Can We Trust Profiling Results?

Understanding and Fixing the Inaccuracy in Modern Profilers

1 SKID EFFECT MODELING ON SIMPLE LOOP

These are some definitions to help explain our mathematical model:

- N: number of instructions in a simple loop.
- $\mathbf{I} = \{I_1, I_2, I_3, ..., I_N\}$ ($|\mathbf{I}| = N$): simple loop consists of N instructions. All instructions are executed one by one.
- Each instruction has its duration in cycles, we define each instruction I_n's cycle duration is c_n, and we have $0 < n \leq N$. Here c_n is also called cycles per instruction (CPI) for instruction I_n. We can also define a CPI vector for this cyclical code sequence: $\mathbf{c} = \{c_1, c_2, c_3, ..., c_N\}$.
- S: skid cycles duration, quantified in cycles.
- N_s: the total sampled instructions in a simple loop.
- T: the instruction sampling period.
- N_e: the number of times the simple loop $\mathbf{I}$ executed.
- $g(S, m, \mathbf{c})$: the skid length of the instruction I_m, quantified in instructions.
- $\mathbf{D}(S, \mathbf{c})$: the instruction distribution of the loop I_n affected by skid effects with a skid duration S in cycles.

Since all instructions in this loop are executed circularly, there is an instruction sequence $\mathbf{I}$ exactly after $\mathbf{I}$ executed. For a better understanding, we have:

$$I_{N*K+n} = I_n, K \in \mathbf{N}^+, 0 < n \leq N$$

$$c_{N*K+n} = c_n, K \in \mathbf{N}^+, 0 < n \leq N$$

1.1 Ideal Instruction Sampling Without Skid

Before we go deep into the skid problem, we need to have a mathematical explanation of ideal sampling without skid effects. Such an ideal model of sampling scheme could help us understand more complex scenarios. The sampling period T is usually a very large number ($T \gg N$) to minimize the sampling overhead. Additionally, T will be set as a prime number to ensure that instruction sample points are evenly distributed in all instructions in $\mathbf{I}$. The total sampled instructions in this loop is:

$$N_s = \frac{N * N_e}{T}$$

For this whole loop, we can use N_S to estimate total instructions: $N_S * T$, without bias. To ensure there are enough sample points number N_S, we need to guarantee $N * N_e \gg T$. The sampled points number distribution $\mathbf{D}(0, \mathbf{c})$ can be represented as:

$$\mathbf{D}(0,\mathbf{c}) = \begin{matrix} 1 & 2 & \cdots & N \end{matrix} \begin{bmatrix} N_e & N_e & \cdots & N_e \end{bmatrix} = \begin{bmatrix} \frac{N_s*T}{N} & \frac{N_s*T}{N} & \dots & \frac{N_s*T}{N} \end{bmatrix}$$

Since we can guarantee instruction profiling result is not biased at the instruction level, each basic block or function will also get an unbiased profiling result. However, in real sampling based profiling, skid effects will cause the offset between the sampled instruction and instruction causing the hardware counter overflow. We will explain skid effects modeling to calculate sample points' distribution with skid duration in cycles in the next subsection.

1.2 Mathematical Modeling of Skid Effect

To describe the mathematical model quantifying the skid effects, instruction I_m from the simple loop instructions $\mathbf{I}$ is used to explain the offset caused by skid effects. If the instruction counter overflow happens at I_m, and the attributed instruction is $I_{m'}$, we can guarantee $m' > m$. For skid length in cycles S, we can obtain its range :

$$\sum_{n=m+1}^{m'-1} c_n < S \leq \sum_{n=m+1}^{m'} c_n$$

Skid length quantified in instruction g(S,m,**c**) can be represented as :

$$g(S, m, \mathbf{c}) = m' - m$$

$g(S, m, \mathbf{c})$ is decided by three input coefficients:

- Skid duration S cycles. We assume $S \leq \sum_{n=1}^{N} c_n$.
- Instruction index m for I_m in the code sequence $\mathbf{I}$.
- All CPI information $\mathbf{c}$.

For instruction I_m, at which counter overflow happens, from $\mathbf{I}$, $g(S, m, \mathbf{c})$ can be defined as a periodic step function for S:

$$g(S,m,\mathbf{c}) = \begin{cases} 1, & 0 < S \leq c_{m+1} \\ 2, & c_{m+1} < S \leq c_{m+1} + c_{m+2} \\ 3, & c_{m+1} + c_{m+2} < S \leq \sum_{n=m+1}^{m+3} c_n \\ \dots & \\ N-1, & \sum_{n=m+1}^{m+N-2} c_n < S \leq \sum_{n=m+1}^{m+N-1} c_n \\ N, & \sum_{n=m+1}^{m+N-1} c_n < S \leq \sum_{n=m+1}^{m+N} c_n \end{cases} \quad (1)$$

I_m's sampling point distribution without skid effects can be defined as:

$$\mathbf{d}(0,\mathbf{c},m) = \begin{matrix} 1 & \cdots & m-1 & m & m+1 & \cdots & N \end{matrix} \begin{bmatrix} 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \end{bmatrix}$$

After we obtain skid length $g(S, m, \mathbf{c})$ for a specific instruction I_m, I_m's sampling point distribution with skid S can be obtained using the shift matrix:

$$\mathbf{d}(S,\mathbf{c},m) = \mathbf{d}(0,\mathbf{c},m) \times \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & & \ddots & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & 0 & \cdots & 0 & 0 \end{bmatrix}^{g(S,m,\mathbf{c})}$$

All attributed instruction distribution $\mathbf{D}(S, \mathbf{c})$ can be obtained after sampling:

$$\mathbf{D}(S, \mathbf{c}) = \frac{N_S * T}{N} * \sum_{m=1}^{N} \mathbf{d}(S, \mathbf{c}, m) \tag{2}$$

2 FORMULATING OPTIMIZATION PROBLEM TO ELIMINATING SKID EFFECTS ON CONTROL FLOW GRAPH

Below are some definitions used to explain our methodology:

- **K**: a control flow graph containing multiple conditional branches, it can be decomposed into multiple simple loops. **K**, we have L simple loops.
- K_l: a simple loop in **K**. The execution time for a simple loop K_l is F_l. **K** can also be represented as $\{K_1, K_2, K_3, ..., K_l, ..., K_L\}$.
- I'_i: a instruction in the complex loop **K**.
- $\mathbf{F} = \{F_1, F_2, ..., F_L\}$: a set of the number of execution times each simple loop in **K**.
- B_b: a basic block in the control flow graph of K, which contains several instructions. One basic block may belongs to one or several simple loops in **K**.
- $\delta(i, b)$: a binary value indicates whether instruction I_i belongs to basic block B_b.
- $\lambda_{i,l}$: the index of instruction I'_i in simple loop K_l. if instruction I_i is not in simple loop K_l, it will be 0.
- $d\left(S, \mathbf{c}_l, \lambda_{i,l}\right)$: the value of $\mathbf{D}(S, \mathbf{c}_l)$'s $\lambda_{i,l}$th element. We define $d\left(S, \mathbf{c}_l, 0\right) = 0$.
- E_i: the instruction profiling result for I'_i from sampling based profiler.
- $E_i(\mathbf{F})$: given a simple loop execution frequency vector **F**, instruction executed frequency for instruction I'_i.
- $\hat{E}_i(\mathbf{F})$: the instruction count of I'_i after skid effects emulation with a specific simple loop frequency vector **F**.
- C_i: the cycle profiling result for I'_i from sampling based profiler.
- c_i: the average CPI for instruction I'_i.
- $\varphi(\mathbf{F})$: a metric to quantify the edge frequency result at basic block level close enough to the sampling profile result.

2.1 Complex Loop Decomposition

For a control flow graph **K**, we cannot directly emulate skid effects with instruction and cycle profiling results. We then focus on the simple loops $\{K_1, K_2, K_3, ..., K_l, ..., K_L\}$ to do skid effects emulation. Each simple loop K_l can be treated as a simple loop introduced in section **??**.

We use the example of `hmmer` in SPEC CPU2006 to illustrate the decomposition of complex loop into simple loops. Listing 1 shows the code of for-loop in `hmmer`, which takes more than 80% of instructions and nearly 90% of cycles. Figure 1 plots the control flow graph of this for-loop. For each basic block, we use the first IP (instruction pointer) to represent it. This for-loop contains 3 basic blocks. After decomposing the control flow graph, we obtain two simple loops: `40c570` → `40c612` → `40c659` and `40c570` → `40c659`. Both simple loops contain the specific basic block `40c659`. While for `40c612`, only one simple loop contains it.

```
133 for (k = 1; k <= M; k++) {
134   mc[k] = mpp[k-1]   + tpmm[k-1];
135   ...
136   if (k < M) {
137     ic[k] = mpp[k] + tpmi[k];
138     if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
139     ic[k] += is[k];
140     if (ic[k] < -INFTY) ic[k] = -INFTY;
141   }
142 }
```

Listing 1: Code for for-loop in fast_algorithms.c from `hmmer` in SPEC CPU2006. Unlike the `if` branch at line 4, the `if` branch at line 6 and line 8 will not be translated into branch instructions. As a result, a control flow graph in Figure 1 can be generated by this for-loop.

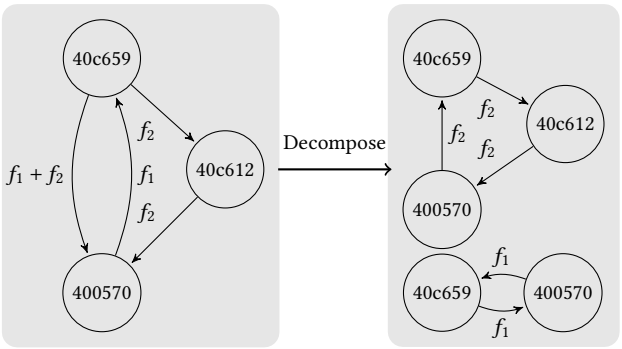


Figure 1: Decomposing control flow graph of `hmmer` in SPEC CPU2006.

The total instructions for a specific loop is not biased by sampling based profiler. For complex loop **K**,the total instructions number $|E|$ for the whole loop can be represented with simple loop execution frequency **F** and all basic blocks in each simple loop K_l as:

$$|E| = \sum_{F_l \in \mathbf{F}} F_l \left(\sum_{B_b \in K_l} |B_b| \right) \tag{3}$$

As a result, **F**'s degree of freedom is $L - 1$ here. Furthermore, each basic block instruction number $|B_b|$ is fixed, which can be obtained by static analysis. For a specific instruction I'_i, its execution frequency is:

$$E_i(\mathbf{F}) = \sum_{F_l \in \mathbf{F}} F_l \left(\sum_{B_b \in K_l} \delta(i, b) \right) \tag{4}$$

$$\delta(i, b) = \begin{cases} 1, & I'_i \in B_b \\ 0, & I'_i \notin B_b \end{cases}$$

Moreover, as we explained in section **??**, the cycle profiling result here is also not biased at all. For a specific instruction I'_i, based on the its cycle profiling result C_i and execution frequency $E_i(\mathbf{F})$, the CPI estimation for I'_i can be obtained as:

$$c_i = \frac{C_i}{E_i(\mathbf{F})} \tag{5}$$

Different from the mathematical model for simple loop, the instruction and cycle profiling result cannot provide the cycle execution frequency information F_l for each simple loop K_l. We cannot directly obtain the CPI information for each instruction I'_i in K_l without edge frequency information for all branches. Since each

instruction's CPI information is up to the overall simple loop frequency **F**, we run emulate skid effects to get the problematic instruction distribution for any possible **F**. Based on this property, an optimization problem can be formulated to obtain **F**. We will describe optimization formulation in detail in the next subsection.

2.2 Formulating an Optimization Problem

With a candidate setting of simple loop frequency **F**, the CPI information $\mathbf{c}_l$ for each simple loop can be obtained based on equation 5. Since the skid cycle duration S can be measured for a specific machine, Skid effects emulation can be done for simple loop K_l by equation 2 to generate the instruction distribution $\mathbf{D}(S, \mathbf{c}_l)$.

After doing skid effects emulation for each simple loop with **F**, we can get the overall instruction count $\hat{E}_i(\mathbf{F})$ of a specific instruction I', which can be represented as:

$$\hat{E}_i(\mathbf{F}) = \sum_{F_l \in \mathbf{F}} \left(F_l \sum_{B_b \in K_l} \left(\sum_{I_i \in B_b} \delta(i,b) d\left(S, \mathbf{c}_l, \lambda_{i,l}\right) \right) \right) \tag{6}$$

We can use sampling based profiling to get instruction number E_i for instruction I_i. To quantify the edge frequency result close enough to the sampling profile result, we define a metric to quantify the difference between E_i and $\hat{E}_i(\mathbf{F})$ at basic block level:

$$\varphi\left(\mathbf{F}\right) = \sum_{B_b} \left(\sum_{I_i \in B_b} E_i - \sum_{I_i \in B_b} \hat{E}_i(\mathbf{F}) \right)^2$$

Then we can formulate an optimization problem to obtain **F**:

$$\arg\min_{\mathbf{F}} \left(\varphi(\mathbf{F})\right)$$

The optimization problem formulation is based on the connection of simple loop execution frequency **F** and CPI information. The skid effects emulation with **F** can reproduce the instruction distribution caused by skid effects, for all basic blocks in this complex loop, if the **F** is close enough to the ground truth.

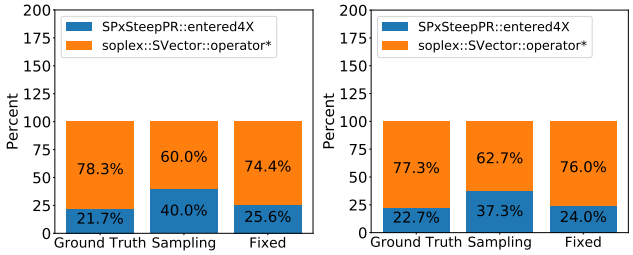
2.3 Solving the Optimization Problem

To solve this optimization problem, we adopt Gibbs Sampling method [2]. Algorithm 1 describes the procedure to obtain the simple loop frequency **F**. At the beginning, we prepared the cycle and instruction profiling result for a problematic loop with complex branches. After we analyze assembly code, the control flow graph of this loop is decomposed to simple loops set as **K**. **F** is initialized with random values subject to equation 3. Since **F**'s degree of freedom is $L-1$, we focus on the sampling values for $\{F_1, F_2, ..., F_{L-1}\}$. For specific $F_{l'}$, its value range is fixed when $\mathbf{F}_{l'}^{-}$ are fixed and E is known after doing instruction profiling. $p(F_{l'}|\mathbf{F}_{l'}^{-})$'s distribution is estimated by doing skid emulation with candidate $F_{l'}$ value within its range and $\mathbf{F}_{l'}^{-}$. After that, we are able to sample candidate value for $F_{l'}$. We iteratively update each $F_{l'}$ until the $\varphi(\mathbf{F})$'s value become constant and small enough.

Algorithm 1: Recover Edge Profiles

Input: skid cycle duration: S, cycle profiling result **c**, instruction profiling result **I**
Output: simple loop frequency $\mathbf{F} = \{F_1, F_2, \ldots, F_L\}$

```
Initialization: Obtain simple loop information K through static analysis;
Choose a random F;
runningFlag ← 1;
while runningFlag do
    for l' ← 1 to L − 1 do
        F_{l'}^- ← {F_1, F_2, ..., F_{l'}, F_{l'+1}, ..., F_{L−1}};
        Obtain F_{l'}'s range by F_{l'}^- and |E| using equation 3;
        Obtain F_{l'}'s probability distribution P(F_{l'}, F_{l'}^-) with its range
          and skid emulation;
        Sampling F_{l'} from the distribution P(F_{l'}, F_{l'}^-);
        Calculate F_L by F_{l'} and F_{l'}^-;
        Update F_{l'} and F_L in F ;
    end
    if φ(F) not converge to a stable value then
        runningFlag ← 0
    end
end
```



(a) `libsvm`,Intel-SandyBridge **(b)** `libsvm`,Intel-Skylake

Figure 2: Fixed result of 2 mis-attributed functions' instruction profile in `soplex` for Intel-SandyBridge and Intel-Skylake.

3 CASE STUDY

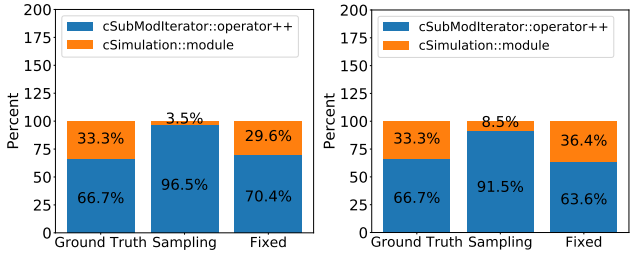
3.1 SPEC CPU2006 soplex

`soplex` is a benchmark solving linear programming with reference input, written in C++. Our metric ϵ reports that 15% of the instructions are mis-attributed on function level, and its problematic loop takes nearly 20% of the total instructions.

Listing 2 lists the code for the problematic loop and small function called by this for-loop, respectively. This loop is belong to function `entered4X` from `spxsteeppr.cc`. In this loop, an inline operator function `operator*` from `svector.h` is called at line 8. This operator function is used to obtain the dot product of two vectors. Before doing dot product computation, the two vectors need to be loaded to the CPU. These memory load instructions are much heavier than the instructions of this function. As a result, `operator*`'s instruction number is under-attributed, while `entered4X`'s instruction number is over-attributed. Such miss-attribution terribly affects the function `entered4X` IPC result. Especially for Intel-Skylake platform, the IPC of `entered4X` is over-estimated to 1.82× .

```
// caller entered4X in spxsteepper.cc
void SPxSteepPR::entered4X(SPxId, int n, int start2, int incr2,
      int start1, int incr1){
  ...
  for (j = pIdx.size() - 1 - start2; j >= 0; j -= incr2){
    i = pIdx.index(j);
    xi_ip = xi_p * pVec[i];
    x = penalty_ptr[i] += xi_ip * (xi_ip * pi_p
        - 2 * (thesolver->vector(i) * workVec));
    if (x < delta)
      penalty_ptr[i] = delta;
    else if (x > infinity)
      penalty_ptr[i] = 1 / thesolver->epsilon();
  }
  ...}
...
// callee operator* in svector.h
Real operator*(const Vector& w) const{
  Real x = 0;
  int n = size();
  Element* e = m_elem;
  while (n--){
    x += e->val * w[e->idx];
    e++;
  }
  return x;
}
```

Listing 2: Code of problematic loop in `soplex`.



(a) `libsvm`,Intel-SandyBridge **(b)** `libsvm`,Intel-Skylake

Figure 3: Fixed result of 2 mis-attributed functions' instruction profile in `omnet` for Intel-SandyBridge and Intel-Skylake.

Figure 2a and Figure 2b show the fixed result of the two functions. For both `entered4X` and `operator*`, our method recovers the instruction profile with less than 18% error, compared with ground truth result by CCTLib.

3.2 SPEC CPU2006 omnet

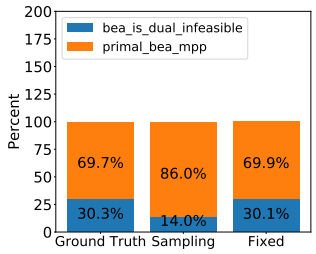
Application `omnet` focus on discrete event simulation of a large Ethernet network , which is also written in C++ and configured with reference input. Based on `omnet`'s ϵ value, 27% and 36% of overall instructions are mis-attributed.

Listing 3 gives the code of the problematic loop, which belongs to function `cSubModIterator::operator++` (short for `operator++`) from `cmodule.cc`. This loop accounts the 33.3% of the total instructions. Inline function `module` from `csimul.h` is called by this loop at line 6. The purpose of this loop is to search a target element in an array and return its parent. Every element's parent needs to be checked with a point-chasing memory access pattern at line 7, which incurs heavy data loading from memory. Due to such heavy instruction outside `module`, `module`'s instruction profile is significantly biased.

Figure 3a plots the instruction counts for these two functions from HPCToolkit, CCTLib and fixed version. HPCToolkit result shows the IPC of the `module` is only 10% of the ground-truth. Such result will mislead programmer's optimization. Our algorithm fixes it with at most 11.25% error.

```
//caller cSubModIterator::operator++ in cmodule.cc
cModule *cSubModIterator::operator++(int){
  ...
  do{
    id++;
    cModule *mod = simulation.module(id);
    if (mod!=NULL && parent==mod->parentModule())
      return mod;
  }
  while (id<=lastId);
  ...
}
...
//callee module in csimul.h
cModule *module(int id) const {
  return id>=0 && id<size ? vect[id] : NULL;
}
```

Listing 3: Code for problematic loop in `omnet`.



(a) `libsvm`,Intel-SandyBridge

Figure 4: Fixed result of 2 mis-attributed functions' instruction profile in `mcf` for Intel-SandyBridge and Intel-Skylake.

```
// caller primal_bea_mpp in pbeampp.c
arc_t *primal_bea_mpp( long m,  arc_t *arcs, arc_t *stop_arcs,
      cost_t *red_cost_of_bea ){
  ...
  for( ; arc < stop_arcs; arc += nr_group ){
      if( arc->ident > BASIC ){
          red_cost = arc->cost - arc->tail->potential + arc->
              head->potential;
          if( bea_is_dual_infeasible( arc, red_cost ) ){
              basket_size++;
              perm[basket_size]->a = arc;
              perm[basket_size]->cost = red_cost;
              perm[basket_size]->abs_cost = ABS(red_cost);
          }
      }
  }
...}
...
// callee bea_is_dual_infeasible in pbeampp.c
int bea_is_dual_infeasible( arc_t *arc, cost_t red_cost ){
    return(     (red_cost < 0 && arc->ident == AT_LOWER)
            || (red_cost > 0 && arc->ident == AT_UPPER) );
}
```

Listing 4: Code for problematic loop in `mcf`.

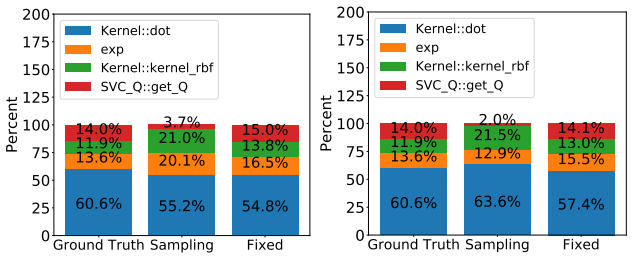
3.3 SPEC CPU2006 mcf

`mcf` is an application implemented to do single-depot vehicle scheduling in public mass transportation. We configure it with reference input. Its ϵ value indicates 13% of the total instructions are mis-attributed on Intel-SandyBridge.

Listing 4 lists the code for the problematic loop, which accounts nearly 40% of the total instructions. This loop comes from function `primal_bea_mpp` in `pbeampp.c`. `primal_bea_mpp` calls `bea_is_dual_infeasible`, which is an inline function called inside this loop. There are a lot of data loading with stride memory access pattern inside this loop. Such memory loading instructions are with more cycles than other instructions in function `bea_is_dual_infeasible`. As a result, the instruction number of `bea_is_dual_infeasible` is under-estimated in HPCToolkit result.

```
// Callee: Kernel::kernel_rbf in svm.cpp
double kernel_rbf(int i, int j) const{
  return exp(
    -gamma*(x_square[i]+x_square[j]-2*dot(x[i],x[j]))
  );
}
...
// Caller: SVC_Q::get_Q's for-loop in svm.cpp
for(j=start;j<len;j++)
  data[j] = (Qfloat)(y[i]*y[j]*(this->*kernel_function)(i,j));
```

Listing 5: Code of problematic functions in `libsvm`.



(a) `libsvm`, Intel-SandyBridge **(b)** `libsvm`, Intel-Skylake

Figure 5: Fixed result of 4 mis-attributed functions' instruction profile in `libsvm` for Intel-SandyBridge and Intel-Skylake.

Figure 4a gives the instruction profiling result for these two functions from HPCToolkit, CCTLib and the fixed result. HPCToolkit under-estimates `bea_is_dual_infeasible`'s IPC to 46% of the ground truth. Our algorithm fixes its IPC with an error less than 5.7%.

3.4 libsvm

The library `libsvm` is a popular software package for support vector machines [1]. We run the training phase of the default radial basis function kernel with the input `cod-rna` containing 59,535 training samples with two-class labels. Listing 5 shows the hottest for-loop in `svm.cpp` from the function `SVC_Q::get_Q`, covering nearly 85% of the total executed instructions. It calls `Kernel::kernel_rbf` that further invokes math functions `exp` and `Kernel::dot` to compute dot product in each loop execution.

Figure 5a and Figure 5b show the fixed result of this hot for-loop on Intel-SandyBridge and Intel-Skylake, respectively. After comparing the HPCToolkit and CCTLib profiling results of this for-loop, we find that `SVC_Q::get_Q` is significantly under-attributed on both platforms. `SVC_Q::get_Q` contains lighter for-loop control instructions, compared with data load and multiply instructions. While `Kernel::dot`'s for-loop is self-contained, the few for-loop instructions skid to other functions. There is a heavy divide instruction inside `exp`, but it is not very close to the `exp`'s exit, so that makes the profiling result of `exp` varies between two platforms. `Kernel:kernel_rbf` is a small function with around 30 instructions calling `Kernel::dot` and `exp`. As a result, `Kernel:kernel_rbf`'s instructions are divided into several fragments, each fragment is easily affected by other functions. The function `Kernel:kernel_dot` in the instruction profiling is doubled, compared with ground truth. However, `SVC_Q::get_Q`'s IPC is also substantially overestimated, which could mislead users' optimization strategies.

REFERENCES

[1] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[2] Alan E Gelfand, Susan E Hills, Amy Racine-Poon, and Adrian FM Smith. Illustration of bayesian inference in normal data models using gibbs sampling. *Journal of the American Statistical Association*, 85(412):972–985, 1990.