Low-Overhead Dynamic Instruction Mix Generation using Hybrid Basic Block Profiling

Andrzej Nowak
CERN openlab and EPFL
Andrzej.Nowak@cern.ch

Paweł Szostek
Criteo
Pawel.Szostek@gmail.com

Ahmad Yasin
Intel Corporation
Ahmad.Yasin@intel.com

Willy Zwaenepoel
EPFL
Willy.Zwaenepoel@epfl.ch

***Abstract*—Dynamic instruction mixes form an important part of the toolkits of performance tuners, compiler writers, and CPU architects. Instruction mixes are traditionally generated using software instrumentation, an accurate yet slow method, that is normally limited to user-mode code.**

We present a new method for generating instruction mixes using the Performance Monitoring Unit (PMU) of the CPU. It has very low overhead, extends coverage to kernel-mode execution, and causes only a very modest decrease in accuracy, compared to software instrumentation.

In order to achieve this level of accuracy, we develop a new PMU-based data collection method, Hybrid Basic Block Profiling (HBBP). HBBP uses simple machine learning techniques to choose, on a per basic block basis, between data from two conventional sampling methods, Event Based Sampling (EBS) and Last Branch Records (LBR).

We implement a profiling tool based on HBBP, and we report on experiments with the industry standard SPEC CPU2006 suite, as well as with two large-scale scientific codes. We observe an improvement in runtime compared to software instrumentation of up to 76x on the tested benchmarks, reducing wait times from hours to minutes. Instruction attribution errors average 2.1%.

The results indicate that HBBP provides a favorable tradeoff between accuracy and speed, making it a suitable candidate for use in production environments.

I. Introduction

An instruction mix [1] is a histogram of the frequency of execution of instructions for a given architecture and workload. It can be presented at the granularity of functions, modules or the entire program. Instruction mixes are frequently used by application tuners, compiler writers and CPU architects to study code performance and the interaction between the hardware, the compiler, the operating system and the application. We focus on dynamic instruction mixes, generated at runtime, as opposed to generated by static analysis.

An instruction mix is easily obtained from a basic block execution count (BBEC). If we know how many times a basic block is executed, we also know exactly how many times each instruction within it is executed. We can subsequently combine BBECs with disassembly information to produce an instruction mix or more detailed insights related to specific instructions.

The conventional approach to obtain such a BBEC is by means of software instrumentation. This approach is very accurate, but it is normally limited to user-mode execution, and the data collection overhead and the resulting increase in runtime quickly become prohibitive, especially for long-running programs.

We present Hybrid Basic Block Profiling (HBBP), a new, non-invasive approach for generating instruction mixes. HBBP relies on information obtained from the CPU's Performance Monitoring Unit (PMU [2]), and thus does not require any modification or instrumentation of the program or the operating system. Furthermore, its runtime (collection) overhead is negligibly small. We demonstrate, however, that straightforward methods for collecting PMU data related to BBECs, namely Event Based Sampling (EBS) and Last Branch Records (LBR), introduce significant inaccuracies.

HBBP uses simple machine learning techniques to choose (at analysis time), on a per-basic-block basis, the most accurate form of data collection. The result is significantly improved accuracy, close to that of software instrumentation. As such, HBBP approaches the best of both worlds, the speed of PMU-based data collection and the accuracy of software instrumentation.

We incorporate HBBP into a new tool for instruction mix generation. The tool is composed of two main components: a collector that computes BBECs, and an analyzer that combines the BBECs with static information to produce instruction mixes.

This paper makes the following contributions:

- HBBP - a new and practical approach to procure instruction mixes, without program modification, with good accuracy, with negligible runtime overhead and applicable to kernel-mode code.
- A tool that incorporates HBBP, and allows the generation of instruction mixes for arbitrary programs on standard Linux environments.
- An experimental evaluation of the said approach, which demonstrates near real time performance and an up to 76x speedup over software instrumentation, with an average weighted error of 2.1%. The evaluation is performed on SPEC

CPU2006 workloads and on benchmarks representing large-scale scientific code.

The rest of this paper is structured as follows. In Section II we present our motivation. Section III describes the straightforward methods for data collection using PMUs and their issues. Section IV presents HBBP. Section V presents our tool that incorporates HBBP. Section VI discusses the error metrics we use in the evaluation. Section VII presents our experimental setup. Section VIII reports our experimental results. Section IX covers related work, and Section X concludes the paper.

II. MOTIVATION

A. The use of instruction mixes

Architects use instruction mixes during "black box" discovery to determine which instructions are the most frequently used in their hardware, in order to be able to direct their optimization efforts towards the highest potential benefits for their customers [3].

For compiler writers, instruction mixes offer a high-level insight into the behavior and correctness of tools and programs. For example, it is much easier to discover whether large-scale code vectorizes well using an instruction mix rather than poring through cumbersome compiler reports. An instruction mix can also help determine whether the correct, optimized versions of library functions are being used at runtime. Finally, large amounts of long-latency instructions executed at runtime (e.g., divisions) can easily be discovered and fixed. In such cases, instruction mixes are frequently re-generated, as optimization efforts progress, possibly for various input data and code paths.

For compiler writers and users alike, BBEC-sourced instruction mixes enable automated compiler optimization (PGO or AutoFDO [4]).

Methods used for tuning with performance counters, such as those based on cycle accounting (e.g., TopDown [5] in Intel's VTune or Hierarchical Cycle Accounting [6] [7]), or those used for fine-grained power consumption estimations [8], do not provide a good account of the instructions that a CPU executes. Software developers and tuners are likely to turn to instruction mixes in such cases. One direct use is support for vectorization work, especially assessing possibilities and evaluating outcomes. For example, knowledge about the location and type of vectorization instructions already in use in the program allows locating hotspot candidates for porting from SSE to AVX to AVX2 to AVX512. Another possible use is loop optimization – instruction mixes can reveal not only estimated trip counts but also loop composition and architectural efficiency, or even approximate FLOP rates. Yet another use, of particular importance today, is the study of workloads on accelerators [9]. For example, the Intel Xeon Phi lacks hardware double precision support for some important instructions used in scientific code, e.g., transcendentals. With an instruction mix, it is possible to foresee potential problems by finding hotspots of specific instructions.

B. Obtaining an instruction mix today

Quickly obtaining an accurate instruction mix is currently a difficult undertaking. Two main groups of methods exist: software instrumentation and performance monitoring with hardware support.

In the case of instrumentation, software probes are injected into the workload under test, typically on basic block boundaries. This enables the gathering of precise information [10] at the expense of an increase in runtime. This cost varies with the workload and the efficiency of the monitoring tool. It can easily extend the runtime by a factor of 2-10x, and even 70x in extreme cases, as shown in column (2) of Table 1. Such an increase in collection speed can become an optimization showstopper, e.g., for scientific codes, that need continuous iterative improvement, but where a single run is indivisible and can last many hours. Extracting representative, agile benchmarks from such codes in a reasonable amount of time is often difficult or impossible. Even when it can be done, changes to the main code branch – often actively developed by hundreds or thousands of developers – do not propagate to the benchmark. A second major problem with instrumentation methods is that they change the binary code (and thus the execution path) of the workload. Third, existing tools take even longer when following execution forks or working with multi-threaded programs. Fourth, detailed software instrumentation methods cannot monitor kernel code, or other code running in Ring 0 on x86 or System mode on ARM. Thus, any kernel routines triggered by the code under test remain invisible when using standard software instrumentation.

In the case of hardware-assisted performance monitoring, the PMU provides information about the instructions executed. This information is typically less accurate than that provided by software instrumentation [11]. While methods using the PMU produce occasional interrupts to gather performance data at runtime, they do not disturb the execution, and the runtime cost is usually

Table 1: A comparison of wall clock runtimes in [s] of select benchmarks: clean (1), using software instrumentation with SDE (2)

Benchmark	(1) Clean	(2) SDE
SPEC all	**15'897**	**65'419 (4.11x)**
SPEC povray	224	2710 (12.1x)
SPEC omnetpp	281	2122 (7.56x)
All other benchmarks	**717**	**48'725 (68x)**
Hydro-post benchmark	287	21'959 (76.6x)

miniscule, amounting at most for a few percent of the runtime [12][13].

Some PMUs, such as those in x86 processors, allow the direct collection of instruction-specific performance events. For example, a PMU counter can be programmed to count the number of times a specific computational SSE instruction is executed. However, only a very limited set of instructions, such as a few SSE instructions or divisions, can be monitored in this fashion. The number of such instructions is, moreover, on the decline with more recent processor families (see Table 2 and [14]), dictated by a general trend of reducing PMU complexity.

In this paper we examine improvements to the scope and the accuracy of PMU-based methods. We demonstrate that careful use of omnipresent hardware facilities provides instruction mixes with satisfactory accuracy for all types of instructions and not just a very limited predefined subset.

III. BASE PMU-BASED METHODS

A. *Event Based Sampling*

A well-known approach to obtain instruction execution information is Event Based Sampling (EBS). A PMU counter is programmed to count occurrences of a specific event until a threshold is reached. This threshold is called the Sampling Period. Once the counter overflows, a Performance Monitoring Interrupt freezes the running code and samples the location of the Instruction Pointer (IP). In post-processing, samples are used to build histograms of the number of executions of each instruction. In usual use cases, EBS collection overhead is minor, under 1% [13]. However, as the same study shows, when sampling frequency is increased (aiming for improved accuracy, for example), overheads grow. The overheads do not necessarily grow linearly with the sampling frequency.

To obtain BBECs, we sample on "instructions retired" events. By default, such samples concern only a single instruction, appearing at the address of the sampled IP. We enhance classic EBS by applying every IP sample to all instructions of the enclosing basic block. If the instruction in the sample has executed, the whole block, that contains that instruction, must have executed as well. To obtain proper instruction counts, we must then divide the number of samples recorded for a basic block by the instruction length of that block. In this paper, all further mention of "EBS" refers to EBS with this improvement.

Table 2: Example evolution of computational instruction-specific event support on Intel server PMUs

	Westmere (2010)	**Ivy Bridge (2013)**	**Haswell (2015)**
DIV (cycles)	☑	☑	☒
Math SSE FP	☑	☑	☒
Math AVX FP	N/A	☑	☒
INT SIMD	☒	☒	☒
X87	☑	☑	☒

Amongst other inefficiencies, EBS suffers from two documented problems [11], [15]. "*Skid*" causes the reported IP to be different from the code location that causes the counter overflow. For example, the CPU might be executing other instructions concurrently with the one causing the overflow. As a result, it may be unable to pin-point exactly the source of the overflow. "*Shadowing*" causes samples to disproportionately represent instructions following long-latency instructions in the execution chain. These issues might matter less for large functions, but on smaller functions (e.g., fragmented object-oriented code), and as we aim for accuracy at the instruction level, these two effects quickly become roadblocks.

Several actions can be taken to potentially improve accuracy. First, because of concerns such as skid and shadowing, it is best to sample on a precise variant of the "instructions retired" event [11]. However, even precise variants are affected by these undesirable phenomena, although to a lesser extent. Second, one can increase the amount of collected data. This cannot be done by running multiple simultaneous collections on precise events, because on x86 CPUs they can only be enabled on one of the available PMU counters. Realistically, the only parameter that can be adjusted in the hope of getting more data is the sampling period. Because of the nature of the skid and shadowing problems, however, additional samples tend to pile up in the same code "traps" as before.

B. *Last Branch Records*

Mainstream x86 processors offer a facility called "Last Branch Records" (LBR), which records information about the most recently executed branches. Hardware filtering can enable the collection of only a subset of such branches. Here, we use LBR to obtain BBECs, as described below, as well as by Levinthal and Nowak [7], [11].

A typical LBR record is a stack of 16 entries. Architecturally, the LBR is a circular hardware buffer, continually filled with executed branches. Each of the branches in the LBR stack is stored in the form of a *source-target address pair*. Therefore, we know that no branch occurs between **`Target[i-1]`** and **`Source[i]`**, which in turn means that every basic block encountered on the way is executed. We call such a *target-source* pair a *stream*.

To obtain BBECs, we sample LBR stacks on a "taken branches retired" event. The handling routine picks up the whole LBR stack and stores it away for post-processing.

This technique provides much more information per sample than EBS. Not only is there information about jump sources and targets, but there are many more instructions in each sample, potentially spanning multiple basic blocks between each target and source and spanning multiple usable **`<Target[i-1], Source[i]>`** streams. An LBR stack of size *N* will contain *N-1* such streams. Thus, in order to

obtain BBECs and to normalize the N-1 streams to a single sample, we give each stream a weight of 1/(N-1).

C. Issues with Last Branch Records

LBR sampling provides considerably more information per sample than EBS, and would therefore be expected to offer more accurate BBEC results. However, for a number of basic blocks in a number of workloads, measurements on multiple systems show significant discrepancies between BBECs obtained by LBR and their true values obtained by software instrumentation, sometimes larger than the discrepancies seen with EBS.

A deeper analysis shows that these discrepancies are often triggered by a particular branch occurring a disproportionate number of times (even up to 50% of the time) in entry[0] of the LBR stack. As there is no corresponding **`target[-1]`**, **`source[0]`** cannot be used for the analysis, thereby distorting the results. When we observe a branch occurring in this fashion, we label the corresponding basic block with a "bias" flag, indicating that its analysis by LBR is suspect. These anomalies render LBR by itself insufficient as a basis for accurate generation of instruction mixes.[1]

A second issue, particularly pronounced with LBR but also applicable to EBS, is visible on kernel samples. The Linux kernel includes self-modifying code: it contains probe and trace points which are patched with **`NOP`** instructions when tracing is disabled. In effect, LBR samples suggest the execution stream is ignoring some unconditional branch instructions present in the disassembly. In order to remedy this, after the run we patch the static kernel binary on disk with the **`.text`** extracted from the live kernel image.

D. Summary of issues with EBS and LBR

Table 3 illustrates the issues with the use of EBS and LBR for computing BBECs. It shows for the Fitter program (SSE variant – see Section VIII.C), the BBECs obtained by EBS and LBR, compared to the true values obtained by software instrumentation. Clearly, both EBS and LBR produce major errors on different basic blocks. EBS suffers on short basic blocks, because of skid and shadowing, while LBR suffers on blocks with bias.

IV. HBBP

Given the issues with EBS and LBR used in isolation, HBBP combines the two in an informed way, with very little extra overhead, with the goal of improving overall accuracy.

A. Whether to use EBS or LBR?

For each basic block, the data from EBS and LBR need to be combined to produce a single BBEC. Concretely, we decide (for each basic block) whether to use either EBS or LBR data. Therefore, HBBP does not fix the problems with the individual use of EBS and LBR.

Our intuition, partly based on the knowledge of PMU implementation and the various delays and asynchronies in the processor, is that the length of a basic block and the LBR bias (see Section III.C) have a higher impact on accuracy than other features. We verify this intuition and obtain a cutoff value for the length of a block, below which to use LBR. To arrive at this decision, HBBP learns a rule from training data. Our focus here is not to perform an in-depth machine learning study, but rather to formalize our intuition.

We employ Decision Trees [16], an industry-standard Machine Learning method to determine HBBP criteria. Decision Trees are used as a predictive model that represents combinations of features leading to conclusions. In the tree structure, nodes are feature cutoff values, and leaves are conclusions relating to the class of the target variable. Concretely, we use Classification Trees [16], which have a range of properties relevant to the task at hand. In particular, (1) they can handle both numerical and categorical data; (2) they are simple to interpret (white-box style); (3) they can be represented visually for easy "debugging".

Other popular machine learning models exist, but are harder to interpret, closer to "black-box" style and generally less suited for our purpose. For instance, K-NN [17] is an unsupervised model more suitable for clustering and needs numeric features. SVMs [18] are more complex, less adapted to categorical features and do not offer a guarantee of better performance than Decision Trees.

B. HBBP criteria search

We train our classification trees on approximately 1,100

Table 3: BBEC (in millions) resulting from EBS and LBR in Fitter, compared to those resulting from software instrumentation. Errors >25% are marked in red.

BB	EBS	LBR	SDE
1	3.24	3.16	3.01
2	5.59	2.69	6.00
3	3.05	1.84	3.01
4	2.88	3.17	3.00
5	3.48	1.95	3.50
6	2.22	3.44	3.00
7	3.12	1.17	3.01
8	0.38	0.36	0.50
9	3.43	1.63	3.01
10	14.25	10.15	10.46
11	3.31	2.91	3.01
12	2.84	2.91	3.50
13	0.34	0.48	0.50
14	4.75	7.27	6.86
15	8.67	8.32	9.06

[1] Following our report of these anomalies, LBR has been the focus of improvements in future processor designs by the manufacturer.

basic blocks of training input from non-SPEC benchmarks. The training labels are set to "EBS" and "LBR", depending on which method is closer to the result obtained by software instrumentation.

As features we use code parameters that could have an influence on the underlying performance monitoring subsystem, including, for instance, basic block lengths, instruction-related information, execution counts and bias flags, weighted by the number of executions of the basic block.

The expected output is a rule combining one or more features, their number being limited for simplicity, to decide at analysis time which data source to choose for a given basic block – EBS or LBR.

We generate multiple trees, and we experiment with varying the number of leaves, the number of children per node and the weights on different variables. Our final tree is shown in Figure 1.

Consistently, and in line with expectation, the instruction length of a basic block has the strongest predictive value. For instance, in most tests "feature importance" (reported by Scikit [19]) for block length is higher than 0.7 out of a maximum of 1.0. The prevailing predictive variable at the root of the classification tree is therefore the instruction length of a basic block, and the cutoff value is consistently close to 18. We use this rule in deciding whether to use EBS or LBR data: for blocks with 18 instructions or less we choose values from LBR, while for longer blocks we choose values from EBS. One somewhat surprising conclusion from this study is that although the absence of bias points strongly to LBR (especially on short blocks), on its own bias does not suffice as a predictive variable. Block length dominates, dwarfing all other factors, including bias.

V. TOOL

Our tool does not require any modifications to either the kernel or the Linux "perf" program, runs on any modern Linux system "out of the box", and optionally works with the libpfm4 library [20], translating user-friendly strings to performance event codes.

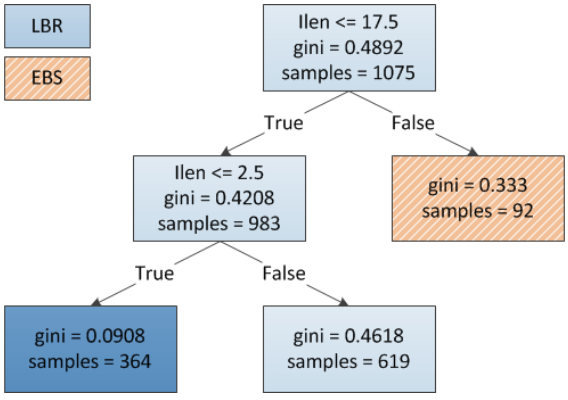

Figure 1: A decision tree generated from HBBP data. Figure abbreviated from Scikit output. "gini" stands for Gini Impurity, which (in general terms) is a measure of how often an element would be incorrectly labelled, if labelled randomly. "Samples" represents the number of training examples in each node.

A. *Collector*

The simultaneous collection of EBS and LBR data is not supported by the Linux kernel. We therefore collect all data in LBR mode, with two collections running in parallel during a single execution of a program, triggering on two different PMU events. We take advantage of the fact that each CPU core has multiple PMU counters. We program two counters to collect LBR simultaneously – one sampling on an "Instructions Retired" event and another on a "Branches Taken" event. We use the former as our EBS data source, and the latter as our LBR data source.

The hardware that gathers LBR samples also gathers additional information, including the "eventing IP", the IP on which the hardware thinks a PMU overflow occurred (much like the IP collected in EBS mode). On interrupts triggered by the "Instructions Retired" event, we collect this IP, which becomes our EBS data source. IPs gathered in this way are used as they would be in standard EBS mode collection. LBR records produced by the PMU on interrupts triggered by the "Instructions Retired" event are discarded during analysis.

On interrupts triggered by the "Branches Taken" event we store the LBR records, later discarding any other information, including the "eventing IP". This is our LBR data source.

While rather unorthodox by standard PMU use methodology, this approach works correctly. As a result, the workload needs to be run only once, the performance impact of the collection remains low, and the output file contains the required two types of data.

The exact events used are the following:

- **`INST_RETIRED:PREC_DIST`** for EBS collection (LBR information discarded)
- **`BR_INST_RETIRED:NEAR_TAKEN`** for LBR collection (IP information discarded)

The sampling periods have some influence on the accuracy as well as on the runtime overhead. Following recommendations from Nowak et al. [11] and additional observations, we choose the values for the two respective events depending on the runtime of the workload (see Table 4). LBR sampling is done with a smaller period than EBS sampling, because LBR data collection only happens on branches taken, which are less frequent than all instruction retirements, on which EBS samples. The memory and performance overhead of our collector could be optimized once simultaneous EBS and LBR collections are supported by the Linux kernel, no longer requiring two parallel LBR collections.

The collector gathers raw data from "perf" at runtime, which is later processed to extract EBS and LBR samples, as well as to include LBR bias information.

Table 4: EBS and LBR sampling periods in HBBP

Runtime	EBS sampling period	LBR sampling period
Seconds	1 000 037	100 003
~1-2 minutes	10 000 019	1 000 037
Minutes (SPEC workloads)	100 000 007	10 000 019

Additional data collected in the **`perf.data`** file includes process events (e.g. **`fork`**, **`exec`**, etc.) as well as memory map changes for subsequent virtual to physical address conversion.

As in standard "perf", the typical size of the raw data files goes up to a GB for a given workload. Post-analysis files used to generate HBBP views take around 10MB of space per workload. Both the user space (Rings 1-3) and the kernel (Ring 0) are monitored.

B. Analyzer

Analysis software, developed in Python and C for speed, produces dynamic instruction mixes from raw sample input by processing additional static information. The analyzer caches key information, including samples or disassembly, analyzing most workloads in a minute or less.

We implement a custom disassembler based on XED, the "X86 Encoder Decoder Software Library" [21]. This choice is dictated by the necessity to extract detailed opcode information and to achieve analysis speed suitable for interactive use.

Dynamic (sample) information is mapped onto static basic block maps. Using the adjusted sample data, we produce a histogram of BBECs according to HBBP.

The final instruction mix data is output as a pivot table, a format frequently used for exploratory data analysis, with user-configurable headers and values in tables. It gives complete analysis freedom to the user and facilitates machine processing or report generation. Custom or traditional views such as top functions, top mnemonics, or instruction family breakdowns, are produced in a few clicks. Data can be filtered, aggregated or broken down using different granularity levels: by thread ID, binary module, symbol (function), basic block or source line. Furthermore, to enhance analysis capabilities, the disassembly is annotated with static properties of the instructions within, such as:

- the instruction class, ISA, family and category,
- types, numbers, sizes and attributes of operands.

In addition to using direct attributes, we generate secondary instruction attributes such as memory read and write flags, packed and scalar flags, etc. We also enable the easy creation of custom instruction taxonomies based on instruction properties. For instance, a user-defined instruction group called "long latency instructions" would contain instructions such as **`DIV, SQRT,`** "**`XCHG R,M`**", or a group called "synchronization instructions" would have items such as XADD, LOCK variants [22]. This seamless mixing of dynamic and static information enables easy customization and shortens the time to solution for practitioners, because it becomes easier to tell which parts of the code are of interest.

VI. Examining errors

A. Reference definition

To provide information useful to programmers, we focus on *instruction mnemonics*. The baseline reference method in terms of speed and accuracy is software instrumentation, which maintains an internal histogram of every instruction the workload under test executes. Therefore, the number of executions per mnemonic is expected to be accurate, and this number is used as the ground truth value.

B. Error metric definition

When discussing "error", we refer to the difference between the reference (V_{ref}) and measured ($V_{measured}$) values (i.e., absolute inaccuracy) divided by the reference value, *for every instruction mnemonic M*. We thus obtain as error a percentage of the reference value that is over- or undercounted in the measurement.

$$Error(M) = \frac{abs(V_{ref}(M) - V_{measured}(M))}{V_{ref}(M)}$$

Therefore, if we obtain a reference value of 500 executions of MOV, and measure 510 executions of MOV with HBBP, the error for that mnemonic is reported as 10/500 = 2%.

This metric is relevant, because ultimately it is the number of mnemonics of a specific kind that is interesting to the user. Later these numbers can be combined in various formulas or ratios (e.g., the ratio of computational to noncomputational instructions).

For aggregated results, we use a derived measure. This metric provides information about the practical runtime relevance of observed errors. The *average weighted error* is the sum of errors for each mnemonic M multiplied by its frequency of its occurrence in a given workload:

$$Avg.\ w.error = \sum_{all\ M} Error(M) * \frac{V_{ref}(M)}{\#instructions_{ref}}$$

VII. Experimental setup

A. Hardware setup

We evaluate our approach on an Intel Xeon E5-2695 v2 processor ("Ivy Bridge"). This choice is dictated by LBR support in both hardware and software at the time of writing, as well as by support for a "Precisely Distributed Instructions Retired" event, **`INST_RETIRED.PREC_DIST`**. We

stabilize the system for benchmarking. Among other things, we disable frequency scaling, "turbo mode" and C-states.

B. *Software*

We use a Linux kernel from the 4.7.2 branch on a 64-bit RHEL6-compatible system. We disable the NMI watchdog and all nonessential daemons. We also adjust the maximum sample rate of perf in order to avoid overloading the system with samples (throttling), which could generate incorrect results.

We obtain reference results from the unmodified Intel PIN tool [23], in the Intel Software Development Emulator (SDE), v. 7.39 [24]. It is the industry-standard tool that we find to be most robust, working well with large workloads, and capable of following execution chains (e.g., **`execve`**). We check PIN results against instruction-specific PMU counts and PMU-reported total instruction counts, and find that they match. Like other mainstream instrumentation tools and earlier work (such as EEL [25]), PIN works in user mode and cannot capture kernel samples. To remain fair, except in Section VIII.D, our accuracy comparisons consider only user mode instructions.

VIII. EXPERIMENTAL RESULTS

We first compare the runtime and the accuracy of HBBP and software instrumentation. We then compare the accuracy of HBBP to the accuracy of EBS and LBR used in isolation. For these comparisons we use the SPEC2006 benchmark suite and workloads from two large-scale scientific codes. We also use one of these workloads to show a practical use case of HBBP. We conclude with a demonstration that HBBP is indeed capable of providing instruction mixes for kernel code.

A. *SPEC CPU2006*

Our experiments with HBBP, repeated three times on the whole suite, last for 4 hours and 25 minutes on average, which is a 0.5% time penalty vs. a clean run, and close to the natural float in SPEC runtimes. The same tests take 18 hours 10 minutes for SDE, a fourfold increase. The maximum slowdown, 12.1x, is observed on povray (see Figure 2).

Figure 2 also shows average weighted errors for individual benchmarks.[2] The overall average weighted error for HBBP is 1.83%, with errors on individual benchmarks ranging from 0.2% to 4.4%. The overall average weighted errors for LBR and EBS are 3.15% and 4.43%, respectively.

Errors for either EBS or LBR are at least 2x larger than HBBP errors in 2/3 of the cases, and at least 3x larger in 1/4 of the cases. In extreme cases, EBS is 5.3x worse (HMMER) and LBR 8x worse (GAMESS). In only one case, LBM, HBBP is worse than LBR, where it has a 1.1% error, as opposed to 0.5% for LBR. Aside from the fact that errors are very small in both cases, this result stems from a code sequence in which long latency instructions (disturbing EBS measurements) immediately precede a long basic block. The considerable length causes HBBP to choose EBS as data source.

[2] SDE produces incorrect results for x264ref, as evidenced by PMU counting verification. X264ref is removed from the calculation of the average weighted error. Results point to a bug in the PIN tool.

B. *Particle simulation (Test40)*

Test40 is an application built on a scientific toolkit called Geant4 [26, p. 4], written in C++ and commonly used to simulate the passage of particles through matter. Geant4 is used in aerospace, medicine and particle physics. We choose it, because it represents an important class of complex, object-oriented workloads that process data for the Large Hadron Collider experiments at CERN, while running in multiple copies on up to 500'000 cores. It is also an appropriate test: it is difficult to deal with using EBS,

Figure 2: A comparison of SDE and HBBP overhead, and average weighted errors for HBBP, LBR and EBS on SPEC2006

	SDE overhead	HBBP overhead	HBBP average weighted error	LBR average weighted error	EBS average weighted error
400.perlbench	540%	0.1%	2.2%	4.6%	5.5%
401.bzip2	209%	0.3%	1.9%	2.5%	4.8%
403.gcc	491%	0.2%	3.5%	4.0%	7.0%
410.bwaves	161%	0.3%	0.9%	1.4%	1.3%
416.gamess	369%	0.0%	0.5%	4.0%	1.3%
429.mcf	176%	1.8%	3.0%	3.3%	11.9%
433.milc	193%	0.6%	0.9%	1.4%	2.3%
434.zeusmp	79%	-0.6%	0.6%	1.1%	1.3%
435.gromacs	100%	-0.7%	0.8%	5.2%	1.6%
436.cactusADM	4%	1.4%	0.2%	2.4%	0.2%
437.leslie3d	92%	0.6%	0.5%	0.8%	0.4%
444.namd	50%	0.1%	1.8%	3.7%	4.3%
445.gobmk	419%	-0.1%	2.1%	3.4%	6.3%
447.dealII	833%	0.2%	1.8%	2.2%	9.1%
450.soplex	822%	1.7%	2.1%	2.2%	10.4%
453.povray	1109%	-0.1%	3.8%	7.0%	5.5%
454.calculix	187%	0.1%	2.7%	3.2%	3.2%
456.hmmer	101%	0.1%	0.8%	2.0%	4.3%
458.sjeng	406%	-0.1%	3.3%	6.1%	4.5%
459.GemsFDTD	88%	0.4%	1.3%	1.3%	2.4%
462.libquantum	287%	1.5%	1.0%	1.0%	4.1%
464.h264ref	736%	0.1%			
465.tonto	682%	0.2%	1.9%	3.7%	2.3%
470.lbm	65%	2.1%	1.1%	0.5%	1.9%
471.omnetpp	656%	1.3%	3.3%	3.8%	8.5%
473.astar	230%	-0.1%	2.6%	7.7%	5.3%
481.wrf	284%	0.1%	1.6%	4.0%	2.3%
482.sphinx3	199%	0.3%	0.8%	1.0%	1.7%
483.xalancbmk	887%	0.1%	4.4%	4.5%	10.3%

because its methods are short. Test40 is also used for compiler studies and regression tests.

Table 5 presents the execution time penalties for running the application with HBBP and SDE, showing a 9-fold increase for SDE vs. a 2.3% increase for HBBP. The average weighted error for HBBP remains below 1%, demonstrating the good tradeoff achieved by HBBP between runtime overhead and accuracy.

Figure 3 presents the mnemonic frequencies obtained by HBBP for the top-20 instruction retiring mnemonics (bars, left axis), and their errors compared to SDE (dots, right axis). Figure 4 shows a comparison of errors per mnemonic between HBBP, EBS and LBR. For instance, for the top 5 instruction retiring mnemonics, LBR errors are between 4% and 7%, while for HBBP they are under 2%. Further down, EBS errors reach 15-25% for **`POP`**, **`RET_NEAR`** and **`JMP`**, while HBBP produces results with less than 1% error.

These results are not an isolated case, and they underline the need for HBBP, as opposed to raw EBS or LBR, even with custom enhancements applied.

Table 5: Test40 evaluation

	Clean	HBBP	SDE
Runtime [s]	27.1	27.7	277.0
Time penalty	N/A	2.3%	923%
Avg W Error	N/A	0.94%	0%

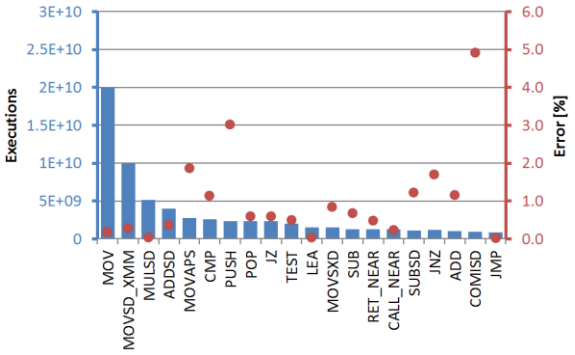

Figure 3: Test40 instruction execution counts (left) and error percentages (right), for the top 20 instruction executing mnemonics

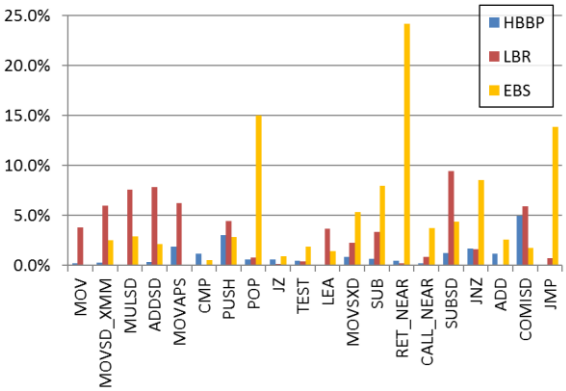

Figure 4: Test40 error percentages for HBBP, LBR and EBS, for the top 20 instruction executing mnemonics

C. *Fitter*

Fitter is a scientific program written in C++, fitting sparse position measurements into tracks of object movements in 3D space (related to [27]). It is representative of compact, high-performance code, that is both CPU-intensive and vectorizable. In production, this code runs in low-latency environments and must produce results within 1-2µs. However, with SDE, the three variants (x87 scalar, SSE, and AVX) of the application run 4-120x more slowly, increasing response time beyond production limits and necessitating a benchmark extraction.

In the SSE variant, we observe 13% errors on LBR, vs. 2-3% for EBS and HBBP. However, the same benchmark in AVX mode has 12% errors on EBS, vs. 2% for LBR and HBBP. Hence, neither EBS nor LBR alone can reasonably be used to study performance, while HBBP provides good accuracy for all versions of the benchmark.

When profiling code with profilers such as perf or Intel VTune, it is often clear where the time is spent, but not how. Instruction mixes can be particularly useful to study compute-intensive workloads and vectorization, as in this case.

The workload is examined in three variants, each having a different underlying structure for computation: x87 scalar, SSE and AVX single precision vectors. While working with a beta version of the Intel compiler, we noticed that AVX performance was significantly (20x) lower than expected from previous compilations. Expected values were determined using earlier compilations and runs, and supported with data from PIN. We suspected a compiler regression related to AVX instruction generation, and possible SSE-AVX transitions (which generate penalties on some CPUs). However, through the use of HBBP we concluded that the number of executed vector instructions was not suspicious. At the same time, the instruction mix showed a high number of call instructions, which in turn led us to trace the problem to the lack of inlining. The problem was thus indeed a compiler regression linked to AVX support, but not at all a problem with the emission of AVX instructions.

Table 6 presents our results obtained with this benchmark. The expected values are shown in the upper half of the table, and the measured values in the bottom half. Values for the problematic AVX code are shown in the column labelled “AVX”, while values for the fixed version in the rightmost column labelled “AVX fix”.

D. *Synthetic kernel benchmark*

Instruction mixes in kernel space might be of interest to device driver writers and OS architects. Such experts are particularly conscious of the code they write, as it is more difficult to debug, and the kernel environment puts constraints on code style (e.g., avoidance of floating point) and available compiler optimizations.

To show the validity of our approach in kernel space, we construct a small synthetic prime number search benchmark in user space. We then insert the same code into a live kernel as a device driver module, and trigger it from user space by reads. Calls to kernel code are separated in time to simulate real behavior. Table 7 shows instruction frequencies for the user-level code, obtained by both SDE and HBBP, and for the kernel-level code, obtained by HBBP. As can be seen, the results are in very good agreement. Results for EBS and LBR are not shown in detail, but EBS errors reach 15%, while LBR and HBBP errors are around 1%.

E. Other reports

We make the following additional short reports on *detection capabilities* in a concise manner:

- HBBP was used to correctly detect a vectorization opportunity and an issue with `#omp simd` reduction in CLForward, an online HPC code. HBBP signaled a large number of scalar instructions. Developers made the code more compiler-friendly, a large fraction of these scalar instructions were replaced by a smaller number of packed instructions, and performance improved by 8% (see Table 8)
- HBBP was used to search for suspicious convert instructions (e.g., CVTSI2SD) in random number generation. Ultimately, it was shown that contrary to a 30% penalty expectation, the issue had only a 5% impact. Optimization efforts moved elsewhere.
- HBBP was used to characterize heap pressure in the OS kernel on an HPC simulation. Developers remodeled `calloc()` calls and cut 15-20% system time to nearly 0.

IX. Related Work

BBECs can be extracted with minimal overhead at runtime using a variety of PMU-based sampling methods surveyed by Nowak et al. [11] and further discussed in this paper. Modern tuning methods, such as those implemented in Intel VTune or Gooda [28] use LBRs to generate partial call graphs and infer execution paths from the gathered data. However, we use LBR content and disassembly for BBECs, by sampling on an event which relates to the frequency of taken branches.

Ammons et al. [29] and Ball et al. [30] focus primarily on context information added to PMU counters through instrumentation, for the purpose of monitoring and predicting workload code paths. These methods may have overheads under 2x, but are not fully precise and change counter values during profiling. HBBP is a simpler, purely PMU-based approach and does not use software context information nor disturb the workloads (in particular the caches).

We use PMU *counting* for cross-reference. It has a number of documented, verified and understood issues, described in the works of Weaver [31]–[33], [34] and Mytkowicz [35], [36].

Table 6: Expected vs. Measured values (millions) for the Fitter benchmark. AVX fix denotes inlining fixed

		x87	SSE	AVX	AVX fix
Expected	x87 inst	512	374	367	367
	SSE inst	**10'898**	**2'724**	0	0
	AVX inst	0	0	**1'387**	**1'387**
	CALLs	107	106	99	99
	Time/track	1.71us	0.50us	0.38us	0.38us
Measured	x87 inst	493	362	3'425	397
	SSE inst	**10'886**	**2'736**	0	0
	AVX inst	0	0	**1'439**	**1'387**
	CALLs	103	100	6'150	97
	Time/track	1.73us	0.51us	7.78us	0.39us
	AvgW Err	0.96%	2.97%	1.78%	2.65%

Table 7: Instructions in the kernel sample (millions)

Method	**SDE**	**HBBP**	
Module	hello (user space)	hello.ko (kernel)	hello (user space)
Function	hello_u	hello_k	hello_u
ADD	1286	1289	1283
CDQE	57	55	53
CMP	550	547	545
IMUL	57	55	53
JLE	191	188	188
JNLE	57	55	56
JNZ	302	304	302
JZ	151	148	150
MOV	823	808	808
MOVSXD	191	188	188
SUB	191	188	188
TEST	151	148	150
Total	4005	3972	3964

Table 8: HBBP view of CLForward vectorization (billions of instructions). A large number of scalar instructions has been replaced by a smaller number of packed (vectorized) ones.

INST SET	**PACKING**	**BEFORE**	**AFTER**
AVX		16.2	14.3
	NONE	0.0	3.3
	SCALAR	**14.7**	0.4
	PACKED	1.5	**10.6**
BASE		2.9	1.5
	NONE	2.9	1.5
TOTAL		**19.2**	**15.8**

X. Conclusions

In this paper we demonstrated HBBP, a method for obtaining dynamic instruction mixes in near real time, using modern PMUs. HBBP does not disturb workloads in terms of the execution path nor runtime and is capable of providing instruction mixes also for code running in kernel space. HBBP collection incurs limited runtime overheads, below 1.3% on average, with an average error below 2.1% - suitable for tests in production environments and on applications with long runtimes.

ACKNOWLEDGMENT

We thank our colleagues for the invaluable input to this work: Omar Awile (CERN), Mirela-Madalina Botezatu (Google), Stephane Eranian (Google), Vincenzo Innocente (CERN), David Levinthal (Microsoft), Sebastien Valat (CERN), Liviu Valsan (CERN).

REFERENCES

[1] S. K. S. Ma and L. L. Wear, "Dynamic instruction set evaluation," 1974, pp. 9–11.

[2] R. L. Sites, "The Alpha AXP Architecture and 21064 Processor," *IEEE Micro*, 1993.

[3] R. Bhargava, L. K. John, B. L. Evans, and R. Radhakrishnan, "Evaluating MMX technology using DSP and multimedia applications," in *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, 1998, pp. 37–46.

[4] "AutoFDO - GCC Wiki." [Online]. Available: https://gcc.gnu.org/wiki/AutoFDO. [Accessed: 16-Nov-2016].

[5] A. Yasin, "A Top-Down Method for Performance Analysis and Counters Architecture," presented at the 2014 IEEE International Symposium Performance Analysis of Systems and Software (ISPASS), 2014.

[6] A. Nowak, D. Levinthal, and W. Zwaenepoel, "Hierarchical cycle accounting: a new method for application performance tuning," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, 2015, pp. 112–123.

[7] D. Levinthal, "Performance Analysis and software optimization for HPC on Intel Core i7, Xeon 5500 and 5600 family Processors," CERN, Jul-2010.

[8] J. Haj-Yihia, A. Yasin, Y. B. Asher, and A. Mendelson, "Fine-Grain Power Breakdown of Modern Out-of-Order Cores and Its Implications on Skylake-Based Systems," *ACM Transactions on Architecture and Code Optimization*, vol. 13, no. 4, pp. 1–25, Dec. 2016.

[9] Y. S. Shao and D. Brooks, "Energy characterization and instruction-level energy model of Intel's Xeon Phi processor," in *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, 2013, pp. 389–394.

[10] S. K. Sadasivam and S. T. Selvi, "Comparative performance study of SPEC INT 2006 benchmarks on nehalem, sandybridge and haswell microarchitectures," in *Computer, Information and Telecommunication Systems (CITS), 2015 International Conference on*, 2015, pp. 1–5.

[11] A. Nowak, A. Yasin, A. Mendelson, and W. Zwaenepoel, "Establishing a Base of Trust with Performance Counters for Enterprise Workloads," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, Santa Clara, CA, 2015, pp. 541–548.

[12] S. Moore, "A comparison of counting and sampling modes of using performance monitoring hardware," *Computational Science—ICCS 2002*, pp. 904–912, 2002.

[13] G. Bitzes and A. Nowak, "The overhead of profiling using PMU hardware counters," *CERN openlab report*, 2014.

[14] Intel Corporation, "PerfMon Events," *Intel Processor Event Reference*. [Online]. Available: https://download.01.org/perfmon/index/. [Accessed: 28-Feb-2018].

[15] D. Chen *et al.*, "Taming Hardware Event Samples for Precise and Versatile Feedback Directed Optimizations," *IEEE Transactions on Computers*, vol. PP, no. 99, p. 1, 2013.

[16] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984.

[17] N. S. Altman, "An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, Aug. 1992.

[18] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, Sep. 1995.

[19] L. Buitinck *et al.*, "API design for machine learning software: experiences from the scikit-learn project," *arXiv:1309.0238 [cs]*, Sep. 2013.

[20] S. Eranian, "perfmon2 - libpfm4," 11-Oct-2016. [Online]. Available: https://sourceforge.net/projects/perfmon2/files/libpfm4/. [Accessed: 11-Oct-2016].

[21] M. Charney, *Intel X86 Encoder Decoder Software Library*. 2016.

[22] A. Fog, "Instruction Tables." Technical University of Denmark, 09-Jan-2016.

[23] C.-K. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, 2005, pp. 190–200.

[24] "Intel Software Development Emulator." [Online]. Available: https://software.intel.com/en-us/articles/pre-release-license-agreement-for-intel-software-development-emulator-accept-end-user-license-agreement-and-download. [Accessed: 01-Feb-2016].

[25] J. R. Larus and E. Schnarr, "EEL: Machine-independent executable editing," in *ACM Sigplan Notices*, 1995, vol. 30, pp. 291–300.

[26] J. Apostolakis, "Geant4—a simulation toolkit," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 506, no. 3, pp. 250–303, Jul. 2003.

[27] I. Kisel, I. Kulakov, and M. Zyzak, "Parallel Implementation of the KFParticle Vertexing Package for the CBM and ALICE Experiments," in *Computing in High Energy and Nuclear Physics 2012*.

[28] Google, *Gooda - a pmu event analysis package (http://code.google.com/p/gooda/)*. 2012.

[29] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," *ACM Sigplan Notices*, vol. 32, no. 5, pp. 85–96, 1997.

[30] T. Ball and J. R. Larus, "Efficient path profiling," in *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, 1996, pp. 46–57.

[31] V. M. Weaver and S. A. McKee, "Can hardware performance counters be trusted?," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, 2008, pp. 141–150.

[32] V. Weaver, "Can Hardware Performance Counters Produce Expected, Deterministic Results?," presented at the 3rd Workshop on Functionality of Hardware Performance Monitoring, 2010.

[33] V. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, 2013.

[34] V. M. Weaver, "Self-monitoring overhead of the Linux perf_ event performance counter interface," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, 2015, pp. 102–111.

[35] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney, "We have it easy, but do we have it right?," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008, pp. 1–7.

[36] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!," in *ACM Sigplan Notices*, 2009, vol. 44, pp. 265–276.