# Feedback-Directed Optimizations with Estimated Edge Profiles from Hardware Event Sampling

Vinodha Ramasamy Robert Hundt
Google Inc.
vinodha,rhundt@google.com

Dehao Chen Wenguang Chen
Tsinghua University
danielcdh@gmail.com, cwg@tsinghua.edu.cn

## Abstract

Traditional feedback-directed optimization (FDO) uses static instrumentation to collect profiles. This method has shown good application performance gains, but is not commonly used in practice due to the high runtime overhead of profile collection, the tedious dual-compile usage model, and difficulties in generating representative training data sets. In this paper, we show that edge frequency estimates can be successfully constructed with heuristics using profile data collected by sampling of hardware events, incurring low runtime overhead (e.g., less then 2%), and requiring no instrumentation, yet achieving competetive performance gains. Our initial results show a 3-4% performance gain on the SPEC C benchmarks.

***Keywords*** Compiler, Profiling, Sampling

## 1. Introduction

Compilers use execution profiles consisting of basic block and edge frequency counts to guide optimizations such as instruction scheduling, procedure inlining, and register allocation. The traditional method of execution profile collection involves the following steps:

1. Build an instrumented version of the program.
2. Run the instrumented version with representative training data to collect the execution profile. These runs typically incur significant overhead (reported as 9% to 105% (Ball and Larus 1996) (Ball and Larus 1994), but observed to be much higher, often in the order of 50% to 200% in our experience) due to the additional instrumentation code that is executed.
3. Build an optimized version of the program by using the collected execution profile to guide the optimizations (FDO build).

The instrumentation builds are usually restricted to lower optimization levels or are tightly coupled with the FDO compilations in most compilers. For example, the profile data that is fed back to the open64 compiler must have been collected by an open64 instrumented binary. Additionally, open64 requires that both the instrumentation and FDO builds use the same inline decisions and similar optimization flags (such as with or without the `-ipa` option) to ensure that the control-flow graph (CFG) that is instrumented in the instrumentation build matches the CFG that is annotated with the profile data in the FDO build.

Besides complete execution profiles gathered by static instrumentation, researchers have used profiles gathered using hardware support (Merten et al. 1999) and statistical sampling in guiding code optimizations. These approaches typically have much lower overheads (less then 5%). So far, the profiles collected via hardware support have either been used to augment instrumented run profiles with flow and context-sensitive information (Ammons et al. 1997) or machine-specific information such as cache misses in online optimization systems (Schneider et al. 2007), or used separately for specific optimizations (Froyd et al. 2005) that do not encompass the wide range of optimizations enabled by complete basic block and edge frequency profiles.

Our approach is to use sampling of the Instruction Retired (INST_RETIRED) hardware event, which is available on performance monitoring units of modern processors (e.g., Intel Core-2, AMD Opteron) to *replace*, rather than *augment*, the traditional instrumentation methods to obtain basic block and edge frequency counts. This approach enables different usage models:

1. Profile collection can occur on production systems (e.g., in internet companies) using the default binaries, with the sampled profile data being stored in a profile repository. The profiles shall therefore be readily available for FDO builds *without the need for any special instrumentation*

*Open64 Workshop at CGO'08* 04/06/2008, Boston.
Copyright © 2008 ACM . . . $5.00

*build and run*. Moreover, there is no discrepancy between training run input data and real usage data in this case.

2. In cases where representative training data sets are available, the profile collection could be done using debug or un-optimized binaries. The profile data thus collected during the testing and development phase can then be used to build the optimized binary.
3. The traditional FDO model using instrumented runs to collect profile data is not suitable for cases where execution of the instrumented code changes the behavior of time critical code such as operating system kernel code. Profile collection using hardware event sampling can be used in such cases without perturbing the run-time behavior.

The sampled profile data does not contain any information on the intermediate representation (IR) used by the compiler. Instead, source position information is used to correlate the profile data to specific basic blocks during the FDO build. This method therefore *eliminates the tight coupling between profile collection and profile feedback builds*. In fact, the binary used for profile collection can be built by one compiler, and the profile data thus collected can be fed to another compiler. To make the case, in our experiments, we use gcc built -O0 and -O2 binaries for profile collection and open64 for FDO builds and performance experiments.

In general, deriving exact basic block and edge frequency counts from sample profiles is not always feasible. We use heuristics to derive relative basic block and edge frequency count estimates from the sampled profiles. Increasing the sampling rate will in general increase the quality of the sampled profile at the expense of increasing the overhead of profile collection. Our experiments show we can get sampled profiles with reasonable quality with overheads of less than 2%. We use a sample "goodness" measure derived from the weighted difference in branch biases between execution profiles collected from instrumented runs and sampled profiles as an indicator of the quality of the sampled profile. However, the definitive measure of the sample profile quality is ascertained only from the performance gains obtained in using the sampled profile for feedback-directed optimizations.

We use the Intel Core-2 platform for both profile collection and for our performance runs. Note that the profile metrics collected are platform independent, so the profile data can be used to build a binary optimized for another platform than the one used for profile collection. Since the profile data is stored by samples per source line, it will not matter if the profile collection is done using optimized or unoptimized binaries. Our heuristics depend on the correctness of the source position information present in the binaries to correlate the samples to the corresponding basic blocks. [1]

On the SPEC2000int and SPEC2006 C benchmarks, we currently obtain an average performance gain of approximately 3% using FDO with sampled profiles collected using -O2 binaries, as compared to an average of approximately 6% using traditional FDO runs. We expect to get improved results with better source correlation in gcc. Using -O0 binaries for profile collection, we are able to achieve approximately 85% of the performance gains seen using traditional FDO on Spec2006 C benchmarks (3.59% vs 4.23%).

The rest of the paper is organized as follows: Section 2 gives a background of the hardware event counters used and the modes of operation. Section 3 describes the high-level design and methodology for sampled profile support in the open64 compiler. Section 4 describes the heuristics and implementation details of deriving basic block and edge frequency counts from the sampled profiles. Section 5 describes the experimental evaluation of using FDO with sampled profiles. Finally, sections 6 and 7 discuss related work and conclusions.

## 2. Hardware Event Sampling

### 2.1 Time-based vs Frequency-based Sampling

Most modern microprocessors support hardware event sampling, which works as follows: The Instruction Pointer and other register contents are recorded whenever a specified number of a specified hardware event has occurred. This helps to identify the program locations, i.e., the instruction addresses incurring the measured hardware event. For example, the DCPI tool (Anderson et al.) samples on the event CPU_CYCLES to determine performance bottlenecks in programs.

Events can be differentiated by whether they indicate execution time or execution frequency, i.e., whether they are time-based or frequency-based (Zhang et al. 1997). The CPU_CYCLES is a time-based event, so program locations that take a relatively longer time to execute will incur more CPU_CYCLES event samples. To obtain an execution count from such time-based samples, one must scale by the instruction latency, which necessitates knowing the individual instruction execution latencies and latencies incurred due to TLB misses, cache misses and branch misprediction, as well as other pipeline stalls, which are micro-architecture specific. Additional hardware events (such as cache and TLB misses) will therefore need to be sampled for this purpose, thereby increasing the sampling overhead and making the determination of execution counts from time-based event samples more complex. Most modern microprocessors also support sampling of frequency-based events such as the instruction retired (INST_RETIRED) event, which correlates directly to instruction and basic block execution count. We

[1] We ran into a couple of gcc issues - source information is at times lost during transformations in optimization builds. These issues are being fixed, which will help to improve the accuracy of sample attribution when using optimized binaries for profile collection.

therefore use sampling of the INST_RETIRED event for our execution profile estimation.

### 2.2 Perfmon

We use perfmon2, the hardware-based performance monitoring interface for Linux, and pfmon, the command-line-interface tool to gather INST_RETIRED samples on the Intel Core-2 platform (Intel 2007). Intel Core-2 machines support two modes of sampling: Precise Event-Based Sampling (PEBS) and the regular sampling mode, which we will refer to as non-PEBS. PEBS mode accurately identifies the next instruction address following the instruction incurring the sampled event. Moreover, PEBS incurs less overhead since the CPU collects event samples on its own using a microcode routine and stores them into a buffer supplied by the operating system (OS) kernel module. An interrupt is generated and the OS is involved only when the buffer becomes full. A drawback of PEBS-based sampling is that the hardware does not support randomization of every sample - a feature that helps to reduce sampling skews due to program synchronization with the sampling interval. We expected the use of the randomization feature to improve the quality of the sampling profiles and for samples collected in PEBS mode to yield better results than samples collected in non-PEBS mode. However our experiments (see section 5) show that all three modes (PEBS, non-PEBS with randomization, and non-PEBS without randomization) yield comparable performance gains when used for FDO with sampled profiles.

We determined the sampling rate of the INST_RETIRED events in PEBS and non-PEBS mode which will give a maximum overhead of 2%. When sampling every *n* INST_ RETIRED event, *n* should be chosen to be a prime number, to mitigate the possibility of program synchronization with the sampling interval (for example, in the presence of loops).

An issue with the current implementation of perfmon2 is that it uses a single buffer to record samples, which results in lost program samples whenever the OS processes the samples on buffer overflow. We overcome this problem by using the `--overflow-block` option in pfmon, which blocks the program execution on buffer overflow. As this method incurs considerable overhead, this is only a temporary workaround until perfmon2 implements dual-buffer support.

## 3. High-Level Design

The INST_RETIRED event samples are recorded on the granularity of instruction addresses and attributed to the corresponding program source filename and line number using the source position information present in unstripped binaries. Since source lines with larger number of instructions will have correspondingly larger total number of samples attributed, the total number of samples attributed per source line is divided by the number of contributing instructions to derive the average number of samples per source line, which is stored in the feedback data file.

```
FB_Sample_Hdr
PU_Sample_Hdr for PU 1
Pu_Sample_Hdr for PU 2
...
Pu_Sample_Hdr for PU NUM_PU
Pu_Sample_Hdr for Inline 1
 ...
Pu_Sample_Hdr for Inline NUM_INLINE
STRING TABLE
Fb_Info_Freq 1 for PU 1
...
Fb_Info_Freq N for PU 1
Fb_Info_Freq 1 to N for PU 2
...
Fb_Info_Freq 1 to N for PU NUM_PU
Fb_Info_Freq for Inline 1 to NUM_INLINE
```

**Figure 2.** Feedback data file format

The feedback file is read into the open64 compiler and is used to annotate the WHIRL statements for the current program unit with the relative execution counts of the corresponding source position information. This is done in the VHO phase for both profile feedback data collected from open64 instrumented runs and for profile data collected using sampling. The VHO phase belongs to the IPF phase under `-ipa` compiles and to the backend without `-ipa`. Figure 1 shows the phases where the profile data annotation is done for builds with and without inter-procedural optimizations. In our experiments, we do not use inter-procedural optimizations.

The basic block sample counts are calculated from the samples attributed to the individual WHIRL statements in each block. The counts are then used to derive edge frequency counts using heuristics which are described in detail in the next section.

The derived edge frequency estimates are used to construct the feedback data structures associated with each program unit. At the end of this phase, the feedback data structures will be initialized in a manner consistent with the initialization of these structures when using instrumented feedback data. This allows us to leverage the existing support in open64 for feedback-directed optimizations and for maintaining valid feedback information in later phases through the propagation and verification of the feedback information.

## 4. Implementation Details

### 4.1 Feedback Data File Format

The design of the sample profile feedback data file format is based on the file format used to store profile data collected from instrumented runs. The layout of the sample profile data file is given in figure 2.

`Fb_Sample_Hdr` is the file header. The data structure `Pu_Sample_Hdr` holds the header information pertain-

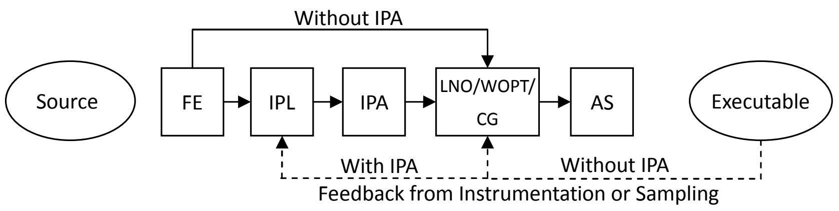

**Figure 1.** Open64 Modules and Flow

ing to each program unit. A program unit corresponds to a function. This format supports the aggregation of samples for inlined functions by caller function. If a function A has 3 inlined functions B, C and D with samples, the program header corresponding to A will have the `pu_num_inline_entries` set to 3 and assign the offset of the inline program header to `pu_inline_hdr_offset` (which shares the same structure as `Pu_Sample_Hdr`) corresponding to the inlined instance of B within function A. The inline headers for the inlined instances of functions C and D within function A will be stored consecutively following the inline header for B. The samples attributed to each inlined function can then be handled in a manner similar to non-inlined functions.

The data structure `Fb_Info_Freq` is used to store the sample count associated with each source line within a function. The `Fb_Info_Freq` data associated with a function will be stored consecutively. The `Pu_Sample_Hdr` for the function has the offset of the first `Fb_Freq_Info` data in the `pu_freq_offset` field and the number of `Fb_Freq_Info` associated with its function.

## 4.2 Basic Block Annotation

The sample count corresponding to each WHIRL statement (IR.count) is obtained from the sample count of its associated source line. The basic block sample count (BB.count) is then computed from its associated WHIRL statements as shown below:

$$BB.count = \frac{\sum_{i=1}^{N_{statements}} IR.count_i}{N_{statements}} \tag{1}$$

When scaling the basic block count, all statements are given the same weight - i.e., we do not differentiate the WHIRL statements by the type of operator. If different feedback data files collected with different sampling rates are used, the BB.count should be normalized to a fixed sampling rate. Note that different heuristics from the one used here can be employed to derive basic block sample counts from source code correlated samples.

$$BB.count_{norm} = BB.count * \frac{fixed\_sampling\_rate}{sampling\_rate} \tag{2}$$

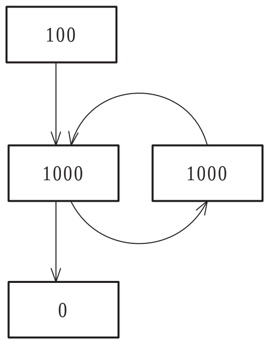

**Figure 3.** A loop with imprecise basic block sample counts

## 4.3 Edge Profiles from Sample Data

The derivation of edge frequencies and branch biases from the sampled basic block counts is a core component of our methodology. Our initial attempts at deriving edge frequency estimates based solely on the structure of the CFG yielded poor results. Take the loop in Figure 3 for example. We want to adjust the basic block samples counts to ensure consistency - i.e., the sum of the incoming edge counts should be equal to the sum of the outgoing edge counts for each basic block after this operation, which we shall refer to as “smoothing”. If we try to adjust the basic block counts solely based on the CFG structure (i.e., by setting the basic block counts to the sum of their incoming edge counts), we may increase the basic block counts within the loop indefinitely, or to a very high value based on the number of iterations executed in the smoothing algorithm for such non-converging values.

To identify and overcome such inherent problems with CFG-based smoothing algorithms, we decided to take into account the higher level program structure, such as IF and LOOP statements, and handle each case separately. This is consistent with the current model in open64, where the profile data is stored by higher level program structure. While building the CFG, extra information is stored in the ENTRY nodes of IF statements to easily identify the first node in the taken and not-taken paths, and a JOIN node is added explicitly. The detailed algorithm is shown in Figure 4.

For example, consider the C code shown in Figure 5. The CFG with basic block counts before applying the smoothing

IF statements:

1. Determine maximum sample count in ENTRY and JOIN blocks (total_sample)
2. Determine maximum basic block sample count for taken and not-taken paths (taken_sample and not-taken_sample). This will require a recursive traversal if any of the basic blocks themselves contain nested IF statements.
3. Compute the corrected total_sample count from taken_count and not-taken_count
4. Call routine `Adjust_Freq` to set all the basic block sample counts belonging to the IF statement to the appropriate total_sample, taken_sample or not-taken_sample count.

LOOP statements:

1. Determine maximum sample count in ENTRY and JOIN blocks (total_sample)
2. Determine maximum basic block sample count for the loop body (body_sample). This will require a recursive traversal if any of the basic blocks contain IF statements or nested LOOP statements.
3. Call routine `Adjust_Freq` to set all the basic block sample counts belonging to the loop body to body_sample.

**Figure 4.** Algorithm to smooth the basic block count for different structures

```
int foo(int total) {

  int *array = Get_Array();     //4000

  int count = 0;                //5000
  if (total > 0) {              //6000
    int i = 0;                  //500
    while ( i < total ) {
      if (array[i] > 0) {       //7954
        count += array[i];      //7875
        array[i] *= 3;          //7969
      }
      else {
        count -= array[i];      //34
        array[i] *= -3;         //27
      }
      i++;                      //8011
    }
    printf ("%d", i);           //420
  }
}
```

**Figure 5.** Code snippet with comments as sampling counts

heurisitics is shown in Figure 6. After smoothing, the basic block counts in the CFG are as shown in Figure 7.

The outer branch is hardly taken in the profile collection runs. As there is no "else" branch for the if statement, we don't have the count for the branch-not-taken basic block. The smoothing algorithm uses the count outside the branch and the branch-taken count to adjust the branch-not-taken count.

An important function of the `Adjust_Freq` routine is to detect cases which are inherently unreliable (in terms of source code correlated samples) and not annotate the

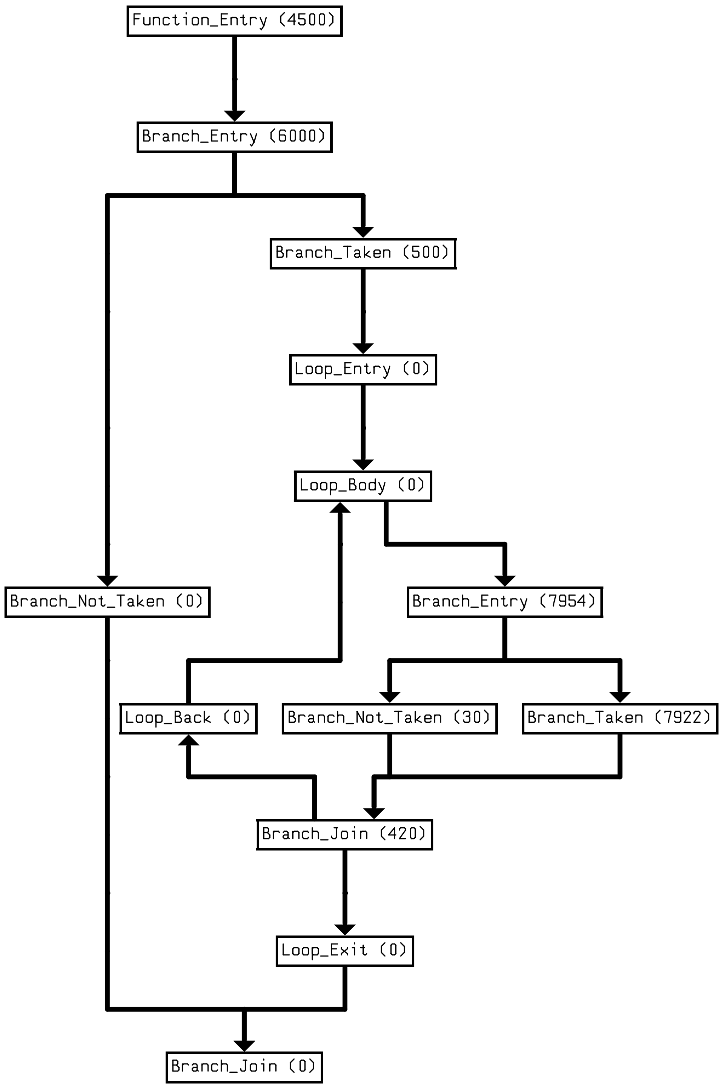

**Figure 6.** Basic block counts before applying the smoothing heuristics

branches in these cases. The following are some of the special cases to be considered:

1. `if (cond) continue;` If there are no samples for both taken and not-taken paths, and the JOIN node has only one predecessor, set sampling count of predecessor to sampling count of the JOIN node.
2. If the sum of the taken_sample and not-taken_sample differs greatly from total_sample, then don't annotate the branch.

## 4.4 Open64 Limitations

We could improve the quality of the edge profiles estimated by augmenting the heuristics to use static edge profile probabilities (Levin et al. 2008). However, the static heuristics functionality is currently implemented in the code gener-

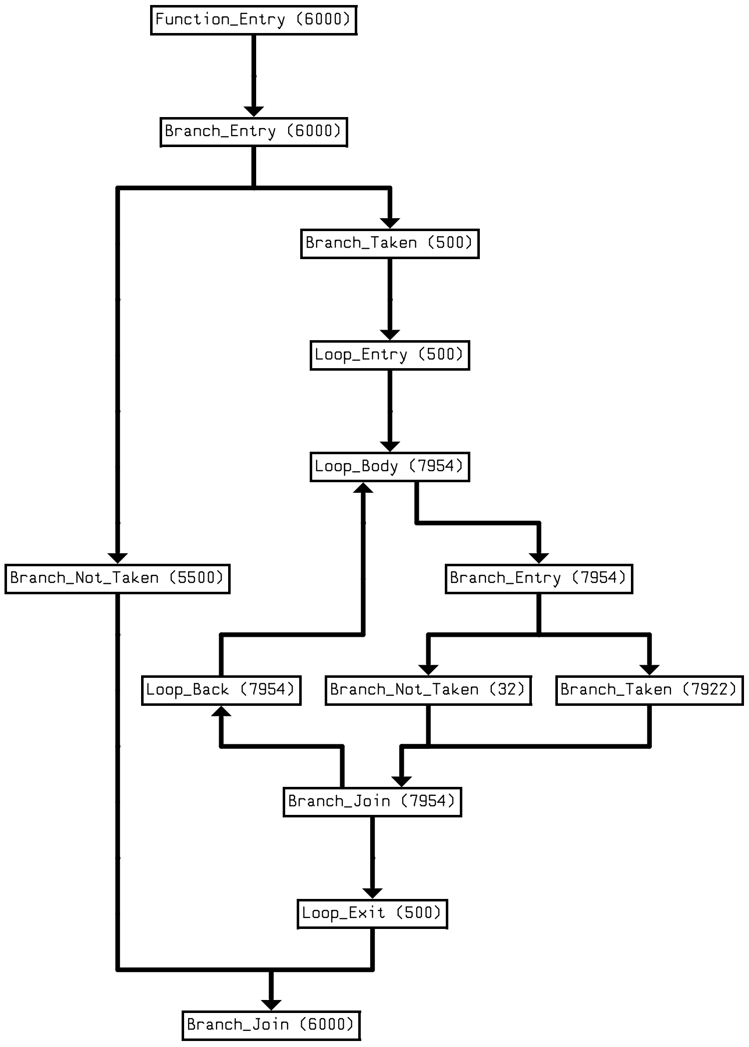

**Figure 7.** Basic block counts after applying the smoothing heuristics

ation (CG) phase in open64. Therefore, it is not straightforward, implementation-wise, to obtain edge probabilities from static heuristics to aid in edge frequency estimation during CFG annotation, which is done right after the FE phase.

Open64 uses the profile feedback data in the LNO phase to identify loops with short trip counts in order to skip aggressive loop transformations for such loops. In the WOPT phase, the frequency information is incorporated into its basic block structure, and is used in addition to other information to decide when it is advantageous to perform speculative code motion. In the VHO phase, the profile data is used to determine the best way to handle case statments. Under `-ipa` option for inter-procedural builds, inlining is done in the IPA phase, and the profile information is used for inlining decisions. However, without the `-ipa` option, the profile data cannot be used for inlining decisions. Currently early inlining is done before the instrumentation/annotation phase in the traditional FDO model. With the use of sampling profiles, there is no requirement for tight coupling between the profile collection and FDO builds. We are investigating using the profile data (separated by caller functions for inlined function instances) for making inlining decisions during the FDO build.

## 4.5 Source Correlation Issues

Since we use source position information to correlate samples with their corresponding source lines, it is important that the source position information is accurate and complete in the binaries used for profile collection. We ran into a few gcc source correlation issues with optimized (-O2) binaries - an example is shown here. Consider the following samples attributed to a hot basic block in procedure `new_dbox()` in the SPEC benchmark 300.twolf. Sample counts are shown as comments.

```
93  if( netptr->flag == 1 ) { //31366
94      newx = netptr->newx ; //3000
95      netptr->flag = 0 ;    //37000
96  } else {
97      newx = oldx ;
98  }
```

No samples are attributed to lines 96 and 97, so our heuristics predict the branch at line 93 as always taken. However, instrumented runs show that the "if statement" on line 93 is taken only 19% of the time.

The reason for no samples being attributed to lines 96 and 97, is the following transformations during optimization in gcc.

1. Initial basic block corresponding to line 97:

```
<bb 7>:
[dimbox.c : 97] newx_25 = oldx_22;
<bb 8>:
# newx_3=PHI<newx_24(6), newx_25(7)
```

2. After copy propagation into the PHI node:

```
<bb 7>:
[dimbox.c : 97] newx_25 = oldx_22;
<bb 8>:
 # newx_3=PHI<newx_24(6), oldx_22(7)>
```

3. Now the copy in bb_7 is dead and therefore eliminated during the dead code elimination phase.
4. The bb_7 is then regenerated from the PHI node when transitioning out of SSA. However, the corresponding source position information is lost at this stage.

```
<bb 7>:
 newx = oldx;
 goto <bb 9>;
```

We see a similar problem in the 175.vpr binary compiled with -O2 -g in function `get_non_updateable_bb()`. gcc is currently being enhanced to maintain source position information across copy propagation into PHI nodes and regeneration from PHI nodes in order to fix this issue.

### 4.6 Measures for Edge Profile Quality

The quality of the estimated edge profiles depend both on the quality of the sample profiles and the effectiveness of the heuristics. We use the term branch bias to denote a branch's taken probability. For example, if a branch is taken 30% of the time, then it's branch bias is 0.3, and if a branch is taken 85% of the time, it's branch bias is 0.85. As a measure of the estimated edge profile quality, we compare each branch bias obtained by sample profiling with the corresponding branch bias of the real run (obtained by instrumentation based profiling). We define the Weighted Difference (WD) measurement as follows:

| Benchmark | -O2 P | -O2 NP | -O2 NP/NR |
|---|---|---|---|
| 164.gzip | 0.15 | 0.12 | 0.13 |
| 175.vpr | 0.28 | 0.31 | 0.27 |
| 176.gcc | 0.17 | 0.38 | 0.22 |
| 181.mcf | 0.25 | 0.39 | 0.25 |
| 186.crafty | 0.27 | 0.25 | 0.27 |
| 197.parser | 0.28 | 0.45 | 0.28 |
| 253.perlbmk | 0.22 | 0.23 | 0.22 |
| 254.gap | 0.31 | 0.39 | 0.34 |
| 255.vortex | 0.25 | 0.21 | 0.24 |
| 256.bzip2 | 0.20 | 0.16 | 0.19 |
| 300.twolf | 0.22 | 0.24 | 0.10 |

**Figure 8.** Weighted Differences (WD) using -O2 binaries and different sampling modes for profile collection, P = PEBS, NP = Non-PEBS with randomization, NP/NR = Non-PEBS without randomization.

$$WD = \frac{\sum_{i=1}^{n} abs(samp\_bias_i - instr\_bias_i) * count_i}{\sum_{i=1}^{n} count_i} \tag{3}$$

The WD measurements for the Spec2000int C benchmarks are shown in Figure 8. We also define the Simple Weighted Difference (SWD) measurement, where the difference between the sample and instrumented branch bias is set to 0 if the branch biases are in the same direction (i.e., taken or not-taken), and set to 1 if the branch biases are in opposite directions.

$$SWD = \frac{\sum_{i=1}^{n} diff_i * count_i}{\sum_{i=1}^{n} count_i} \tag{4}$$

In both the WD and the SWD measurements, the differences in branch biases are weighted by the execution count of the branch. The SWD measurements for the Spec2000int C benchmarks are shown in Figure 9.

Lower WD and SWD measures imply better branch bias estimates. A random branch bias prediction is a random assignment of branch biases, without using any information about the branch. The WD measurement using sample profiles is less than 0.25 in most cases for the SPEC2000int C benchmarks used, which is much better than the the average WD measurement of 0.33 using random branch bias predictions. Similarly, the SWD measurement is less than 0.35 in most cases which is much better than the average SMD measurement of 0.50 using random branch bias predictions. However, benchmarks such as 186.crafty and 197.parser show much larger average WD and SWD measures with sample profiles. These benchmarks have short code sequences (1 or less source code lines) in the body of the taken and not_taken paths. Such short code sequences expose source code correlation issues (incorrect source position information in the binaries, or incorrect attribution of

| Benchmark | -O2 P | -O2 NP | -O2 NP/NR |
|---|---|---|---|
| 164.gzip | 0.15 | 0.21 | 0.20 |
| 175.vpr | 0.35 | 0.39 | 0.37 |
| 176.gcc | 0.14 | 0.39 | 0.19 |
| 181.mcf | 0.13 | 0.27 | 0.28 |
| 186.crafty | 0.41 | 0.34 | 0.54 |
| 197.parser | 0.31 | 0.50 | 0.31 |
| 253.perlbmk | 0.25 | 0.28 | 0.27 |
| 254.gap | 0.39 | 0.51 | 0.44 |
| 255.vortex | 0.24 | 0.22 | 0.29 |
| 256.bzip2 | 0.07 | 0.15 | 0.19 |
| 300.twolf | 0.28 | 0.36 | 0.19 |

**Figure 9.** Simple Weighted Differences (SWD) using -O2 binaries and different sampling modes for profile collection, P = PEBS, NP = Non-PEBS with randomization, NP/NR = Non-PEBS without randomization.

samples to the source lines due to sampling skew), resulting in bad branch bias estimation by the smoothing heuristics.

## 5. Experimental Evaluation

Our experiments were carried out using 64-bit binaries of the SPEC2000int and SPEC2006 C benchmarks on Intel Core-2 machines. We compare the performance gains of open64 FDO using sample profiles collected in PEBS mode (sampling every 202001 INST_RETIRED events) and non-PEBS mode (sampling rate 280001, using randomization between 0 to 255 for every sample) using gcc-built -O2 binaries vs. open64 FDO using profiles collected from instrumented runs. All performance gains are compared against default open64 -O2 runs without FDO. We also show the performance results of FDO on 32-bit platforms using samples collected with gcc built -O0 -g binaries for comparision to highlight the potential performance gains possible for FDO using sampled profiles in the absence of source correlation issues.

In open64's support for traditional FDO, the instrumentation and annotation is done after early inlining, so that the profile data collected for inlined functions can be correctly matched during annotation. We are currently enhancing our heurisitics to use sample profiles for inlining decisions. Since this is work in progress, we have focused on the set of C benchmarks for this paper, omitting C++ benchmarks with inlined functions.

### 5.1 Results

The first set of results (Figure 10 and Figure 11) is for sampled profiles collected using -O2 binaries. FDO runs using sample profiles collected in 3 different modes (PEBS, non-PEBS with randomization and non-PEBS without randomization) are compared with the base -O2 run, and traditional FDO run, which uses profile data collected from instrumented runs.

Figure 10 shows that the average % gain for the Spec2000int C benchmarks using -O2 binaries for profile collection is as follows: PEBS - 2.42%, Non-PEBS with randomization - 1.44%, Non-PEBS without randomization - 1.62%, Instrumented - 7.29%. In PEBS mode, 175.vpr and 300.twolf show performance degration due to the source correlation issues discussed in section 4.5. Omitting these two benchmarks, the average % gain for PEBS mode rises to 3.33%. The average % gain using PEBS mode is higher than the average % gain using non-PEBS mode, as expected. However, when comparing the % gains for the non-PEBS modes with and without randomization, we see that the % gain is higher without randomization (1.62% and 2.14% vs. 1.44% and 1.59%), which is contrary to our expectations.

Figure 11 shows the average % gain for the Spec2006 C benchmarks using -O2 binaries for sample profile collection: PEBS = 2.04%, non-PEBS (with randomization) = 2.66%, Instrumented = 4.48%. Here the non-PEBS mode yields better performance gains compared to the PEBS mode.

We do not find much correlation between the WD/SWD measurements shown in the previous section and the performance gains using the same feedback data files shown in Figure 10. This is reasonable as the performance gains can be attributed to the following:

1. Quality of basic block count annotation
2. Quality of branch bias annotation
3. Quality of loop annotation
4. Relative importance of the annotations for each of the above. For example, even if we predict the biases of 99 of the 100 branches correctly, we might still not get the expected performance gains if the single branch that is incorrectly predicted is the most important one in terms of performance impact.

Of the 4 items mentioned above, WD and SMD only model the branch bias annotation quality. A metric to accurately model the performance gains will have to take into account all of the 4 items mentioned above.

The results for 32-bit Spec2006 FDO runs, using -O0 binaries for profile collection are shown in Figure 12. This shows an average % gain of 3.59% for PEBS mode, 3.07% for non-PEBS mode with randomization, and 4.23% for FDO using instrumented runs. We see that when -O0 binaries are used for profile collection, the performance gains of FDO using sampled profiles collected in PEBS mode is comparable to performance gains with traditional FDO, achieving approximately 85% (3.59% vs. 4.23%) of the performance gains.

From our experimental results, there is no clear winner among the three different sampling modes (PEBS, non-PEBS with/without randomization) to collect sample profiles. We would recommend using the PEBS mode since it allows a higher sampling rate then using non-PEBS mode for a given sampling overhead.

| Benchmark | Base | PEBS | Non-PEBS | NP/NR | Instr | P gain | NP gain | NP/NR gain | I gain |
|---|---|---|---|---|---|---|---|---|---|
| 164.gzip | 1274 | 1313 | 1315 | 1308 | 1305 | 3.06 | 3.22 | 2.67 | 2.43 |
| 175.vpr* | 1450 | 1424 | 1460 | 1448 | 1483 | -1.79 | 0.69 | -0.14 | 2.28 |
| 176.gcc | 1740 | 1778 | 1761 | 1800 | 1862 | 2.18 | 1.21 | 3.45 | 7.01 |
| 181.mcf | 1431 | 1691 | 1574 | 1488 | 1681 | 18.17 | 9.99 | 3.98 | 17.47 |
| 186.crafty | 2224 | 2255 | 2177 | 2262 | 2220 | 1.39 | -2.11 | 1.71 | -0.09 |
| 197.parser | 1118 | 1142 | 1125 | 1123 | 1334 | 2.24 | 0.63 | 0.45 | 19.32 |
| 253.perlbmk | 2050 | 2107 | 2136 | 2209 | 2292 | 2.78 | 4.20 | 7.76 | 11,80 |
| 254.gap | 1786 | 1776 | 1781 | 1791 | 1847 | -0.56 | -0.28 | 0.28 | 3.42 |
| 255.vortex | 2039 | 2051 | 2020 | 1991 | 2204 | 0.59 | -0.93 | -2.35 | 8.09 |
| 256.bzip2 | 1622 | 1624 | 1595 | 1643 | 1674 | 0.12 | -1.66 | 1.29 | 3.21 |
| 300.twolf* | 2241 | 2207 | 2260 | 2212 | 2271 | -1.52 | 0.85 | -1.29 | 1.39 |
| Mean | | | | | | 2.42 | 1.44 | 1.62 | 7.29 |
| Mean w/o * | | | | | | 3.33 | 1.59 | 2.14 | 8.65 |

**Figure 10.** Runtime comparision for 64-bit Spec2000INT (Numbers are SPEC scores, higher is better). Sampled Profiles were collected with -O2 binaries. Mode of profile collection: P - PEBS, NP - Non-PEBS, NR - No Randomization, I - Instrumentation. Arithmetic mean used for average %gain. *Source correlation issues for 175.vpr and 300.twolf are explained in section 4.5.

| Benchmark | Base | PEBS | Non-PEBS | Instr | P % gain | NP % gain | I % gain |
|---|---|---|---|---|---|---|---|
| 400.perlbench | 13.9 | 14.8 | 14.9 | 16.8 | 6.47 | 7.19 | 20.86 |
| 401.bzip2 | 12.4 | 12.1 | 12.1 | 12.9 | 2.42 | -2.42 | 4.03 |
| 403.gcc | 12.2 | 12.3 | 12.4 | 12.3 | 0.82 | 1.64 | 0.82 |
| 429.mcf | 12.1 | 12.6 | 13.4 | 12.7 | 4.13 | 10.74 | 4.96 |
| 445.gobmk | 12.2 | 12.4 | 12.4 | 31.1 | 1.64 | 1.64 | 7.38 |
| 456.hmmer | 14.6 | 14.6 | 14.4 | 14.7 | 0.00 | -1.40 | 0.68 |
| 458.sjeng | 13.6 | 13.9 | 13.9 | 14.0 | 2.21 | 2.21 | 2.94 |
| 464.h264ref | 19.3 | 19.2 | 19.2 | 19.6 | -0.52 | -0.52 | 1.55 |
| 433.milc | 10.0 | 10.5 | 10.5 | 10.0 | 4.76 | 4.76 | 0.00 |
| 470.lbm | 13.8 | 14.2 | 14.2 | 14.3 | 2.90 | 2.90 | 3.62 |
| 482.sphinx3 | 16.1 | 16.5 | 16.5 | 16.5 | 2.48 | 2.48 | 2.48 |
| Mean | | | | | 2.04 | 2.66 | 4.48 |

**Figure 11.** 64-bit Spec2006 results. Sample profiles collected using -O2 binaries. Mode of profile collection: P - PEBS, NP - Non-PEBS, I - Instrumentation. Arithmetic mean used for average % gain.

| Benchmark | Base | PEBS | Non-PEBS | I | P(%) | NP(%) | I(%) |
|---|---|---|---|---|---|---|---|
| 400.perlbench | 13.2 | 14.6 | 14.4 | 16.1 | 10.61 | 9.09 | 21.97 |
| 401.bzip2 | 11.1 | 11.1 | 11.1 | 11.6 | 0 | 0 | 4.5 |
| 403.gcc* | | | | | | | |
| 429.mcf | 16.3 | 17.9 | 17.3 | 16.7 | 9.82 | 6.13 | 2.45 |
| 445.gobmk | 12.1 | 12.3 | 12.5 | 12.7 | 1.65 | 3.31 | 4.96 |
| 456.hmmer | 11.9 | 11.8 | 11.8 | 11.7 | -0.84 | -0.84 | -1.68 |
| 458.sjeng | 12.2 | 12.9 | 12.9 | 12.7 | 5.74 | 5.74 | 4.1 |
| 464.h264ref | 18.6 | 18.8 | 19.1 | 18.9 | 1.08 | 2.69 | 1.61 |
| 433.milc | 8.57 | 8.63 | 8.76 | 8.55 | 0.12 | 0.93 | -0.23 |
| 470.lbm | 11.6 | 12.1 | 12.1 | 11.9 | 4.31 | 4.31 | 2.59 |
| 482.sphinx3 | 14.8 | 15.1 | 14.7 | 15.1 | 3.38 | -0.68 | 2.03 |
| Mean | | | | | 3.59 | 3.07 | 4.23 |

**Figure 12.** 32-bit SPEC2006 results. Sample profiles collected using -O0 binaries. Mode of profile collection: P - PEBS, NP - Non-PEBS, I - Instrumentation. *403.gcc had runtime failures when built with FDO so results are not available.

## 6. Related Work

In a recent paper, Roy Levin, Ilan Newman, and Gadi Haber (Levin et al. 2008) use sampled profiles of the INST_RETIRED hardware event to construct edge profiles for FDO in IBM's FDPR-Pro, *post-link time* optimizer. The samples are directly correlated to the corresponding basic blocks without the need to use any source position information, as this is done post-link time. The problem of constructing a full edge profile from basic block sample counts is formalized as a Minimum Cost Circulation problem. Here, the flow conservation rule is that for each vertex in the CFG, the sum of the incoming edge frequency counts should be equal to the sum of the outgoing edge frequency counts.The idea is that by ensuring the flow conservation rule, and at the same time, limiting the amount of weighted change from the initial edge weights predicted by static profiles (Wu and Larus 1994) to a mininum, a near approximation to actual edge counts obtained via instrumentation can be achieved.

An algorithm to solve the above Minimum Cost Circulation problem is described in (Goldberg and Tarjan 1989) and is based on repeatedly finding a residual cycle with negative cost and canceling it by pushing enough flow through the cycle to saturate an arc. While this is a higher order polynomial time algorithm, the compile time was found to be within acceptable levels for the SPEC benchmarks. Experimental results (Levin et al. 2008) show that -O3 FDO runs using edge profiles from sampling profiles are able to achieve nearly 100% of the performance gains of -O3 FDO runs using exact edge profiles from instrumented runs. An interesting observation is that a large percentage of the performance benefits (70% to 80%) can be obtained using the initial edge profile estimation predicted by static profiles and the sampled basic block counts alone. The gcc compiler already has support for using static heuristics and its implementation model makes it easier for us to experiment with using this algorithm. Please note the differences between the approach of Levin et. al., and what is described in this paper.

1. We are working on source level, whereas the work by Levin et. al., is on the binary level.
2. The platforms used are different - Out-of-order x86 vs. IBM's POWER4/AIX platforms.
3. Our approach allows using binaries built by any compiler for the purpose of profile collection.

Another hardware-based profiling technique (Conte et al. 1996) is to sample the contents of the branch-prediction hardware, namely

1. The target address, i.e., the destination of the edge,
2. The buffer tag, i.e., the source of the edge, and
3. The prediction information, i.e., the edge's weight

using kernel-mode instructions. This mechanism requires the program to be compiled with a special identity token indicating that it contains a table of CFG edges. During execution, the kernel periodically reads the hardware branch-prediction buffer and updates the edge table stored in the executable. The profiling overhead is estimated to be between 0.4% and 4.6%. The basic block counts are then estimated from the sampled edge weights.

Other methods of edge profile estimation build on ideas from both program instrumentation and statistical sampling. In (Traub et al.), an approach for estimation of traditional edge profiles using ephemeral instrumentation is described. A branch's bias is sampled by periodically inserting instrumentation code to capture a small and fixed number of the branch's executions. A post-processing step is used to derive traditional edge profiles from the ephemeral branch biases collected. The problem of obtaining a weighted CFG from a CFG annotated with branch biases is equivalent to the problem of finding the limiting probabilities on an irreducible, finite-state Markov chain. Their experimental results show that the ephemeral profiles show competitive performance gains when compared with using complete edge profiles to drive a superblock scheduler.

A similar framework for performing instrumentation sampling with low overhead is described in (Gloy et al.). Their sampling technique does not rely on any hardware or operating system support, but performs code duplication and uses compiler-inserted counter-based sampling to switch between instrumented and non-instrumented code in a controlled, fine-grained manner.

The Morph system (Zhang et al. 1997) is a framework for automatic collection of profiles via statistical sampling of the program counter on clock interrupts. It also provides a framework for profile data management and dynamic profile-driven optimizations. Since optimizations are applied without the use of source code, the Morph system requires the executable to also contain the extra information pertaining to the compiler intermediate representation, which is used to map the instruction-level samples to their corresponding basic blocks. This solution has a major drawback in that it requires new software standards and standards compliances across the software industry. Their use of a time-based sampling also skews the basic block counts towards higher latency instructions, which is mitigated in our method of sampling the INST_RETIRED event.

Stack sampling has been used, without the use of any instrumentation, to implement a low-overhead call path profiler (Froyd et al. 2005). This method requires marking a stack frame with a sentinal to reduce the overhead of constructing the call paths from the stack samples. This method could be used to augment the sample profiles collected by INST_RETIRED event sampling.

## 7. Conclusions

We presented a new methodology of using INST_RETIRED hardware event sampling to construct basic block and edge frequency profiles that are used to guide feedback directed optimizations at compile-time. This method overcomes the many shortcomings of the traditional FDO usage model, and initial experimental results show promising performance gains. We expect to get further improvements in performance by continued tuning and enhancement of the heuristics described in this paper.

## Acknowledgments

We would like to thank Brad Chen for providing us with machine resources, Stephane Eranian for enthusiastically helping us resolve perfmon related issues, Martin Thuresson for bringing to our attention the very relevant research by Roy Levin et. al., Seongbae Park for help in analysis of GCC source correlation issues, and our colleagues Preston Briggs, Mark Heffernan, and David Li, as well as the anonymous reviewers for their very valuable review feedback.

## References

Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, 1997. URL `citeseer.../ammons97exploiting.html`.

J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone. Technical report.

Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994. URL `citeseer.../ball94optimally.html`.

Thomas Ball and James R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996. URL `citeseer.ist.psu.edu/ball96efficient.html`.

Thomas M. Conte, Burzin A. Patel, Kishore N. Menezes, and J. Stan Cox. Hardware-based profiling: An effective technique for profile-driven optimization. *International Journal of Parallel Programming*, 24(2):187–206, 1996. URL `citeseer.ist.psu.edu/30422.html`.

Nathan Froyd, John Mellor-Crummey, and Rob Fowler. Low-overhead call path profiling of unmodified, optimized code. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 81–90, New York, NY, USA, 2005. ACM. ISBN 1-59593-167-8. doi: http://doi.acm.org/10.1145/1088149.1088161.

N. Gloy, Z. Wang, C. Zhang, B. Chen, and M. Smith. Profile-based optimization with statistical profiles. Technical report.

Andrew V. Goldberg and Robert E. Tarjan. Finding minimum-cost circulations by canceling negative cycles. *J. ACM*, 36(4):873–886, 1989. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/76359.76368.

Intel. *Ia-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming.* Intel Press, 2007.

Roy Levin, Ilan Newman, and Gadi Haber. Complementing missing and inaccurate profiling using a minimum cost circulation algorithm. In *HiPEAC*, pages 291–304, 2008.

Matthew C. Merten, Andrew R. Trick, Christopher N. George, John C. Gyllenhaal, and Wen mei W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *ISCA*, pages 136–147, 1999. URL `citeseer.../merten99hardwaredriven.html`.

Florian T. Schneider, Mathias Payer, and Thomas R. Gross. Online optimizations driven by hardware performance monitoring. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 373–382, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: http://doi.acm.org/10.1145/1250734.1250777.

O. Traub, S. Schechter, and M. Smith. Ephemeral instrumentation for lightweight program profiling. Technical report.

Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. Technical Report CS-TR-1994-1248, 1994. URL `citeseer.../wu94static.html`.

Catherine Xiaolan Zhang, Zheng Wang, Nicholas C. Gloy, J. Bradley Chen, and Michael D. Smith. System support for automated profiling and optimization. In *Symposium on Operating Systems Principles*, pages 15–26, 1997. URL `citeseer.ist.psu.edu/zhang97system.html`.