvm_profile_counters_bar(%rip)
retq
```

If bar() should be always inlined, .L__llvm_profile_counters_bar(%rip) is redundant -- the counter won't help downstream optimizations. On the other hand, if bar() is a large function and may not be suitable to be inlined for all callsites, this counter updated is necessary in order to produce more accurate profile data for the out-of-line instance of the callee.

If foo() is a hot function, the overhead of updating two counters can be significant. This is especially bad for C++ program where there are tons of small inline functions.

There is another missing opportunity in FE based instrumentation. The small functions’ control flow can usually be simplified when they are inlined into caller contexts. Once the control flow is simplified, many counter updates can therefore be eliminated. This is only possible for a middle end based late instrumenter.

Defining a custom clean-up pass to remove redundant counter update is unrealistic and cannot be done in a sane way without destroying the profile integrity of neither the out-of-line nor inline instances of the callee.

A much simpler and cleaner solution is to do a pre-inline pass to inline all the trivial inlines before instrumentation. In addition to removing the unnecessary count updates for the inline instances, another advantage of pre-inline is to provide context sensitive profile for these small inlined functions. This context senstive profile can further improve the PGO based optimizations.

Here is a contrived example:

```c
void bar (int n) {
  if (n&1)
    do_sth1();
  else
    do_sth2();
}

void caller() {
  int s = 1;
```
for (; s<100; s+=2)
    bar(s);

for (s = 102; s< 200; s+=2)
    bar(s);
}

The direction of the branch inside bar will be totally opposite between two different callsites in 'caller'. Without pre-inlining, the branch probability will be 50-50 which will be useless for later optimizations. With pre-inlining, the profile will have the perfect branch count for each callsite. The positive performance impact of context sensitive profiling due to pre-inlining has been observed in real world large C++ programs. Supporting context sensitive profiling is another way to solve this, but it will introduce large additional runtime/memory overhead.

### 1.2 Non-optimal placement of count update

Another much smaller showdown factor is the placement of the counter updates.

Current front-end based instrumentation applies the instrumentation to each front-end lexical construct. It also minimizes the number of static instrumentations. Note that it always instruments the entry count of the CFG. This may result in higher dynamic instruction counts. For example,

```
BB0
  | 100
BB1
  90  /  \ 10
BB2 BB3
  90  \  / 10
BB4
```

Like the the above example, FE based instrumentation will always insert count update in BB0. The dynamic instrumentation count will be either 110 (Instrument bb0->bb1 and bb1->bb2) or 190 (bb0->bb1 and bb1->bb3). A better instrumentation is to instrument (bb1->bb2 and bb1->bb3) where the dynamic instrumentation count is 100.

Our experimental shows that the optimal placement based on edge hotness can improve instrumented code performance by about 10%. While it's hard to find the optimal placement of count update, compiler heuristics can be used to get the better placement. These heuristics can be based on static profile prediction or user annotations (like __builtin_expect) to estimate the relative edge hotness and put instrumentations on the less hot edges. The initial late instrumentation has not fully implemented this placement strategy yet. With that implemented, we expect even better results than what is reported here.
For real world programs, another major source of the slowdown is the data racing and false sharing of the counter update for highly threaded programs. Pre-inlining can alleviate this problem as the counters in the inline instances are not longer shared. But the complete solution to the data racing issue is orthogonal to the problem we try to solve here.

2. High Level Design

We propose to perform a pre-profile inline pass before the PGO instrumentation pass. Since the instrumentation pass is after inine, it has to be done in the middle-end.

(1) The pre-inline pass
We will invoke a pre-inline pass before the instrumentation. When PGO is on, the inlining will be split into two passes:

- A pre-inline pass that is scheduled before the profile instrumentation/annotation
- A post-inline pass which is the regular inline pass after instrumentation/annotation

By design, all beneficial callsites without requiring profile data should be inlined in the pre-inline pass. It includes all callsites that will shrink code size after inlining. All the remaining callsites will be left to the regular inline pass when profile data is available.

After pre-inline, a CFG based profile instrumentation/annotation will be done. A minimum weight spanning tree (MST) in CFG is first computed, then only the edges not in the MST will be instrumented. The counter update instructions are placed in the basic blocks.

(2) Minimum Spanning Tree (MST) based instrumentation
A native way of instrumentation is to insert a count update for every edge in CFG which will result in too many redundant updates that makes the runtime very slow. Knuth [1] proposed a minimum spanning tree based method: given a CFG, first compute a spanning tree. All edges that not in the MST will be instrumented. In the profile use compilation, the counters are populated (from the leaf of the spanning tree) to all the edges. Knuth proved this method inserts the minimum number of instrumentations. MST based method only guarantees the number static instrumentation are minimized, not the dynamic instance of instrumentation. To reduce the number of dynamic instrumentation, edges of potentially high counts will be put into MST first so that they will have less chance to be instrumented.

3. Experimental Results

3.1 Measurement of the efficiency of instrumentation

Other than the runtime of the instrumented binaries, a more direct measurement of the instrumentation overhead is the the sum of the raw profile count values. Note that regardless
what kind of instrumentations are used, the raw profile count should be able to reconstruct all
the edge count values for the whole program. All raw profile value are obtained via incrementing
the counter variable value by one. The sum of the raw profile count value is roughly the dynamic
instruction count of the instrumented code. The lower of the value, the more efficient of the
instrumentation.

3.2 LLVM instrumentations runtime for SPEC2006 C/C++ programs and SPEC2K eon
The performance speedup is computed by
\[(\text{FE\_instrumentation\_runtime} / \text{ME\_instrumentation\_runtime} - 1)\]
We run the experiments on all C/C++ programs in SPEC2006 and 252.eon from SPEC2000. For C programs, except for one outlier 456.hmmer, there are small ups and downs across
different programs. Late instrumentation improves hmmer a lot, but it is probably due to
unrelated loop optimizations (90% of runtime spent in one loop nest).

For C++ programs, the performance impact of late instrumentation is very large, which is as
expected. The following table shows the result. For some C++ program, the time speedup is
huge. For example, in 483.xalancbmk, late instrumentation speeds up performance by ~60%.
Among all the SPEC C++ programs, only 444.namd is an outlier -- it uses a lot of macros and is
a very C like program.

<table>
<thead>
<tr>
<th>Program</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>471.omnetpp</td>
<td>16.03%</td>
</tr>
<tr>
<td>473.astar</td>
<td>5.00%</td>
</tr>
<tr>
<td>483.xalancbmk</td>
<td>58.57%</td>
</tr>
<tr>
<td>444.namd</td>
<td>-0.90%</td>
</tr>
<tr>
<td>447.dealII</td>
<td>60.47%</td>
</tr>
<tr>
<td>450.soplex</td>
<td>8.20%</td>
</tr>
<tr>
<td>453.povray</td>
<td>11.34%</td>
</tr>
<tr>
<td>252.eon</td>
<td>35.33%</td>
</tr>
</tbody>
</table>

| Geomean     | 21.01%   |

3.3 Statistics of LLVM profiles for SPEC2006 C/C++ programs
We also collect some statistic of the profiles generated by FE based instrumentation and late
instrumentation, namely, the following information:
(1) the number of functions that being instrumented,
(2) the result profile file size,
(3) the sum of raw count values that was mentioned earlier -- we used it to measure the
efficiency of the instrumentation.

Next table shows the ratios of the each metrics by late instrumentation for the C++ programs,
with FE based instrumentation as the base : column (1) shows the ratios of instrumented
functions; column (2) shows the ratios of the profile file size; column (3) shows the ratios of the sum of raw count values.

<table>
<thead>
<tr>
<th></th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>471.omnetpp</td>
<td>85.36%</td>
<td>110.26%</td>
<td>46.52%</td>
</tr>
<tr>
<td>473.astar</td>
<td>64.86%</td>
<td>72.72%</td>
<td>63.13%</td>
</tr>
<tr>
<td>483.xalancbmk</td>
<td>51.83%</td>
<td>56.11%</td>
<td>35.77%</td>
</tr>
<tr>
<td>444.namd</td>
<td>75.36%</td>
<td>82.82%</td>
<td>85.77%</td>
</tr>
<tr>
<td>447.dealII</td>
<td>43.42%</td>
<td>46.46%</td>
<td>26.75%</td>
</tr>
<tr>
<td>450.soplex</td>
<td>71.80%</td>
<td>87.54%</td>
<td>51.19%</td>
</tr>
<tr>
<td>453.povray</td>
<td>78.68%</td>
<td>83.57%</td>
<td>64.37%</td>
</tr>
<tr>
<td>252.eon</td>
<td>72.06%</td>
<td>91.22%</td>
<td>30.02%</td>
</tr>
<tr>
<td>Geomean</td>
<td>66.50%</td>
<td>76.36%</td>
<td>47.01%</td>
</tr>
</tbody>
</table>

For FE based instrumentation, profile count variables generated for the dead functions will not be removed (like __llvm_prf_names, __llvm_prf_data, and __llvm_prf_cnts) from the data/text segment, nor in the result profile. There is a recent patch that removes these unused data for COMDAT functions, but that patch won’t touch regular functions. This is the main reason for the larger number of instrumented functions and larger profile file size for the FE based instrumentation. The reduction of the sum of raw count values is mainly due to the elimination of redundant profile updates enabled by the pre-inlining.

For C programs, we observe similar improvement in the profile size (geomean ratio of 73.75%) and smaller improvement in the number of instrumented functions (geomean ratio of 87.49%) and the sum of raw count values (geomean of 82.76%).

### 3.4 LLVM instrumentations runtime performance for Google internal C/C++ benchmarks

We also use Google internal benchmarks (mostly typical C++ applications) to measure the relative performance b/w FE based instrumentation and late instrumentation. The following table shows the speedup of late instrumentation vs FE based instrumentation. Note that C++benchmark01 is a very large multi-threaded C++ program. Late instrumentation sees 4x speedup. Larger than 3x speedups are also seen in many other programs.

<table>
<thead>
<tr>
<th>C++_benchmark01</th>
<th>416.98%</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++_benchmark02</td>
<td>6.29%</td>
</tr>
<tr>
<td>C++_benchmark03</td>
<td>22.39%</td>
</tr>
<tr>
<td>C++_benchmark04</td>
<td>28.05%</td>
</tr>
<tr>
<td>C++_benchmark05</td>
<td>2.00%</td>
</tr>
<tr>
<td>C++_benchmark06</td>
<td>675.89%</td>
</tr>
<tr>
<td>C++_benchmark07</td>
<td>359.19%</td>
</tr>
<tr>
<td>C++_benchmark08</td>
<td>395.03%</td>
</tr>
<tr>
<td>C_benchmark09</td>
<td>15.11%</td>
</tr>
</tbody>
</table>
### 3.5 Statistics of LLVM profiles for Google internal benchmarks

The following shows the profile statics using Google internal benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++_benchmark01</td>
<td>36.84%</td>
<td>40.29%</td>
<td>2.32%</td>
</tr>
<tr>
<td>C++_benchmark02</td>
<td>39.20%</td>
<td>40.49%</td>
<td>42.39%</td>
</tr>
<tr>
<td>C++_benchmark03</td>
<td>39.37%</td>
<td>40.65%</td>
<td>23.24%</td>
</tr>
<tr>
<td>C++_benchmark04</td>
<td>39.13%</td>
<td>40.68%</td>
<td>17.70%</td>
</tr>
<tr>
<td>C++_benchmark05</td>
<td>36.58%</td>
<td>38.27%</td>
<td>51.08%</td>
</tr>
<tr>
<td>C++_benchmark06</td>
<td>29.50%</td>
<td>27.87%</td>
<td>2.87%</td>
</tr>
<tr>
<td>C++_benchmark07</td>
<td>29.50%</td>
<td>27.87%</td>
<td>1.73%</td>
</tr>
<tr>
<td>C++_benchmark08</td>
<td>29.50%</td>
<td>27.87%</td>
<td>4.17%</td>
</tr>
<tr>
<td>C++_benchmark09</td>
<td>53.95%</td>
<td>68.00%</td>
<td>11.18%</td>
</tr>
<tr>
<td>C++_benchmark10</td>
<td>53.95%</td>
<td>68.00%</td>
<td>31.74%</td>
</tr>
<tr>
<td>C++_benchmark11</td>
<td>36.40%</td>
<td>37.07%</td>
<td>46.12%</td>
</tr>
<tr>
<td>C++_benchmark12</td>
<td>38.44%</td>
<td>41.90%</td>
<td>73.59%</td>
</tr>
<tr>
<td>C++_benchmark13</td>
<td>39.28%</td>
<td>42.72%</td>
<td>29.56%</td>
</tr>
<tr>
<td>C++_benchmark14</td>
<td>38.59%</td>
<td>42.20%</td>
<td>13.42%</td>
</tr>
<tr>
<td>C++_benchmark15</td>
<td>57.45%</td>
<td>48.50%</td>
<td>66.99%</td>
</tr>
<tr>
<td>C++_benchmark16</td>
<td>36.86%</td>
<td>42.18%</td>
<td>16.53%</td>
</tr>
<tr>
<td>C++_benchmark17</td>
<td>37.82%</td>
<td>39.77%</td>
<td>13.68%</td>
</tr>
<tr>
<td>C++_benchmark18</td>
<td>37.82%</td>
<td>39.77%</td>
<td>7.96%</td>
</tr>
<tr>
<td>C++_benchmark19</td>
<td>37.52%</td>
<td>40.46%</td>
<td>1.85%</td>
</tr>
<tr>
<td>C++_benchmark20</td>
<td>32.37%</td>
<td>30.44%</td>
<td>19.69%</td>
</tr>
<tr>
<td>C++_benchmark21</td>
<td>37.63%</td>
<td>40.42%</td>
<td>88.81%</td>
</tr>
<tr>
<td>C++_benchmark22</td>
<td>36.28%</td>
<td>36.92%</td>
<td>21.62%</td>
</tr>
</tbody>
</table>

Geomean: 83.03%
4. Implementation Details:

We need to add new option(s) for the alternative PGO instrumentation pass in the middle end. It can in one of the following forms:

1. Complete new options that are on par with current PGO options:
   - `fprofile-late-instr-generate[=<profile_file>]` for PGO Instrumentation, and
   - `fprofile-late-instr-use[=<profile_file>]` for PGO USE.

2. Or, late instrumentation can be turned on with an additional option
   - `fprofile-instr-late` with current PGO options. I.e.
   - `fprofile-instr-late -fprofile-instr-generate[=<profile_file>]` for PGO instrumentation, and
   - `fprofile-instr-late -fprofile-instr-use[=<profile_file>]` for PGO use.

3. Alternatively to (2), only keep `fprofile-instr-late` option in PGO instrumentation. Adding a magic tag in profile so that FE based profile and late instrumented profile can be automatically detected by profile loader. In PGO use compilation. This requires a slight profile format change.

In our prototype implementation, two new passes are added in the beginning of PassManagerBuilder::populateModulePassManager(), namely PreProfileInlinerPass and PGOLateInstrumentationPass.

4.1 Pre-inline pass:
It is controlled by back-end option "-preinline" and "-disable-preinline". If the user specifies any llvm option of "-fprofile-late-instr-{generate|use}, option "-mllvm -preinline" will be automatically inserted in the driver. To disable the pre-inliner when late instrumentation is enabled, use option "-mllvm -disable-preinline".

For now, only minimum tuning is done for the pre-inliner, which simply adjusts the inline threshold: If -Oz is specified, the threshold is set to 25. Otherwise, it is 75.

The following clean up passes are added to PassManager, right after the PreProfileInliner pass:

- `createEarlyCSEPass()`
- `createJumpThreadingPass()`
- `createCorrelatedValuePropagationPass()`
- `createCFGSimplificationPass()`
- `createInstructionCombiningPass()`
createGVNPass(DisableGVNLoadPRE)
createPeepholePASS()
Some of them might not be necessary.

4.2 Late Instrumentation Pass:
The late instrumentation is right after the pre-inline pass and it's cleanup passes. It is controlled by opt option "-pgo-late-instr-gen" and "-pgo-late-instr-use". For "-pgo-late-instr-use" option, the driver will provide the profile name. For "-pgo-late-instr-gen", a pass that calls createInstrProfilingPass() is also added to PassManager to lower the instrumentation intrinsics.

PGOLateInstrumentation is a module pass that applies the instrumentation to each function by class PGOLateInstrumentationFunc. For each function, perform the following steps:

1. First collect all the CFG edges. Assign an estimated weight to each edge. Critical edges and back-edges are assigned to high value of weights. One fake node and a few fake edges (from the fake node to the entry node, and from all the exit nodes to the fake node) are also added to the worklist.
2. Construct the MST. The edges with the higher weight will be put to MST first, unless it forms a cycle.
3. Traverse the CFG and compute the CFG hash using CRC32 of the index of each BB. The above three steps are the same for profile-generate and profile-use compilation.

In the next step, for profile-generation compilation, all the edges that not in the MST are instrumented. If this is a critical edge, split the edge first. The actual instrumentation is to generate Intrinsic::instrprof_increment() in the instrumented BB. This intrinsic will be lowered by pass createInstrProfilingPass().

For -fprofile-use compilation, first read in the counters and the CFG hash from the profile file. If the CFG hash matches, populate the counters to all the edges in reverse topological order of the MST. Once having all the edge counts, set the branch weights metadata for the IR having multiple branches. Also apply the cold/hot function attributes based on function level counts.

4.3 Profile Format:
The late instrumentation profile is mostly the same as the one from front-end instrument-ion. The difference is

- Function checksums are different.
- Function entry counts are no longer available.

For llvm-profdata utility, options -lateinstr needs to be used to differentiate FE based and late instrumentation profiles, unless a magic tag is added to the profile.
5. References: