Reprinted from the

Proceedings of the GCC Developers' Summit

June 17th–19th, 2008
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Ben Elliston, *IBM*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Google*
Gerald Pfeifer, *Novell*
Ian Lance Taylor, *Google*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

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

Vinodha Ramasamy
Google Inc.
vinodha@google.com

Paul Yuan
Peking University
yingbo.com@gmail.com

Dehao Chen
Tsinghua University
danielcdh@gmail.com

Robert Hundt
Google Inc.
rhundt@google.com

Abstract

Traditional feedback-directed optimization (FDO) in GCC uses static instrumentation to collect edge and value 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 competitive performance gains. We describe the motivation, design, and implementation of FDO using sample profiles in GCC and also present our initial experimental results with SPEC2000int C benchmarks that show approximately 70% to 90% of the performance gains obtained using traditional FDO with exact edge profiles.

1 Introduction

This paper is a continuation of our previous work [13]. We have reproduced with minor modifications and slightly extended the introduction. Readers familiar with the motivation for this work may skip directly to Section 3.

GCC uses execution profiles consisting of basic block and edge frequency counts to guide optimizations such as instruction scheduling, basic block re-ordering, function splitting, and register allocation. The current method of feedback-directed optimization in GCC (shown in Figure 1) involves the following steps:

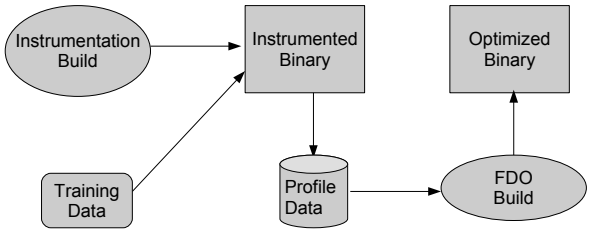


Figure 1: Traditional FDO Model

1. Build an instrumented version of the program for edge and value profiling.
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% [3] [2], 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 and FDO builds are tightly coupled. GCC requires that both builds use the same inline decisions and similar optimization flags 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.

To overcome the limitations of the current FDO model, we propose skipping the instrumentation step altogether. Instead, we 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, Itanium) to obtain estimated edge profiles. 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 sample 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 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 by sampling of 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. This is similar to the instrumentation-based FDO model, except that the overhead of profile collection is much lower.

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.

4. The current instrumentation-based FDO model does not support obtaining execution counts for kernel code, as the counters are written out at application/process exit time. Sample-based profile collection is therefore an apt choice to enable FDO for kernel code.

The sample 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 [13], we use GCC-built binaries for profile collection and open64 for FDO builds and performance experiments. In this paper, we focus on FDO support for sample profiles in GCC.

In general, deriving *exact* basic block and edge frequency counts from sample profiles is not always feasible [12]. We use heuristics to derive *relative* basic block and edge frequency count estimates from the sample profiles. We've found that these approximations are sufficient for all practical purposes.

Increasing the sampling rate will in general increase the quality of the sample profile at the expense of increasing the overhead of profile collection. Our experiments show that we can get sample profiles with reasonable quality with overheads of less than 2%.

We use a *degree of overlap* measure [9] which compares the relative edge weights between the edge profiles constructed from instrumented runs and sample profiles as an indicator of the quality of the sample profiles and the heuristics used. However, the definitive measure of the sample profile quality and effectiveness of the heuristics employed is ascertained only from the performance gains obtained in using the sample profiles for feedback-directed optimizations.

In open64, our edge count estimation algorithm used higher level program constructs such as branches and loops for recursively smoothing the basic block sample counts [13]. Levin *et al*. [9] describe another algorithm used in IBM's post-link time optimizer, FDPR-Pro, for deriving edge profile estimates from basic block sample counts. Our task is more challenging since we need to rely on source correlation to attribute samples to basic blocks, since the feedback is done at compile-time rather than post-link time. However, the edge estimation algorithm described in [9] is directly applicable to our sample profile support in GCC.

The source line execution metrics collected via sampling are mostly platform independent, so the profile data collected on one platform can be used to build a binary optimized for another platform. We use the Intel Core-2 platform for profile collection and the AMD Opteron platform for our performance runs. Since the profile data is stored by samples per source line, it does not matter if the profile collection is done using optimized or unoptimized binaries in most cases. 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 C benchmarks, we currently obtain an average performance gain of 2.13% (2.46% if only a subset of the edge profile specific options are enabled) using FDO with sample profiles collected using -O2 binaries, as compared to an average of 2.94% using traditional FDO runs with edge profiles alone. We expect to get improved results with better source correlation support in GCC. Using -O0 binaries for profile collection, we are able to achieve an average performance gain of 2.52%, which is approximately 86% of the performance gains seen using traditional FDO with edge profiling.

The rest of the paper is organized as follows: Section 2 gives a background of hardware event sampling. Section 3 describes the design and algorithms used for sample profile support in the GCC compiler. Section 4 gives a background of current instrumentation-based FDO support in GCC and then describes the implementation details for adding sample profile-based FDO support. Section 5 discusses challenges faced and open issues. Section 6 describes the experimental evaluation of using FDO with sample profiles. Finally, Section 7 discusses current status and future work for support of FDO with sample profiles in GCC.

2 Hardware Event Sampling

Most modern microprocessors support hardware event sampling, which works as follows: the Program Counter (PC) and other register contents are recorded whenever a specified number of the hardware event of interest has occurred. This helps to identify the program locations, i.e., the instruction addresses incurring the measured hardware event. For example, the DCPI tool [1] 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 [14]. 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 therefore use sampling of the INST_RETIRED event for our execution profile estimation.

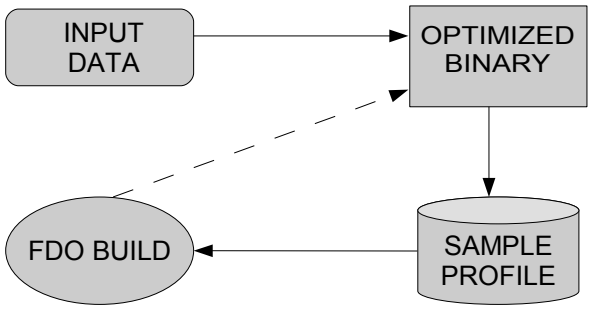


Figure 2: FDO Model with Sample Profiles

3 Design

In our FDO model using sample profiles (see Figure 2), the instrumentation step is skipped altogether. Instead, INST_RETIRED event samples gathered using profiling tools such as `perfmon2/pfmon` are used to create the feedback data. The 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. Consider two source lines, $S1$ and $S2$, in the same basic block which have identical execution counts. If 5 assembly instructions are generated for $S1$ and 10 for $S2$, then $S2$ will have approximately twice the total number of samples of $S1$—i.e., source lines with larger number of instructions will have correspondingly larger total number of samples attributed. Therefore, the total number of samples attributed per source line is divided by the number of contributing instructions to derive the

[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.

average number of samples per source line, which is stored in the feedback data file. In the example below, the sample count attributed to each individual instruction of the source line `pbla.c:60` is shown in the first column of the disassembly code. The sample count derived for this source line is 70 as shown.

```
pbla.c:60 iplus = iplus->pred;
// (100 + 30 + 70 + 80)/ 4 = 280/4 = 70

100 : 804a8b7: mov  0x10(%ebp),%eax
30  : 804a8ba: mov  0x8(%eax),%eax
70  : 804a8bd: mov  %eax,0x10(%ebp)
80  : 804a8c0: jmp  804a94b <this+0x137>
```

The feedback file is read into GCC, and is used to annotate the IR statements for the current program unit with the relative execution counts of the corresponding source position information (`IR.count`). This is done in the same pass (`pass_tree_profiling`) as the original GCC profile instrumentation/annotation for instrumentation-based FDO. The basic block sample count (`BB.count`) is then computed from its associated IR statements as shown below:

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

When scaling the basic block count, all statements are given the same weight—i.e., we do not differentiate the IR statements by the type of operator. If different feedback data files collected with different sampling rates are used, the basic block count should be normalized to a fixed sampling rate.

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

Note that different heuristics from the one used here can be employed to derive basic block sample counts from source-code-correlated samples. The basic block counts are then used to derive edge frequency counts using heuristics which are described in more detail in Section 3.1.

At the end of this pass, GCC internal data structures will be initialized appropriately with estimated basic block

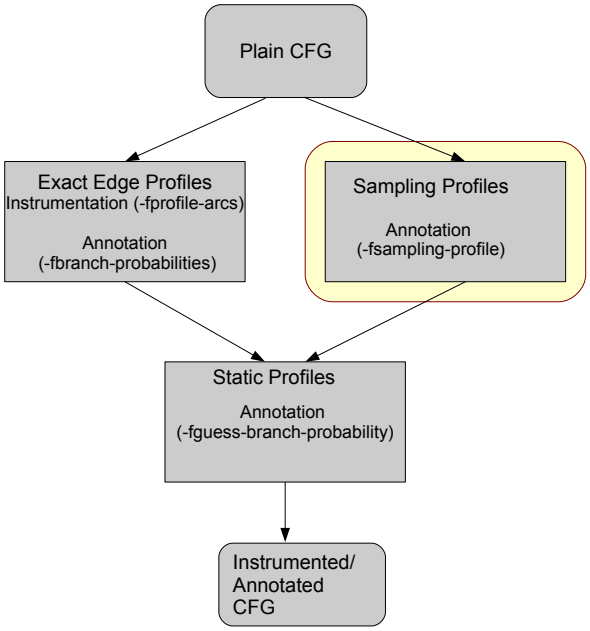


Figure 3: Constructing an Instrumented/Annotated CFG

and edge frequency counts determined from the sample profile data, in a manner similar to what is done when using instrumentation-based profile data. It should therefore not matter to later optimization phases whether the feedback data was collected via sampling or via instrumentation. This makes the design and implementation modular, and helps to leverage existing feedback-based optimization methods, and support in GCC to maintain, propagate, and verify the feedback data.

3.1 Edge frequency estimation

The derivation of edge frequencies from the basic block sample counts is a core component of the sample profile support. We use the edge estimation algorithm outlined in [9], which formalizes the problem as a minimum-cost circulation problem [7]. In this case, the flow conservation rule is that for each vertex in a procedure's 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 [2] to a mininum, a near approximation to actual edge counts obtained via instrumentation can be achieved.

The minimum-cost circulation problem is equivalent to a minimum-cost maximal flow problem. To formulate

the problem of computing the intra-procedural edges as a minimum-cost, maximal-flow problem, we need to construct the following [9]:

- $G' = (V', E')$: the fixup graph
- $min(e), max(e)$: minimum and maximum capacities for flow on each edge, e in E'
- $k(e)$: confidence constant for any edge e in E'. The values are set as following in [9]:

$$b = \sqrt{avg\_vertex\_weight(cfg)} \quad (3)$$

$$k^+(e) = b \quad (4)$$

$$k^-(e) = 50b \quad (5)$$

where $k^+(e)$ is used when increasing the flow on the edge e, and $k^-(e)$ is used when decreasing the flow on edge e.

Cost coefficent function for the edges:

$$cp(e) = k'(\Delta(e))/\ln(w(e)+2) \quad (6)$$

where

$$k'(\Delta(e)) = k^+, \text{ if } \Delta(e) \geq 0,$$

$$k'(\Delta(e)) = k^-, \text{ if } \Delta(e) < 0,$$

and $w(e)$ is the initial assigned edge weight.

These values ensure that the cost of decreasing the weight on an edge is significantly larger than increasing the weight on an edge and higher confidence in an initial value of e results in a higher cost for changing the weight of that edge.

Let $G = (V, E)$ be the CFG with initial weights:

$\forall < u, v > \in E : w(< u, v >) \leftarrow w(u) * p(< u, v >)$

where $w(u)$ is the sample count of the basic block u, and $p(< u, v >)$ is the probability of the edge $< u, v >$ as determined using static profiles [2].

The algorithm to construct the fixup graph $G'(V', E')$ from $G = (V, E)$ is outlined below:

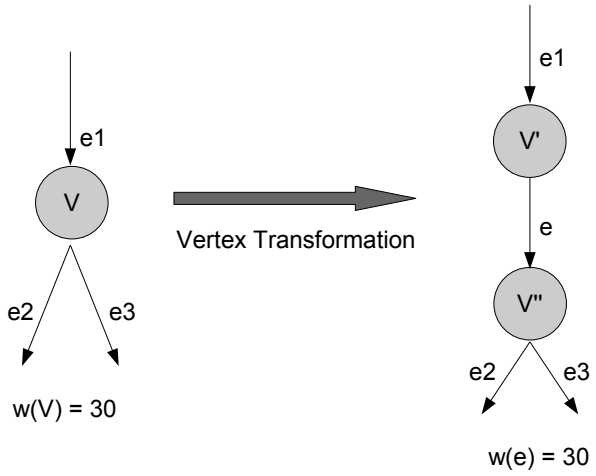
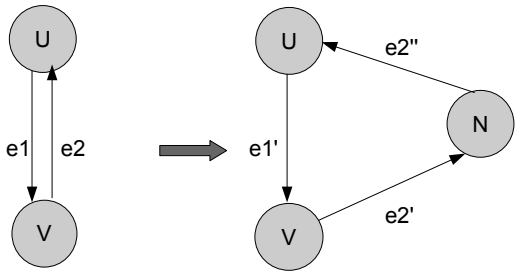


Figure 4: Vertex Transformation




```
max(e1')  = max(e1); k'(e1')  = 0.5 * k'(e1)
max(e2')  = max(e1); k'(e2')  = 0.5 * k'(e1)
max(e2'') = max(e2); k'(e2'') = k'(e2)
```

Figure 5: Normalization

1. **Vertex Transformation**
   Construct $G_t = (V_t, E_t)$ from the initial CFG $G = (V, E)$ by doing vertex transformations $\forall v \in V$. Split each vertex v into two vertices v' and v'', connected by an edge from v' to v''. The weight of the new edge $< v', v'' >$ is set to the basic block count of v. This is shown in Figure 4.
2. **Initialize**
   (a) For each vertex $v \in V_t$, let:
   $D(v) = \sum_{e_i \in out(v)} w(e_i) - \sum_{e_j \in in(v)} w(e_j)$
   (b) For each $e \in E_t$, do:
   $min(e) \leftarrow 0, max(e) \leftarrow \infty, k'(e) \leftarrow k^+(e)$
   (c) $E_r \leftarrow \emptyset, L \leftarrow \emptyset$
3. **Add Reverse Edges**
   For each $e =< u, v > \in E_t$ such that $e_r =< v, u > \notin E_t$, do:
   - Add edge e_r
   - $min(e_r) \leftarrow 0, max(e_r) \leftarrow w(e), k'(e_r) \leftarrow k^-(e)$
   - $E_r \leftarrow E_r \cup \{e_r\}$

4. **Create Single Source and Sink**
   Add a source vertex s' and connect it to all function entry vertices. Add a sink vertex t' and connect it to all function exit vertices.
   (a) $\forall s \in S$ where S is the set of function entry vertices, do:
   - Add edge $e_s =< s', s >$
   - $min(e_s) \leftarrow 0, max(e_s) \leftarrow w(s), cp(e_s) \leftarrow 0$
   - $L \leftarrow L \cup \{e_s\}$

   (b) $\forall t \in T$ where T is the set of function exit vertices, do:
   - Add edge $e_t =< t, t' >$
   - $min(e_t) \leftarrow 0, max(e_t) \leftarrow w(t), cp(e_t) \leftarrow 0$
   - $L \leftarrow L \cup \{e_t\}$

5. **Balance edges**
   For each $v \in V_t/(S \cup T)$ do:
   (a) if $D(v) \geq 0$:
   - Add edge $v_t =< v, t' >$
   - $min(v_t) \leftarrow D(v), max(v_t) \leftarrow D(v)$
   - $L \leftarrow L \cup \{v_t\}$

   (b) if $D(v) < 0$:
   - Add edge $v_s =< s', v >$
   - $min(v_s) \leftarrow -D(v), max(v_s) \leftarrow -D(v)$
   - $L \leftarrow L \cup \{v_s\}$

6. **Normalization**
   This step is needed to remove anti-parallel edges. Anti-parallel edges are created by the vertex transformation step from self-edges in the original CFG G and by the reverse edges added during Step 3.
   $\forall e =< u, v > \in E_t \cup E_r$ such that $e_r =< v, u > \in E_t \cup E_r$, do:
   (a) Add new vertex n
   (b) Delete edge $e_r =< v, u >$
   (c) Add edge $e_{vn} =< v, n >$
   $k'(e_{vn}) \leftarrow 0.5 * k' < u, v >$
   $min(e_{vn}) \leftarrow 0, max(e_{vn}) \leftarrow max(< u, v >)$
   (d) Add edge $e_{nu} =< n, u >$
   $k'(e_{nu}) \leftarrow k' < v, u >$
   $min(e_{nu}) \leftarrow 0, max(e_{nu}) \leftarrow max(< v, u >)$
   (e) $k'(< u, v >) \leftarrow 0.5 * k'(< u, v >)$
   (f) $E' \leftarrow E' \cup \{e_{vn}, e_{nu})\}, V' \leftarrow V' \cup \{n\}$
   An example of the normalization step is shown in Figure 5.

7. **Finalize**
   - $E' \leftarrow E' \cup E_t \cup E_r \cup L$
   - $V' \leftarrow V' \cup V_t$

The output of this algorithm is:

1. The fixup graph, $G' = (V', E')$
2. $\forall e \in E' : min(e), max(e), cp(e)$ – the minimum capacity, maximum capacity, and cost of each edge.

This is used as input to the minimum-cost, maximal-flow problem.

The solution of the minimum-cost, maximal-flow problem will be a flow function $f(e) \forall e \in E'$.

The fixup vector $\Delta(e) \forall e =< u, v >$ in the original edge set E is calculated as follows:

$$\Delta(e < u, v >) = f < u, v > -f < v, u > \quad (7)$$

where $< v, u >$ is the reverse edge added during the fixup graph construction.

The corrected edge weights will be calculated as follows: For each $e \in E$:

$$w^*(e) = w(e) + \Delta(e) \quad (8)$$

By mapping back the edges which were derived from the vertices in the vertex transformation step, we can determine the corrected basic block counts as well [9].

3.2 Minimum-cost Maximal Flow Algorithm

Our implementation of the minimum-cost maximal flow algorithm is based on Klein's negative cycle cancellation algorithm, shown in Figure 6.

Any edge that is not saturated is a *residual* edge. The *residual capacity* c_f of an edge $e =< u, v >$ is defined as $c_f(< u, v >) = max(e) - f(< u, v >)$.

An *augmenting path* is a path where every edge is a residual edge. The residual capacity of an augmenting path is the minimum of the residual capacity of its edges.

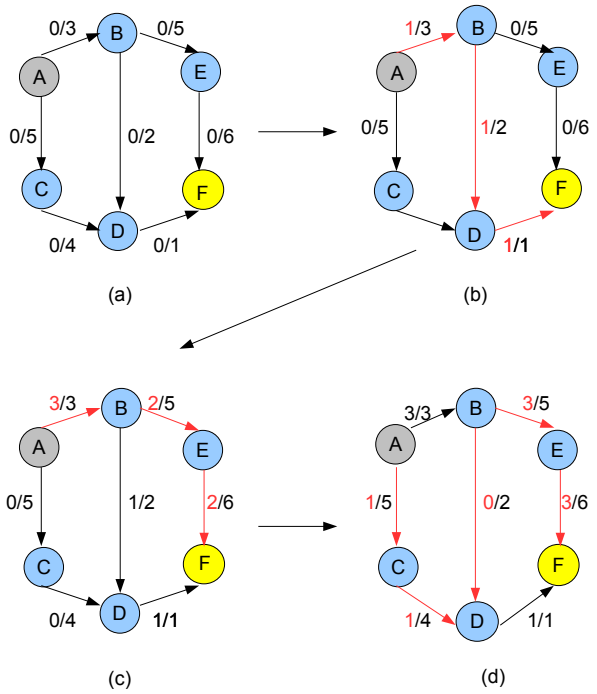
A *residual cycle* is a simple cycle of residual edges. The capacity of a residual cycle is the minimum of the residual capacities of its edges. The cost of a cycle is the sum of the costs of its edges. A residual cycle is negative if it has negative cost.

To find a maximal flow (step 1 of Figure 6), we use the Edmonds-Karp algorithm which is a specific implementation of the Ford-Fulkerson [6] method. The Edmonds-Karp algorithm uses a Breadth-First Search (BFS) to find the augmenting paths.

An example of the Edmonds-Karp algorithm is outlined in Figure 7. (a) shows the graph with initial flow of 0 (Step 1a of Figure 6). Steps (b), (c), and (d) in Figure 7 each demonstrate Steps 2b and 2c in Figure 6 and are explained in more detail below.

1. Use a maximal flow routine to find a flow f of value v for the fixup graph $G'(V',E')$ as follows:
   a Initialize flow to 0:
   $\forall < u,v > \in E' : f(< u,v >) \leftarrow 0$.
   b Find an augmenting path from source s to the sink t.
   c Send flow equal to the path's residual capacity along the edges of this path.
   d Repeat steps b and c until no new augmenting path is found.
2. Form the residual network $G_f(V',E_f)$ which is the network with capacity
   $c_f < u.v > \leftarrow max(< u,v >) - f(< u,v >)$
   $c_f(< v,u >) \leftarrow f(< u,v >)$
   The cost of each reverse edge is set as follows:
   $cp(< v,u >) \leftarrow -cp(< u,v >)$
3. Repeat: While G_f contains a negative cost cycle C, reverse the flow on the found cycle by the minimal residual capacity in that cycle.
4. Form the minimum-cost maximal flow network $G'(V',E')$ from G_f:
   $\forall < u,v > \in E' : f(< u,v >) \leftarrow c_f(< v,u >)$

Figure 6: Mimimum Cost Maximal Flow Algorithm



Edges are labeled with flow/capacity. A = source; F = sink.

Figure 7: Example for Edmonds-Karp Algorithm

(b) The augmenting path ABDF is found and flow equal to its residual capacity of 1 unit is sent through this path.

(c) The augmenting path ABEF is found and flow equal to its residual capacity of 2 units is sent through this path.

(d) The augmenting path ACDBEF is found and flow equal to its residual capacity of 1 unit is sent through this path. Note how flow is pushed back i.e., reversed along path BD. The resulting graph is a maximal-flow network.

The residual network (step 2 of Figure 6) for the example above is shown in Figure 8. This has no cycles and therefore no negative cost cycle. In this case, the maximal flow is also a minimum-cost flow.

We use the Bellman-Ford [4] algorithm to test the existence of a negative directed cycle (step 3 of Figure 6). Figure 9 illustrates the derivation of a minimum-cost maximal flow network.

(a) The maximal flow network with edges labeled with pairs (flow/capacities, cost).

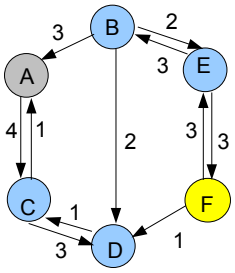


Figure 8: Residual Network

(b) Residual network derived.

(c) Negative cost cycle with minimul residual capacity of 2 units.

(d) New residual network after reversing the flow on the cycle ABCA by 2 units. No more negative cost cycles exist.

(e) Minimum-cost maximal flow network derived.

We are currently evaluating the compile-time performance of the above method. If the compile-time is not within acceptable limits, we will then implement Goldberg and Tarjan's [7] algorithm for solving the

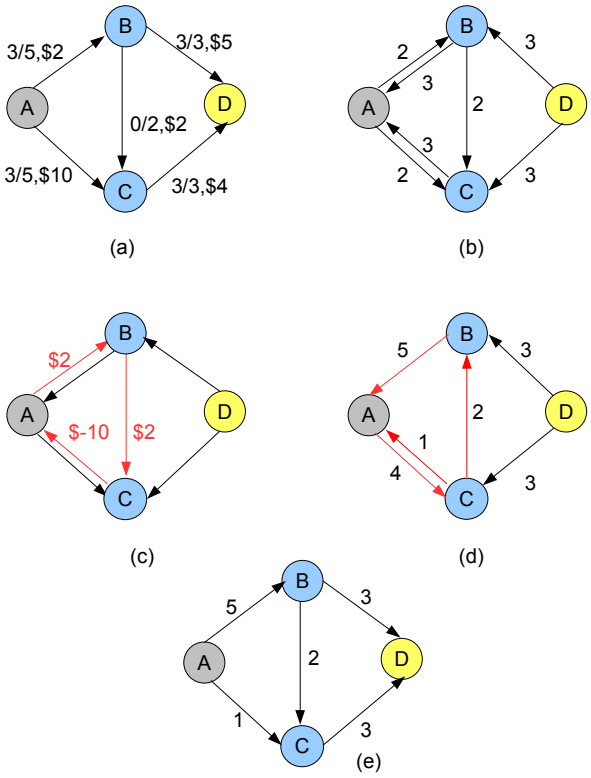


Figure 9: Cycle Canceling Algorithm

minimum-cost circulation algorithm. They propose an improvement over previous negative cycle canceling algorithms by judicious choice of the negative cycle to cancel at each step, namely the cycle with the minimum mean cost.[2]

An interesting observation from [9] is that using sample profiles in combination with static profiles to obtain initial edge frequency count estimates *without* applying the minimum-cost flow algorithm described above, is sufficient to realize a large percentage (> 70%) of the performance gain obtained by instrumentation-based FDO with exact edge profiles. However, in our experiments with the implementation of sampling-based FDO support in GCC (see Section 6), we found that it is necessary to employ the minimum-cost maximal flow algorithm to realize the performance gains.

4 Implementation

This section describes the existing implementation for instrumentation-based FDO support in GCC. An overview of the GCC stages is given in Figure 10. The stages involved in the CFG annotation with profile data are shown highlighted.

[2]The mean cost of a cycle is its cost divided by the number of edges it contains.

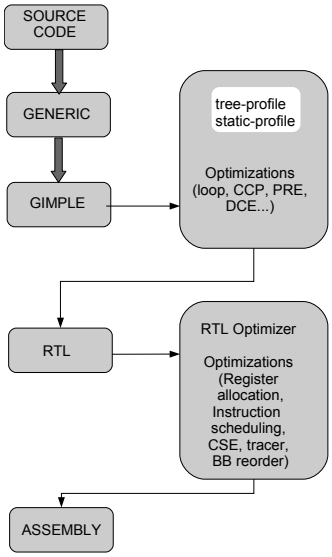


Figure 10: Overview of GCC Stages

4.1 Edge profiles

Edge profiles provide the execution count of each edge in the function CFG, which are then used to compute the basic block execution counts.

Both the TREE and RTL intermediate representations in GCC use data structures `basic_block` and `edge` to describe the CFG.

The instrumentation and annotation passes work on the TREE intermediate representation. The function `branch_prob` in `profile.c` implements these two passes.

Instrumentation

If the `-fprofile-arcs` option is specified, GCC instruments the CFG. For each function's CFG, a spanning tree is computed and counter code inserted on the non-spanning-tree edges. When the program runs, the counter code writes the edge execution counts into a profile data file (`.gcno` and `.gcda`).

Annotation

When the `-fbranch-probabilities` option is specified, GCC reads the profile data file and annotates

the CFG. All edges and basic blocks are marked with execution counts.

There is also support for synthetic profiles in GCC. When the `-fguess-branch-probability` option is specified, GCC predicts branch probabilities and estimates edge profiles using static heuristics [2].

The data structures `basic_block` and `edge` have a 64-bit integer member field `count` to record the execution count during the training run. This field is normalized to a new value in the range 0 to `BB_FREQ_MAX` and stored in the member field `frequency`. This data is used by all profile based optimizations for decision-making.

The following optimizations in GCC use edge profile data:

1. Basic block reordering (tracer)
2. Register allocation (register priority)
3. Instruction scheduling (EBB)
4. Function reordering (hot/cold, use the first basic block frequency as the function frequency)
5. Modulo scheduling (loop trip count)

4.2 Sample Profile Implementation

This section highlights the implementation details for adding sample profile support in GCC. The implementation is done on the GCC 4.3 branch.

4.2.1 Feedback datafile format

The design of the sample profile feedback data file format is based on the open64 feedback file format, where a single file is used to store profile data for an executable. The layout of the sample profile data file is given in Figure 11.

`Fb_Sample_Hdr` is the file header. The data structure `Pu_Sample_Hdr` holds the header information pertaining 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. Please note that the current debug/source position information design in GCC does not support differentiating between different instances of the same callee function inlined in a caller routine.

```
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 11: Feedback Datafile Format

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.

4.2.2 Sample profile annotation

When the new option `-fsample-profile` is enabled:

1. The sample profile datafile is read.
2. sp_annotate() is called in the new pass added `pass_sample_profiling`.

3. Feedback-directed optimizations are enabled.

`sp_annotate` is the main entry of sample profile annotation which annotates the CFG with the sample profile data.

```
# sp_annotate()
sp_read_sample_profile();
for each BB
  sp_annotate_BB();
sp_smooth_cfg();
```

`sp_read_sample_profile` reads the sample profile data to build a hash table with the set of `<source_line_number,execution_count>` mapping of samples per function.

`sp_annotate_BB` computes the basic block sample count from the sample counts of its individual IR statements.

```
# sp_annotate_BB()
long long sum_IR_count=0;
int number_IR=0;
for EACH IR
  number_IR++;
  Get IR_sample_count from hash table;
  if (IR_sample_count > 0)
    sum_IR_count += IR_sample_count;
BB.count=sum_IR_count/number_IR
```

`sp_smooth_cfg` implements the algorithm described in [9].

```
# sp_smooth_cfg()
# 1) Initialize k+(o), k-(o), w(o)
sp_initialize_cfg();
# 2) Build fixup graph G'
sp_build_fixup_graph();
# 3) Minimum cost maximal flow algorithm
sp_minimum_cost();
# 4) Fixup the graph with fixup vector
sp_fixup_graph();
# 5) Convert edge counts to freqs
counts_to_freqs()
```

5 Challenges

Our methodology has several challenges, some due to hardware-event sampling and others due to our reliance on source position information to correlate samples to basic blocks.

5.1 Sampling Issues

INST_RETIRED samples recorded per instruction may not always be representative of actual instruction execution count due to the following reasons:

Program Synchronization

It is possible for the program execution to become syncronized with the sampling rate. This will result in the same instruction being sampled, for example in the presence of program loops. In order to mitigate this problem, when sampling every `n` INST_RETIRED event, `n` should be chosen to be a prime number. Another solution to avoid program synchronization is to apply a randomization factor to every sample—this is supported in the performance monitoring hardware of some architectures (e.g., Intel Core-2 processors).

Hardware

On out-of-order execution machines, such as the x86 platform, the instruction addresses recorded during sampling may be skewed—i.e., the instruction address recorded may not be the actual instruction incurring the hardware event, and the skew distance may vary a lot. For example, on the AMD Opteron microarchitecture, there may be as many as 72 macro-ops in flight. These skews distort the results for finer-grained measurements, for example, measurements done on the granularity of basic blocks, as needed for edge profile estimation. The Intel Core-2 platform supports a Precise Event-Based Sampling (PEBS) mode [8] which accurately records the next instruction address following the instruction incurring the sampled event. We use this sampling mode for our profile collection. One drawback of this sampling mode is that it does not allow randomization of every sample, as supported for the non-PEBS sampling mode.

Profiling Tools

The choice of profiling tools used to collect the hardware event samples also affect the quality of samples collected. We compared `oprofile` [10] and `perfmon2` [11] (details omitted for brevity). `oprofile` does not support the PEBS sampling mode. Moreover, the quality of samples collected using `oprofile` were inferior to those obtained using `perfmon2` in the non-PEBS sampling mode, as determined using our "degree of overlap" measures, and performance runs with FDO using the sample profiles. We therefore use `perfmon2` for profile collection.

5.2 Missing Source Position Information

Since we use source position information to correlate samples to 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 sample counts (shown as comments) attributed to a couple of hot basic blocks in procedure `new_dbox()` in the SPEC2000 benchmark `300.twolf`.

```
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, which seems to indicate that the branch at line 93 is 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. `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()`. We are investigating the possibility of enhancing GCC to maintain source position information across copy propagation into PHI nodes and regeneration from PHI nodes in order to fix this issue.

5.3 Insufficient Source Position Information

Some cases require enhancements to the current debug/source position information in order to be handled correctly by the sample profile-based FDO support.

Control Flow Statements in a Single Source Line

Examples:

```
if (cond) {stmt1} else {stmt2}

(cond) ? (taken_body) : (not_taken_body);
```

In such cases, it is not possible to differentiate the samples that should be attributed to the basic block containing the branch condition and the samples that should be attributed to the basic block containing the taken or not-taken body of the branch, since all the samples will be attributed to the single source line. In order to handle such source statements, the debug information in GCC should be enhanced to discriminate control transfers within a single source line.

Early Inlined Routines

At the time of sample annotation of the basic blocks, the CFG contains early inlined routines. Samples that are attributed to the basic blocks of the early inlined routines should be scaled appropriately, if the aggregated sample counts for the inlined routine are used. Furthermore, the execution profile for a particular inlined instance may not match the aggregated inlined function sample count. Currently, the sample profile feedback file format supports aggregating samples for inlined functions per caller function. It would be more accurate to differentiate the samples for each inlined callee function instance, which requires enhancements to the current source position information format.

Macros

Currently all the instructions pertaining to MACROS are attributed to the first source line of the MACRO use in GCC. It is therefore not possible to differentiate samples

within the multiple statements within a MACRO, especially if the MACRO contains control transfers. Again, enhancements to the source position information are needed to handle sample attribution to MACROS correctly.

6 Results

6.1 Overlap Measures

The accuracy of the estimated edge profiles depend on several factors:

1. The quality of the sample profiles
2. The completeness and accuracy of source position information
3. The effectiveness of the edge profile estimation heurisitics

We use the *degree of overlap* measure used in [9] to compare the edge profiles constructed using the sample profiles ($G1$) with the exact edge profiles ($G2$) obtained using instrumentation. The edge set E in both $G1$ and $G2$ are identical.

$$overlap(G1,G2) = \sum_{e \in E} min(pw(e,G1), pw(e,G2)) \quad (9)$$

where $pw(e,G)$ is defined as the percentage of the edge e's weight of the CFG G's total edge weight. A higher *degree of overlap* number indicates higher accuracy in the edge profile values estimated by the heuristics. The *degree of overlap* numbers obtained for the SPEC2000int benchmarks by comparing the estimated edge profiles for the different cases with the exact edge profiles obtained from instrumented runs is shown in Table 1.

The columns in Table 1 are labeled as follows:

- Base: Default edge profile estimation using static profiles.
- O0/N: Edge profile estimation using sample profiles collected from -O0 binaries *without* applying the minimum-cost maximal flow algorithm.

Benchmark	Base	O0/N	O0	O2
164.gzip	68.5	50.27	77.14	73.68
175.vpr	65.91	70.49	81	75.73
176.gcc	45.45	51.78	54.58	59.41
181.mcf	45.77	70.5	71.08	67.91
186.crafty	59.96	51.61	71.86	64.46
197.parser	72.77	70.59	79.65	71.57
252.eon	87.75	62.01	66.22	62.5
253.perlbmk	61.35	61.65	72.33	75.67
254.gap	71.18	66.6	72.5	78.98
255.vortex	59.26	57.35	61.3	60.26
256.bzip2	51.16	46.45	80.85	79.05
300.twolf	73.74	68.84	79.49	77.46
Average	63.47	60.68	72.33	70.56

Table 1: Degree of Overlap Measures

- O0: Edge profile estimation using sample profiles collected from -O0 binaries.
- O2: Edge profile estimation using sample profiles collected from -O2 binaries.

The average *degree of overlap* measure using static profiles, as done in default -O2 runs is 63.47%, which is used as the base for comparision with edge profiles constructed from sample profiles. We see that if we use sample profiles, *without* employing the minimum-cost maximal flow algorithm, then the *degree of overlap* measure decreases to 60.68%. The value of the measure improves when applying the minimum-cost maximal flow algorithm to estimate the edge profiles when using sample profile data collected with:

- -O0 binaries to 72.33% and
- -O2 binaries to 70.56%.

for all the benchmarks, excepting the C++ benchmark `252.eon`. Currently we do not use the sample profile data present for inlined functions during basic block annotation, and this may result in incorrect edge weights being computed for early inlined routines, thereby affecting the *degree of overlap* measure for `252.eon`. We are currently investigating this issue.

We expect the *degree of overlap* and performance gains to be better for FDO with sample profiles when using -O0 binaries for profile collection as compared to using -O2 binaries due to source correlation issues seen with optimized binaries. The performance run results

outlined in the next section correlate positively to the overall trend in the average *degree of overlap* measures and expectations.

6.2 Experimental Evaluation

Our performance experiments were carried out using 32-bit binaries of the SPEC2000int C benchmarks on the AMD Opteron platform. We compare the performance gains of instrumentation-based FDO with sample profile based FDO using GCC built -O0 and -O2 binaries for profile collection. The sample profile collection was done on Intel Core-2 platform using the PEBS sampling mode. We also compare sample profile-based FDO with and without using the minimum-cost maximal flow algorithm, to show that the application of this algorithm is indeed necessary to realize the performance gains. The base run used for comparision in all our experiments is the default -O2 run using static profiles, i.e., without FDO.

We are currently tuning our heuristics to use samples collected from inlined routines. The C++ benchmark `252.eon` shows a very high performance gain of approximately 18% using instrumented FDO, as compared to a relatively low performance gain of 6% using sample profile based FDO. We are currently looking into this issue. Since this is work in progress, we have omitted the C++ benchmark `252.eon` from our experimental results.

The option `-fprofile-use` enables feedback-directed optimizations which use both value and edge profile data. Specifically, the following options are enabled: `-fbranch-probabilities`, `-fvpt`, `-funroll-loops`, `-fpeel-loops`, and `-ftracer`. Of the above, `-fvpt` applies to using the value profile data, and the remaining options apply to using the edge profile data.

In Table 2, we compare the performance runs using the default edge profile options `-fbranch-probabilities`, `-funroll-loops`, `-fpeel-loops`, and `-ftracer` enabled. The columns are labeled as follows:

- I: -O2 run with instrumentation-based FDO.
- S/O0/N: -O2 run with sampling-based FDO and *without* applying the minimum-cost maximal flow algorithm. Profile data collected from -O0 binaries.

Benchmark	I	S/O0/N	S/O0	S/O2
164.gzip	3.36	-4.75	2.66	1.62
175.vpr	5.28	2.64	4.91	6.04
176.gcc	6.41	1.28	1.68	3.12
181.mcf	0.45	0.00	3.84	3.16
186.crafty	2.98	0.94	4.93	3.06
197.parser	1.21	-1.09	0.48	0.85
253.perlbmk	-0.81	-2.27	-0.73	-0.16
254.gap	2.99	-4.72	1.16	2.50
255.vortex	2.27	-2.27	1.88	1.10
256.bzip2	5.97	3.10	1.38	0.23
300.twolf	2.22	3.38	5.52	1.89
Average	2.94	-0.34	2.52	2.13

Table 2: Performance gains with default edge profile options enabled

- S/O0: -O2 run with sampling-based FDO. Profile data collected from -O0 binaries.
- S/O2: -O2 run with sampling-based FDO. Profile data collected from -O2 binaries.

Instrumentation-based FDO runs show an average gain of 2.94%, whereas sampling-based FDO runs using:

- -O0 binaries show an average gain of 2.52% (approximately 86% of the instrumented FDO gain)
- -O2 binaries show an average gain of 2.13% (approximately 72% of the instrumented FDO gain)

When only the initial edge weights estimated from the static profile heuristics and the basic block sample counts are used, *without* the application of the minimum-cost maximal flow algorithm, the average performance gain degrades to -0.35%. We can therefore conclude that the minimum-cost maximal-flow algorithm is necessary to achieve performance gains with sample profile-based FDO.

We also compare the performance gains when only the option `-fbranch-probabilities` is enabled for the FDO runs as shown in Table 3. The columns are labeled similarly as for Table 2.

For FDO using instrumented profiles, enabling the default edge profile specific options `-fbranch-probabilities`, `-funroll-loops`, `-fpeel-loops`, and `-ftracer` result in a higher *average* performance gain. However, `181.mcf`,

Benchmark	I	S/O0/N	S/O0	S/O2
164.gzip	2.08	-0.12	1.27	1.16
175.vpr	5.28	2.01	4.28	5.79
176.gcc	5.92	-0.16	3.28	3.44
181.mcf	1.81	0.11	3.95	2.03
186.crafty	5.95	3.74	5.44	5.10
197.parser	-0.85	0.24	0.60	0.60
253.perlbmk	5.85	-4.31	1.22	2.19
254.gap	-2.02	-0.58	-0.96	-1.83
255.vortex	3.14	3.29	2.12	2.04
256.bzip2	2.41	3.56	0.57	1.49
300.twolf	-3.71	-1.24	1.81	5.02
Average	2.35	0.60	2.14	2.46

Table 3: Performance gains with only option `-fbranch-probabilities` enabled

`186.crafty`, `253.perlbmk` and `255.vortex` show better performance gains when only the option `-fbranch-probabilities` is enabled. FDO with sample profiles collected using -O2 binaries show an average gain of 2.46% when only the `-fbranch-probabilities` option is enabled, as compared to a slightly lower gain of 2.13% when all the edge profile specific options are enabled. These results seem to indicate that sample profiles are not very effective for the loop-specific optimizations enabled by the `-funroll-loops` and `-fpeel-loops` options. We are currently investigating this further.

7 Current Status and Future Work

Our initial experiments show that edge profiles constructed from INST_RETIRED event samples can be used to achieve the performance gains of traditional FDO with instrumented edge profiles, while overcoming the shortcomings of the traditional FDO usage model. We have identified several problem areas, especially in the shortcomings of the source position/debug information support in GCC that is currently being addressed. Specifically, work is in progress to enhance the debug information and minimum line table information to support better handling of source lines that span multiple basic blocks, inlined routines, and MACROS by the basic block sample annotation heuristics.

We also plan to apply the edge profile estimation heuristics described in this paper to existing problems in GCC due to inconsistent basic block and edge frequency counts obtained in some cases with traditional instrumentation-based FDO. For example, when profiling multi-threaded applications, the basic block and edge frequency counts obtained via instrumentation are under-counted in some cases due to the loss of some of the counter increments when multiple threads increment the same counter without using synchronization primitives. The minimum-cost maximal flow algorithm implemented can be used to effectively fix the inconsistent basic block and edge frequency counts for the above scenario.

One main drawback of our sampling-based FDO method is that it does not support value profiling which is supported by the instrumentation-based FDO method. Our experiments indicate that a large percentage of the performance gains obtained by value profiling is due to their use in memset/memcpy inlining. These performance gains can still be achieved with edge profiling alone, without the use of value profiling, by tuning the inlining heuristics and methods.

The INST_RETIRED samples can be used for procedure re-ordering optimizations—for example, by the Whole Program Optimizer (WHOPR) project [5]. In the future, we would like to extend the sample profile datafile format to support sampling of other hardware events, such as data and instruction cache misses to be used in data layout and procedure re-ordering optimizations and branch mispredict event samples to be used in `if-conversion` optimizations.

8 Acknowledgments

We would like to thank Roy Levin for his help and support in promptly and enthusiastically answering our questions on the algorithm described in [9], Seongbae Park for his help in analysis of GCC source correlation issues, and the reviewers for their valuable feedback.

References

[1] 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, 1997.

[2] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[3] Thomas Ball and James R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.

[4] Richard Bellman. On a routing problem. In *Quarterly of Applied Mathematics, 16(1)*, pages 87–90, 1958.

[5] Preston Briggs, Doug Evans, Brian Grant, Robert Hundt, William Maddox, Diego Novillo, Seongbae Park, David Sehr, Ian Taylor, and Ollie Wild. Whopr - fast and scalable whole program optimizations in gcc, 2008.

[6] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. In *Canadian Journal of Mathematics 8*, pages 399–404, 1956.

[7] Andrew V. Goldberg and Robert E. Tarjan. Finding minimum-cost circulations by canceling negative cycles. *J. ACM*, 36(4):873–886, 1989.

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

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

[10] Oprofile. `http://oprofile.sourceforge.net`.

[11] Perfmon2. `http://perfmon2.sourceforge.net`.

[12] R. L. Probert. Optimal insertion of software probes in well-delimited programs. *IEEE Trans. Softw. Eng.*, 8(1):34–42, 1982.

[13] Vinodha Ramasamy, Dehao Chen, Wenguang Chen, and Robert Hundt. Feedback-directed optimizations with estimated edge profiles from hardware event sampling. In *Open64 Workshop at CGO*, 2008.

[14] 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.