Automated Data Analysis for Defining Performance Metrics from Raw Hardware Events

Daniel Barry, Anthony Danalis, Jack Dongarra
University of Tennessee Knoxville, USA
dbarry@vols.utk.edu
{*adanalis, dongarra*}*@icl.utk.edu*

2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 979-8-3503-6460-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/IPDPSW63119.2024.00134

***Abstract*—Hardware performance events are at the center of application performance analysis. However, the sheer volume of low-level hardware events in modern HPC systems is overwhelming, making them difficult for users to comprehend.**

Understanding which concepts are monitored by performance events can be achieved using a two-step process. The first step is the execution of benchmarks designed to stress different hardware attributes in isolation. For every hardware event we wish to understand, we execute the benchmarks while measuring the event. In the second step, the data produced by executing the benchmarks is analyzed to identify what each event actually measures. In this paper, we present the methodology for analyzing the data from four previously developed benchmarks that stress key hardware attributes—CPU and GPU floating-point units, branching units, and data caches—to map low-level hardware events to high-level programming concepts. We present an automated methodology to express the event data in a well-understood, conceptual basis. We implement a specialized pivoting scheme for QR factorization to identify events that provide distinct information from each other, and techniques for addressing noise in event measurements. Lastly, we utilize least-squares regression to combine the chosen events to define particular metrics of interest.

I. Introduction

Performance metrics in HPC systems are monitored by reading the occurrences of various hardware events. However, as a user transitions from one architecture to another, the mapping between raw performance events and the concepts they measure becomes increasingly ambiguous due to (i) different architectures containing differing sets of raw events and (ii) the vast amounts of events present in newer machines.

Modern HPC systems [1], [2], [3] are becoming more and more heterogeneous in their hardware constitution—*e.g.*, deeper and more nuanced memory hierarchies, custom network infrastructures, and GPU accelerators. As such, they contain on the order of hundreds of thousands of performance events related to the increasingly diverse array of hardware components. Furthermore, with the advent of software-defined events [4], there is an even greater number of more nuanced events present. While these events are documented to some extent in vendors' technical documents [5], they are not always described thoroughly.

This overall lack of clarity makes it challenging for users to comprehend the specific high-level programming concepts—such as total floating-point operations (FLOPs), bidirectional memory bandwidth, *etc.*—that events actually represent. Further exacerbating this issue, there are far fewer physical counters available than there are events, by several orders of magnitude. Therefore, measuring all events at once is not possible. Even if it was possible, the problem of determining meaningful combinations of the vast amounts of available events to define specific metrics is an intractable task to accomplish manually.

Middleware libraries, such as PAPI [6], serve as portability layers that provide definitions for performance-metric presets across diverse architectures using raw events. Since third-party performance tools—TAU [7], Score-P [8], Vampir [9], Caliper [10], *etc.*—utilize middleware layers to access hardware counters, being able to automatically define these performance metrics using available raw events can have a significant impact on the community. While the Counter Analysis Toolkit (CAT) [11] consists of benchmarks to discover the true concepts measured by raw events, it has still been necessary for the developers of PAPI to manually parse its output to define the presets.

The following contributions are achieved in this work.

- We express raw performance event data (from CAT benchmarks) as vectors and form them into data matrices both as-is and in hardware-attribute specific bases.
- We implement a special-purpose column-pivoted QR factorization (QRCP) for our data matrices to identify which raw events represent independent concepts from each other.
- We develop filtering mechanisms to suppress the noise present in event measurements to eliminate bogus outcomes from the linear algebra operations.
- We utilize least-squares regression to identify the best-fit combination of raw events needed to define high-level metrics of interest.
- We showcase how the analysis presented in this paper can be used to automatically define useful performance metrics for recent x86 CPUs and AMD GPUs.

We conduct experiments on two systems:

- **Aurora at the Argonne National Laboratory:** compute nodes contain Intel Sapphire Rapids CPUs. [12]
- **Frontier at the Oak Ridge National Laboratory:** compute nodes contain AMD MI250X GPUs. [1], [13]

979-8-3503-6460-6/24/$31.00 ©2024 IEEE
DOI 10.1109/IPDPSW63119.2024.00134

We use the Aurora supercomputer to collect event measurements from the CAT CPU-FLOPs, branching, and data cache benchmarks. To verify that our methods hold for an entirely different category of compute hardware, we collect event measurements by running a new GPU-FLOPs benchmark on the Frontier supercomputer.

II. Event Analysis Methodology

Suppose a programmer is interested in monitoring the number of double-precision floating-point operations performed by their code. For brevity, we will refer to these operations as DP FLOPs. However, many architectures—such as Intel Sapphire Rapids—do not include a raw event that measures DP FLOPs. Therefore, this quantity has to be constructed by combining measurements from existing events that measure more detailed concepts, such as the floating-point instructions of a particular AVX type (*e.g.,* 256-bit AVX). Combining such events requires that we identify all the relevant events, but it also involves an additional problem that must be addressed. Specifically, these raw events measure *instructions*, not *operations*. In some cases, such as scalar addition and subtraction, instructions and operations have the same count, but instructions such as fused multiply-add (FMA) perform two operations for every one instruction, and the aforementioned 256-bit AVX instructions perform four FLOPs for every instruction. Therefore, to form the concept of operations, the existing events first have to be scaled and then added together.

The challenge now becomes identifying all the events that measure the independent floating-point instruction concepts. We do this by running a series of microkernels wherein each microkernel performs a known, expected number of a specific type of floating-point instructions, such as single-precision scalar, or double-precision AVX256 FMA, *etc.* For every such microkernel execution, we measure the response of all raw events found on the target hardware.

Let us assume that the benchmark contains ten kernels. Then for every raw event e_i, we will get a vector of length ten with values that correspond to the measurements of e_i for the ten kernels. Concatenating the vectors for all events would produce a matrix `A`. Ideally, we could turn our search for a DP FLOPs metric into a linear algebra problem. To do so, we first have to handcraft the vector that DP FLOPs would measure for our ten kernels, if such an event existed on the hardware. We will refer to this handcrafted vector as the *signature* of this metric. Now, we can use the matrix `A` which contains the real measurements of the raw events and solve the problem:

$$A \cdot x = b$$

where b is the signature of the metric we are interested in. Solving this problem would result in a vector of coefficients, x, that would tell us which columns of `A` need to be combined to form the signature of the metric we seek, and by what factor they need to be scaled. Since the columns of `A` directly correspond to raw events found on the target hardware, solving this problem would tell us how to compose the metric of interest using existing raw events.

However, this problem cannot be solved as such, for multiple reasons. First of all, the matrix `A` is singular. That is, it contains multiple columns of only zeros, since multiple raw events will not measure anything that relates to kernels with floating-point operations (*e.g.,* events that measure TLB misses). Even if we removed those columns in a filtering step, there will be other non-zero columns that will appear multiple times, columns that are scaled versions of other columns, and columns that are linear combinations of other columns. For example, events that measure integer operations or branch operations would cause this for the floating-point kernels, since the headers of the loops will contain integer operations and branches. In theory, such columns could also be filtered out of `A` using a matrix orthogonalization algorithm, such as QR. However, noise in the measurements can hide linear dependencies between columns, or create bogus dependencies between columns that ought to be independent. For example, the vectors $(1,1)$ and $(0.99,1.01)$ are numerically linearly independent (since one cannot be expressed as a scaled version of the other) but semantically they are the same vector if their difference is solely due to noise. Besides noise, the vast divergence of scale between different columns would cause QR to pick irrelevant events in the resulting matrix. For example, events measuring cycles would lead to columns that have a significantly larger norm than the column resulting from floating-point events. Since the norm of a vector is the criterion that QR typically uses to select vectors, the resulting matrix would contain many irrelevant events.

In the following sections, we describe the analysis and data manipulation we performed on our measurements in order to overcome all of these difficulties and produce meaningful combinations of raw events that define the metrics in which programmers are interested.

III. Structure of CAT Benchmark Data

The first microkernel in the CAT CPU-FLOPs benchmark contains three loops as shown in Figure 1. The loops contain 24, 48, and 96 double-precision scalar instructions respectively. Let us refer to this kernel as K^{SCAL}. A second microkernel has the same structure, but contains loops with 12, 24, and 48 AVX256 fused multiply-add instructions. Let us refer to this one as K^{256}_{FMA}.

In the interest of readability, in the following discussion we will assume that the target platform only has two types of floating-point instructions, double-precision scalar non-FMA, and double-precision AVX256 FMA.

A. Event Signatures

A metric that performance analysts are often interested in is double-precision floating-point operations, DP FLOPs. However, this metric does not correspond to any raw event in most existing hardware. Therefore, we must compose it by combining existing raw events. The challenge is to identify which existing raw events we need to use and how they need to be scaled in order to properly compose the event in which we are interested.

```
PAPI_start();
for ( iter = 0; iter < 1e6; iter++ ){
  result += DP_SCAL_mul(a, b);        } Block x12 times
  result += DP_SCAL_add(result, c);   } Instructions: 24
  ...
}
PAPI_stop();

PAPI_start();
for ( iter = 0; iter < 1e6; iter++ ){
  result += DP_SCAL_mul(a, b);        } Block x24 times
  result += DP_SCAL_add(result, c);   } Instructions: 48
  ...
}
PAPI_stop();

PAPI_start();
for ( iter = 0; iter < 1e6; iter++ ){
  result += DP_SCAL_mul(a, b);        } Block x48 times
  result += DP_SCAL_add(result, c);   } Instructions: 96
  ...
}
PAPI_stop();
```

Fig. 1: Double-precision scalar floating-point kernel, K^{SCAL}.

On the simplified target hardware that we mentioned above, the values of this metric during the execution of the K^{SCAL} kernel would be (24,48,96) per iteration, for the three loops of the kernel, since each DP instruction in the kernel performs a DP operation. In contrast, when executing the K^{256}_{FMA} kernel, we would expect the FLOP counts for the three loops to be (96,192,384) since each AVX256 FMA instruction performs eight FLOPs. The measurements from these two kernels concatenated together would form the vector (24,48,96,96,192,384), which we will refer to as the *signature* for DP FLOPs.

Since this event does not exist on the target hardware though, the goal now is to form this signature by adding together scaled measurement vectors of events that actually exist on the target architecture. Let us consider that the target hardware has a raw event that measures only DP scalar non-FMA instructions, and another that only measures DP AVX256 FMA instructions (both of which are common in real hardware). If we monitor the scalar event while executing kernel K^{SCAL} followed by kernel K^{256}_{FMA}, we will obtain as output the vector (24,48,96,0,0,0), denoted by D^{SCAL}. If we monitor the AVX event while running the same two kernels, we will obtain as output the vector (0,0,0,12,24,48), denoted by D^{256}_{FMA}. Each AVX256 FMA instruction performs eight floating-point operations, so since we are interested in composing a FLOPs event, we need to scale D^{256}_{FMA} by a factor of eight. Scaling and adding these measurement vectors, as shown in Equation 1, produces the desired signature for all DP FLOPs from these two raw, instruction-counting events.

$$\text{All DP FLOPs} = D^{SCAL} + 8 \cdot D^{256}_{FMA}$$

$$\begin{pmatrix} 24 \\ 48 \\ 96 \\ 96 \\ 192 \\ 384 \end{pmatrix} = \begin{pmatrix} 24 \\ 48 \\ 96 \\ 0 \\ 0 \\ 0 \end{pmatrix} + 8 \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 12 \\ 24 \\ 48 \end{pmatrix} = \begin{pmatrix} 24 \\ 48 \\ 96 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 96 \\ 192 \\ 384 \end{pmatrix} \quad (1)$$

In this simplified example, we demonstrated how we can use the measurements we obtained from monitoring existing raw events while executing kernels to compose an event based on its "signature." Essentially, the signature describes what we expect the composed event to measure when executing a specific set of kernels.

However, the aforementioned description was an over-simplification of the floating-point hardware. In reality, there are more types of floating-point instructions than strictly scalar and AVX256 FMA in double-precision. To capture this complexity, we use more than the two kernels we discussed previously. Namely, we use kernels for scalar and all AVX vector widths, both FMA and non-FMA, and in both single- and double-precision: Space={scalar, 128, 256, 512}×{FMA,non-FMA}×{SP,DP}

B. Event Measurement Normalization

We use the term *expectation* to refer to the vector of expected measurements when monitoring an ideal event while executing all those kernels in sequence. In the above example, the expectation vectors were D^{SCAL} and D^{256}_{FMA}. Note that we used the term "ideal event," as opposed to "raw event," because some of the expectation vectors might refer to events that do not exist in a target architecture. For example, several Intel processors only offer raw events that count both FMA and non-FMA instructions, instead of raw events that count strictly one or the other. Similarly, several AMD processors do not offer different events for strictly single-precision, or strictly double-precision instructions. In contrast, our expectation vectors span all of these ideal performance concepts.

Scaling and combining *all* the relevant expectations results in the correct DP FLOPs signature as follows:

$$1 \cdot \mathrm{D}^{\mathrm{SCAL}} + 2 \cdot \mathrm{D}^{128} + 4 \cdot \mathrm{D}^{512} + 8 \cdot \mathrm{D}^{512} + 2 \cdot \mathrm{D}^{\mathrm{SCAL}}_{\mathrm{FMA}} + 4 \cdot \mathrm{D}^{128}_{\mathrm{FMA}} + 8 \cdot \mathrm{D}^{256}_{\mathrm{FMA}} + 16 \cdot \mathrm{D}^{512}_{\mathrm{FMA}}$$

A different way to view this is that we use the expectation vectors to project the signature of the performance metric we are trying to compose (*e.g.,* DP FLOPs) onto a set of "ideal hardware dimensions" defined by the ideal events.

Combining the entire set of expectation vectors into a matrix forms an *expectation basis*, E, which we can use as a coordinate system to represent different events. For example, in the coordinate system formed by the expectation basis:

$$E = \begin{pmatrix} | & | & | & | & | & \cdots & | & | & \cdots & | & | & \cdots & | \\ \mathrm{S}^{\mathrm{SCAL}} & \mathrm{S}^{128} & \mathrm{S}^{256} & \mathrm{S}^{512} & \mathrm{D}^{\mathrm{SCAL}} & \cdots & \mathrm{D}^{512} & \mathrm{S}^{\mathrm{SCAL}}_{\mathrm{FMA}} & \cdots & \mathrm{S}^{512}_{\mathrm{FMA}} & \mathrm{D}^{\mathrm{SCAL}}_{\mathrm{FMA}} & \cdots & \mathrm{D}^{512}_{\mathrm{FMA}} \\ | & | & | & | & | & \cdots & | & | & \cdots & | & | & \cdots & | \end{pmatrix}$$

the DP FLOPs signature has the representation:

$$(0,0,0,0,1,2,4,8,0,0,0,0,2,4,8,16)$$

Note that half the values are zeros because they correspond to single-precision (SP) expectations, which do not contribute to the signature of DP FLOPs.

After we have established our expectation basis E for the FLOPs benchmark, we can obtain the representation for the measurement vector m_e of a raw event e by solving the linear algebra problem:

$$E \cdot x_e = m_e$$

where E is the expectation basis, and x_e is the resulting representation of m_e.

However, because this linear system is rectangular for some expectation bases, we solve it using least squares. If the least-squares error is too large, then an event cannot be sufficiently represented in the expectation space and therefore is disregarded from further analysis.

After using this process to produce the x_e vector for each event e, we concatenate all such vectors into a matrix X. Note that there is a distinct matrix X for each set of benchmark kernels (*i.e.,* events for FLOPs, branches, caches, *etc.* are handled independently). In Section IV, we describe how we suppress the noise of the vectors that we add into matrix X, and in Section V, we explain how we generate a new matrix $\hat{\mathrm{X}}$ that contains linearly independent columns of X so that we can use it to define useful metrics.

C. GPU FLOPs

The analysis we are describing in this paper is not limited to one type of events, nor only events that originate on the CPU, nor any other event-specific limitation. In this section, we utilize a GPU benchmark that stresses the floating-point units. This benchmark contains kernels that do one of addition, subtraction, multiplication, square root, and fused multiply-add. We tested these kernels on Frontier's AMD MI250X GPUs. The symbols we use in this expectation basis are denoted by T_P, where T is the type of operation—A, S, M, SQ, or F—standing for the aforementioned operations. P is the precision of the operation—H, S, or D—denoting half-, single-, or double-precision. Table II lists the signatures for floating-point metrics on the GPU. The portions of signatures corresponding to the FMA kernels are scaled by two because the kernels issue instructions, but an FMA is two arithmetic operations per instruction.

$$E_{\text{GPU\_FLOP}} = \begin{pmatrix} | & | & | & | & \cdots & | & \cdots & | & \cdots & | \\ \mathrm{A_H} & \mathrm{A_S} & \mathrm{A_D} & \mathrm{S_H} & \cdots & \mathrm{M_H} & \cdots & \mathrm{SQ_H} & \cdots & \mathrm{F_D} \\ | & | & | & | & \cdots & | & \cdots & | & \cdots & | \end{pmatrix} \quad (2)$$

D. Branching

We use the same notation as [14] for the branching basis. The symbols CE, CR, T, D, and M denote *Conditional Branches Executed, Conditional Branches Retired, Conditional Branches Taken, Unconditional (Direct) Branches, and Mispredicted Branches*, respectively.

Table III lists the signatures for branching metrics. *Conditional Branches Not Taken* is equivalent to *Conditional Branches Retired* minus *Conditional Branches Taken*. *Correctly Predicted Branches* is equivalent to *Conditional Branches Retired* minus *Mispredicted Branches*. All other signatures have a one-to-one mapping with the expectations.

$$E_{\text{branch}} = \begin{pmatrix} | & | & | & | & | \\ \mathrm{CE} & \mathrm{CR} & \mathrm{T} & \mathrm{D} & \mathrm{M} \\ | & | & | & | & | \end{pmatrix} = \begin{pmatrix} 2 & 2 & 1.5 & 0 & 0 \\ 2 & 2 & 1 & 0 & 0 \\ 2 & 2 & 2 & 0 & 0 \\ 2 & 2 & 1.5 & 0 & 0.5 \\ 2.5 & 2.5 & 1.5 & 0 & 0.5 \\ 2.5 & 2.5 & 2 & 0 & 0.5 \\ 2.5 & 2 & 1.5 & 0 & 0.5 \\ 3 & 2.5 & 1.5 & 0 & 0.5 \\ 3 & 2.5 & 2 & 0 & 0.5 \\ 2 & 2 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix} \quad (3)$$

E. Data Caches

The CAT data cache benchmark performs a pointer chase on a buffer. By controlling the size of the buffer, it can incur hits or misses on the different levels of the cache hierarchy. This benchmark uses multiple concurrent threads working independently on disjoint buffers to add more pressure on the memory subsystem than a single thread would impose. The following symbols are used to denote the expectations for the data caches: $\mathrm{L1_{DM}}$, $\mathrm{L1_{DH}}$, $\mathrm{L2_{DH}}$, and $\mathrm{L3_{DH}}$. In this basis, the symbols DM and DH denote *Demand Misses* and *Demand Hits*. The signatures for various data cache metrics are listed in Table IV.

IV. Noise Analysis

As mentioned in Section II, variability in event measurements due to noise is problematic. Thus, we filter such noisy events out of our analysis pipeline. To quantify the variability of an event, we collect the event's measurement vector from multiple repetitions of a CAT benchmark. We compute the root normalized mean-square error (RNMSE) between each pair of measurement vectors (m_e^i and m_e^j) and keep the maximum value. The maximum RNMSE is given by Equation 4. In this formula, m_e^i and m_e^j are measurement vectors, and N is the number of elements in each vector. When the denominator in this formula is zero, it means that the average value of an event's measurement vector ($\bar{m}_e^i$ or $\bar{m}_e^j$) is zero. If one of these two quantities is zero, then we define the variability to be one, corresponding to a 100 percent error.[1] We introduce a noise threshold, τ; if an event has a variability measure greater than the noise threshold τ, it is regarded as being too noisy to be reliably controlled by CAT benchmarks, and the event is not considered for further analysis.

$$\text{Max. RNMSE}(m_e) = \max_{i \neq j} \frac{\|m_e^i - m_e^j\|_2}{\sqrt{N * \bar{m}_e^i * \bar{m}_e^j}} \quad (4)$$

In Figure 2a, we show the max-RNMSE value for each event measurement from the CAT branching benchmark, sorted in increasing order. There is a cluster of events with a zero variability (these are plotted as machine epsilon on the y-axis for the sake of visualization on a logarithmic scale). From these results, we can see that setting τ to any value from

[1] If all measurements of an event are zero, then the event is discarded as irrelevant.

10^{-4} to 10^{-15} unambiguously divides the zero-noise events from the noisy events. For the experiments described in this paper, we chose the value 10^{-10} for τ. We discard events with noise above this threshold (indicated by the shaded regions in Figure 2). For events with noise below this threshold, we can keep the average measurement vector from all repetitions, or any one of the vectors since all vectors are identical.

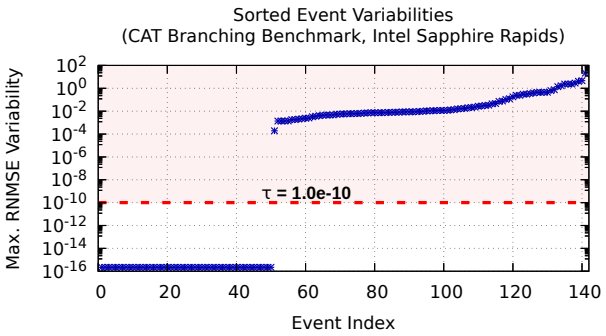
We repeat this analysis for the event measurements from the other CAT benchmarks, as shown in Figures 2b-2d. The event measurements from the CPU- and GPU-FLOPs benchmarks have a cluster of zero-noise events, similarly to those from the branching benchmark. Therefore, for these two benchmarks, we also set τ to 10^{-10}. For the data cache benchmark measurements, the value of τ is not as clear as it is for the other benchmarks, due to the unpredictability and complex behavior of the cache hierarchy. However, from empirical observations, we found that setting τ to 10^{-1} is sufficient. This is true because this filtering step is only meant to reduce the amount of noisy, irrelevant measurements that will proceed to the next stage of the analysis. Choosing a lenient filtering threshold leading to false positives is better than not applying this step at all.

These results show that τ can vary depending on the part of the hardware to which a class of events relates, because different hardware components exhibit different levels of noise. Other studies [15] have also found that certain hardware counters are less prone to noise than others. For most classes of events, the separation between noisy and noise-free events is clear, and choosing a cutoff threshold does not require a very careful selection process. For the cache events, where noise is more prominent, we choose a lenient cutoff threshold to only filter out the noisiest events, because in a subsequent step we will use multiple measuring threads and select the median measurement among them to further suppress noise.

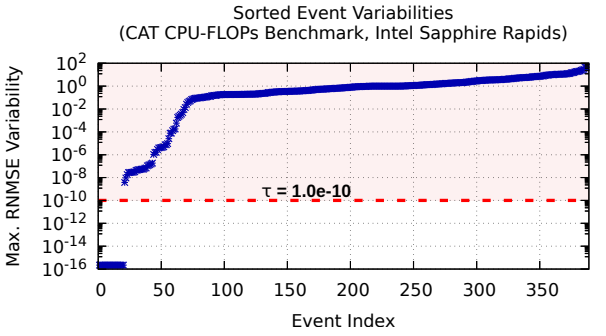
V. Specialized QRCP Factorization

As we mentioned before, in order to define useful metrics from the raw events, we need the columns of matrix `X` to be linearly independent. The mathematical reason behind this requirement is that if there are linearly dependent columns in a matrix `A`, then when trying to solve the system $A \cdot x = b$, there can be infinite solutions. However, for the purposes of defining a performance metric, there is only one solution (*i.e.,* one combination of raw events) that is the most meaningful. Another way to view this is that linearly independent event representations are semantically equivalent to events that provide distinct information from each other, and therefore are best equipped to construct higher-level metrics.

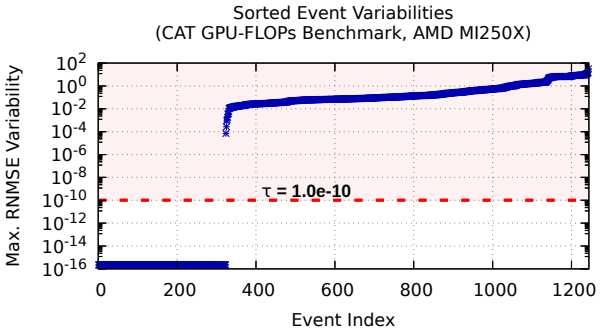
QRCP is an orthogonal matrix factorization that provides a linearly independent subset of a matrix's columns. Algorithm 1 outlines the basic steps in computing the QRCP, where matrix `X` is the input matrix. As can be seen in the listing, π is the array containing the permuted column-indices of `X`. We say that a matrix has rank k if it has k linearly independent columns. In this case the first k entries of π will correspond to the linearly independent columns, and the others will correspond to the



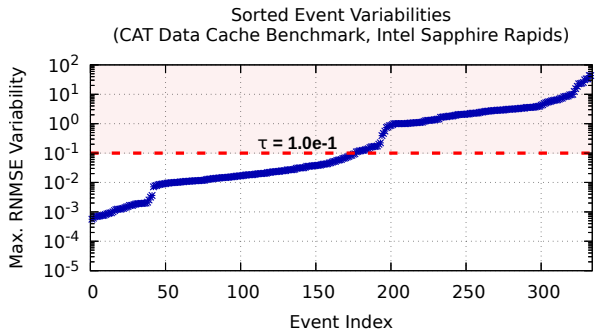
(a) CAT branching benchmark.



(b) CAT CPU-FLOPs benchmark.



(c) CAT GPU-FLOPs benchmark.



(d) CAT data cache benchmark.

Fig. 2: Event variabilities in CAT benchmarks.

remaining columns (which are linearly dependent on the first k).

Algorithm 2 is our modified version of the QRCP. Our modifications are focused on the pivot step, so that the linearly independent columns that are chosen by the algorithm best fit the needs of our analysis. Our modified algorithm takes matrix X as input and provides the permutation array as output; however, it also takes a tolerance α as input. The primary difference between Algorithms 1 and 2 is that Algorithm 2 uses a special pivoting scheme (Line 3) that prioritizes columns that are closer to the expectations in the basis. In other words, columns that contain a few values of one and multiple values of zero are prioritized over columns that contain an assortment of values that differ from one and zero. In contrast, in the standard QRCP, the pivot is chosen as the column with the largest norm [16], which is the opposite of what we want for our analysis. Regardless of the pivoting scheme, linear independence of the resulting events is guaranteed by the orthogonalization inherent to QR.

For this scheme to be practical, we introduce the tolerance α to account for noise in event measurements. Each element u of X ($u := X_{ij}$) is rounded to the closest integer within a tolerance of α using the formula:

$$R(u) = \alpha \cdot \lfloor \frac{u}{\alpha} + 0.5 \rfloor$$

After rounding the values in X, the pivoting scheme scores each column in X as follows. Every element $v := |X_{ij}|$ of a column j contributes to the pivoting score of the column based on the following formula:

$$Sc(v) = \begin{cases} v, & \text{if } v \geq 1 \\ 1/v, & \text{if } 0 < v < 1 \\ 0, & \text{if } v = 0 \end{cases}$$

The global minimum score is tracked along the way. If multiple columns have this minimum score, then the tie is broken by choosing the column in X with the smallest norm. However, if the norm of a column is smaller than a threshold β (where we define β to be the norm of the vector in which each element is α), then this column is disregarded. This ensures that columns close to the zero-vector are not chosen as a pivot. If all pivot candidates have a norm that is smaller than β, then the algorithm terminates (in Algorithm 2, we show this as setting the pivot to -1).

To give an example of the two operations of the two previous formulas, for $\alpha = 0.01$, the vector: $(1.002, 0.001, -0.5, 1.5)$ would have the score:

$$1 + 0 + \frac{1}{0.5} + 1.5 = 4.5$$

Using the rounding and scoring formulas on the matrix X, followed by the modified QR that we described, results in a matrix $\hat{X}$ which is either square or overdetermined[2], and has linearly independent columns.

[2]The matrix has at least as many rows as it has columns.

Algorithm 1 QRCP.

Input $A \in \mathbb{R}^{m \times n}$
Output $\pi \in \mathbb{N}^n$

1: $\pi \leftarrow [1, \ldots, n]$
2: **for** $i = 1, \ldots, n$ **do**
3: pivot $\leftarrow \text{argmax}_{i \leq j \leq n} \|A_{i:m,j}\|_2$
4: $A_{:,\text{pivot}} \leftrightarrow A_{:,i}$
5: $\pi_i \leftrightarrow \pi_{\text{pivot}}$
6: Update A using column *pivot*.
7: **end for**

Algorithm 2 QRCP with Specialized Pivoting Scheme.

Input $A \in \mathbb{R}^{m \times n}, \alpha \in \mathbb{R}$
Output $\pi \in \mathbb{N}^n$

1: $\pi \leftarrow [1, \ldots, n]$
2: **for** $i = 1, \ldots, n$ **do**
3: pivot $\leftarrow$ get_pivot(A, π, i, α)
4: **if** pivot == -1 **then**
5: BREAK
6: **end if**
7: $A_{:,\text{pivot}} \leftrightarrow A_{:,i}$
8: $\pi_i \leftrightarrow \pi_{\text{pivot}}$
9: Update A using column *pivot*.
10: **end for**

A. CPU FLOPs

Setting $\alpha = 5 \times 10^{-4}$, Algorithm 2 resulted in an $\hat{X}$ whose columns correspond to the following events:
`FP_ARITH_INST_RETIRED:[128|256|512]B_PACKED_[SINGLE|DOUBLE]` and `FP_ARITH_INST_RETIRED:SCALAR_[SINGLE|DOUBLE]`.
These events chosen by the QR are "good" in the sense that they closely correspond with the expected event occurrence patterns and therefore the intended, targeted attributes of the floating-point units.

B. GPU FLOPs

By setting $\alpha = 5 \times 10^{-4}$, Algorithm 2 identified the following key FLOPs events for the GPU:
`SQ_INSTS_VALU_[ADD|MUL|TRANS|FMA]_F[16|32|64]`.[3]
The events `SQ_INSTS_VALU_ADD_F[16|32|64]` occur in equivalent amounts for addition and subtraction kernels, indicating that it counts both types of operations.

C. Branching

After setting $\alpha = 5 \times 10^{-4}$, Algorithm 2 found the events:

- `BR_MISP_RETIRED`,
- `BR_INST_RETIRED:COND`,
- `BR_INST_RETIRED:COND_TAKEN`,
- `BR_INST_RETIRED:ALL_BRANCHES`.

[3]These events are prefixed with 'rocm:::' and suffixed with 'device=0' in PAPI; the 'device' qualifier can assume the value of 0-7 on Frontier since there are 8 GPU devices per node, but we need only define metrics for a single device. These are excluded from the above text for the sake of readability.

D. Data Caches

Setting $\alpha = 5 \times 10^{-2}$, Algorithm 2 chose the events:

- `MEM_LOAD_RETIRED:L3_HIT`,
- `L2_RQSTS:DEMAND_DATA_RD_HIT`,
- `MEM_LOAD_RETIRED:L1_MISS`,
- `MEM_LOAD_RETIRED:L1_HIT`.

E. Threshold Sensitivity

The threshold α that we used for the data cache events was chosen to be higher than the other event categories because α is a noise tolerance threshold and, as we discussed in Section IV, the cache events exhibit higher levels of noise than all other events. The actual value of the threshold is chosen empirically, but it does not have to be a perfect "magic" value. A wide range of values for α lead to the creation of a matrix $\hat{X}$ that contains events that properly capture the behavior of the hardware component that is tested by the corresponding kernels.

VI. Defining Useful Metrics

Since we have the linearly independent events from the analysis performed in Section V, we are able to meaningfully solve the system of the form $\hat{X}y = s$, where $\hat{X}$ is the matrix of events chosen by the QR, and s is the signature for a metric that we want to compose, such as those in Tables I-IV.

TABLE I: CPU FLOPs Metric Signatures

Performance Metric	Signature (S^{SCAL},...,D^{512}_{FMA})
SP Instrs.	(1,1,1,1,0,0,0,0,2,2,2,2,0,0,0,0)
SP Ops.	(1,4,8,16,0,0,0,0,2,8,16,32,0,0,0,0)
SP FMA Instrs.	(0,0,0,0,0,0,0,0,2,2,2,2,0,0,0,0)
DP Instrs.	(0,0,0,0,1,1,1,1,0,0,0,0,2,2,2,2)
DP Ops.	(0,0,0,0,1,2,4,8,0,0,0,0,2,4,8,16)
DP FMA Instrs.	(0,0,0,0,0,0,0,0,0,0,0,0,2,2,2,2)

TABLE II: GPU FLOPs Metric Signatures

Performance Metric	Signature (A_H,...,F_D)
HP Add Ops.	(1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
HP Sub Ops.	(0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0)
HP Add and Sub Ops.	(1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0)
All HP Ops.	(1,0,0,1,0,0,1,0,0,1,0,0,2,0,0)
All SP Ops.	(0,1,0,0,1,0,0,1,0,0,1,0,0,2,0)
All DP Ops.	(0,0,1,0,0,1,0,0,1,0,0,1,0,0,2)

TABLE III: Branching Metric Signatures

Performance Metric	Signature (CE,CR,T,D,M)
Unconditional Branches.	(0,0,0,1,0)
Conditional Branches Taken.	(0,0,1,0,0)
Conditional Branches Not Taken.	(0,1,-1,0,0)
Mispredicted Branches.	(0,0,0,0,1)
Correctly Predicted Branches.	(0,1,0,0,-1)
Conditional Branches Retired.	(0,1,0,0,0)
Conditional Branches Executed.	(1,0,0,0,0)

TABLE IV: Data Cache Metric Signatures

Performance Metric	Signature ($L1_{DM}$, $\cdots$, $L3_{DH}$)
L1 Misses.	(1,0,0,0)
L1 Hits.	(0,1,0,0)
L1 Reads.	(1,1,0,0)
L2 Hits.	(0,0,1,0)
L2 Misses.	(1,0,-1,0)
L3 Hits.	(0,0,0,1)

We solve this system in the following subsections to define metrics as combinations of raw events. If the QR detected fewer linearly independent events than there are expectations in the basis, then $\hat{X}$ will have more rows than columns. Since the system is rectangular in this case, we solve the it using least squares.

$$\text{Backward Error} = \frac{\|\hat{E}y - s\|_2}{\|\hat{E}\|_2 \cdot \|y\|_2 + \|s\|_2} \tag{5}$$

The fitness of a least-squares solution is given by the backward error, provided in Equation 5. [16] This error is listed in Tables V-VII for each signature's least-squares approximation.

A. CPU FLOPs

Using the events chosen by the QR from Section V and the signatures from Table I, we utilize least squares to define the floating-point metrics in Table V. The number of instructions for both single- and double-precision are simply the sum of the scalar and vector events for the respective precision. The number of operations is a weighted sum of the same events. For the operations metrics, each event is scaled by the number of floating-point operands. The least-squares error for most of these metrics is extremely small, indicating that these are indeed good metric definitions. However, the error is relatively large for the FMA instructions metrics. Furthermore, the least squares gives unintuitive linear combination of events for these metrics. These results suggest that there do not exist dedicated FMA-counting events in this architecture, which we can verify by manually inspecting the list of raw events. Therefore, our

analysis correctly identifies both the existence and the absence of events that can compose desired performance metrics.

TABLE V: CPU Floating-Point Metrics

Metric	Combination of Raw Events	Error
SP Instrs.	1×FP_ARITH_INST_RETIRED:128B_PACKED_SINGLE + 1×FP_ARITH_INST_RETIRED:256B_PACKED_SINGLE + 1×FP_ARITH_INST_RETIRED:512B_PACKED_SINGLE + 1×FP_ARITH_INST_RETIRED:SCALAR_SINGLE	1.67e-16
SP Ops.	4×FP_ARITH_INST_RETIRED:128B_PACKED_SINGLE + 8×FP_ARITH_INST_RETIRED:256B_PACKED_SINGLE + 16×FP_ARITH_INST_RETIRED:512B_PACKED_SINGLE + 1×FP_ARITH_INST_RETIRED:SCALAR_SINGLE	6.05e-18
SP FMA Instrs.	0.8×FP_ARITH_INST_RETIRED:128B_PACKED_SINGLE + 0.8×FP_ARITH_INST_RETIRED:256B_PACKED_SINGLE + 0.8×FP_ARITH_INST_RETIRED:512B_PACKED_SINGLE + 0.8×FP_ARITH_INST_RETIRED:SCALAR_SINGLE	2.36e-1
DP Instrs.	1×FP_ARITH_INST_RETIRED:128B_PACKED_DOUBLE + 1×FP_ARITH_INST_RETIRED:256B_PACKED_DOUBLE + 1×FP_ARITH_INST_RETIRED:512B_PACKED_DOUBLE + 1×FP_ARITH_INST_RETIRED:SCALAR_DOUBLE	5.55e-17
DP Ops.	2×FP_ARITH_INST_RETIRED:128B_PACKED_DOUBLE + 4×FP_ARITH_INST_RETIRED:256B_PACKED_DOUBLE + 8×FP_ARITH_INST_RETIRED:512B_PACKED_DOUBLE + 1×FP_ARITH_INST_RETIRED:SCALAR_DOUBLE	1.69e-19
DP FMA Instrs.	0.8×FP_ARITH_INST_RETIRED:128B_PACKED_DOUBLE + 0.8×FP_ARITH_INST_RETIRED:256B_PACKED_DOUBLE + 0.8×FP_ARITH_INST_RETIRED:512B_PACKED_DOUBLE + 0.8×FP_ARITH_INST_RETIRED:SCALAR_DOUBLE	2.36e-1

B. GPU FLOPs

We repeat the least-squares analysis to define the GPU floating-point metrics in Table VI. The error for each of the *HP Add* and *HP Sub* metrics is relatively large. This again suggests that these metrics cannot be defined in isolation on this architecture; however, the metric of *HP Adds and Subs* is indeed defined. The metrics for *All Operations* for each precision are also defined using the results from the least squares. These metric definitions have very small errors.

TABLE VI: GPU Floating-Point Metrics

Metric	Combination of Raw Events	Error
HP Add.	0.5×SQ_INSTS_VALU_ADD_F16	4.14e-1
HP Sub.	0.5×SQ_INSTS_VALU_ADD_F16	4.14e-1
HP Add. and Sub.	1×SQ_INSTS_VALU_ADD_F16	5.55e-17
All HP Ops.	2×SQ_INSTS_VALU_FMA_F16 + 1×SQ_INSTS_VALU_MUL_F16 + 1×SQ_INSTS_VALU_TRANS_F16 + 1×SQ_INSTS_VALU_ADD_F16	2.39-17
All SP Ops.	2×SQ_INSTS_VALU_FMA_F32 + 1×SQ_INSTS_VALU_MUL_F32 + 1×SQ_INSTS_VALU_TRANS_F32 + 1×SQ_INSTS_VALU_ADD_F32	2.39e-17
All DP Ops.	2×SQ_INSTS_VALU_FMA_F64 + 1×SQ_INSTS_VALU_MUL_F64 + 1×SQ_INSTS_VALU_TRANS_F64 + 1×SQ_INSTS_VALU_ADD_F64	2.39e-17

C. Branching

We perform the least-squares analysis for the metrics listed in Table VII. All of these metrics, with the exception of *All Branches Executed*, can be defined for this architecture. The small errors and reasonable linear combinations of raw events produced by least-squares verify that these metrics are well defined; whereas, the near-zero coefficient in the last linear combination, along with the error having the maximum possible value (1) indicate the lack of raw events that can compose an *All Branches Executed* metric.

TABLE VII: Branching Metrics

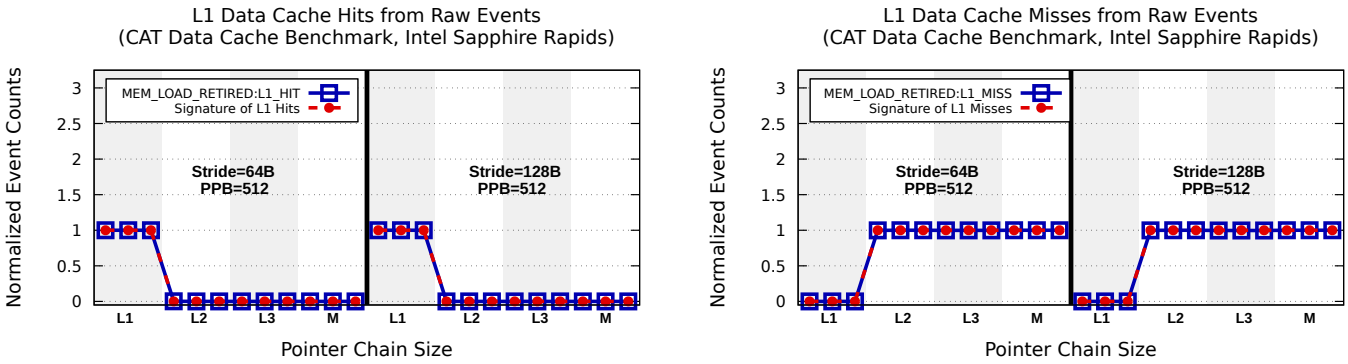
Metric	Combination of Raw Events	Error
Unconditional Branches.	-1×BR_INST_RETIRED:COND + 1×BR_INST_RETIRED:ALL_BRANCHES	4.03e-16
Conditional Branches Taken.	1×BR_INST_RETIRED:COND_TAKEN	2.15e-16
Conditional Branches Not Taken.	1×BR_INST_RETIRED:COND - 1×BR_INST_RETIRED:COND_TAKEN	2.35e-16
Mispredicted Branches.	1×BR_MISP_RETIRED	9.26e-17
Correctly Predicted Branches.	-1×BR_MISP_RETIRED + 1×BR_INST_RETIRED:COND	2.80e-16
Conditional Branches Retired.	1×BR_INST_RETIRED:COND	4.93e-16
Conditional Branches Executed.	2.22e-16×BR_MISP_RETIRED + 5.62e-16×BR_INST_RETIRED:COND + 1.53e-16×BR_INST_RETIRED:COND_TAKEN + 9.86e-32×BR_INST_RETIRED:ALL_BRANCHES	1.0

D. Data Caches

Table VIII shows the least-squares results for the data cache metrics. For all of these metrics, there is very little least-squares error. The coefficients are not exactly zero or one in the event combinations in Table VIII; however, this can be attributed to the noise from the data cache benchmark. Notice that the coefficients are either within 2% of one, or smaller than 5.87×10^{-3}. Therefore, we can easily round them to one or zero. If we round the coefficients to zero or one to form a new combination, then we can evaluate how well the combination compares to the signature. Figure 3 shows that rounding the coefficients from least squares provides an exact match for the signatures. This means that the least squares yields accurate raw-event combinations, even for the noisy data cache events. These results show that we must account for architectural nuance when forming signatures; we are able to discover valuable combinations of raw events to the extent that we understand the noise present in the hardware attributes of a given architecture.

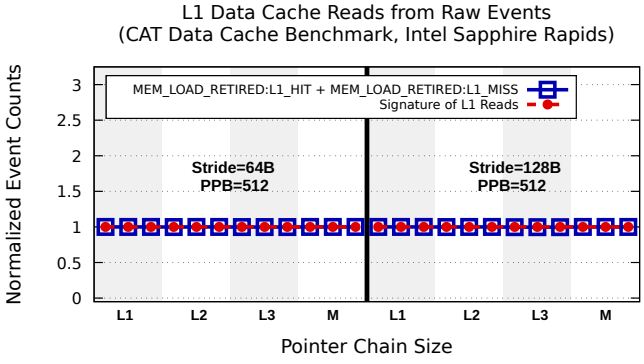
VII. Conclusion and Future Work

In this paper, we have introduced an automated mathematical analysis to parse through thousands of events and identify those most pertinent to specific hardware components. We

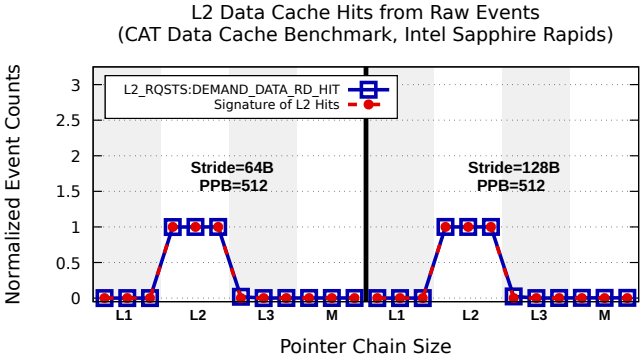


(a) L1 Hits.

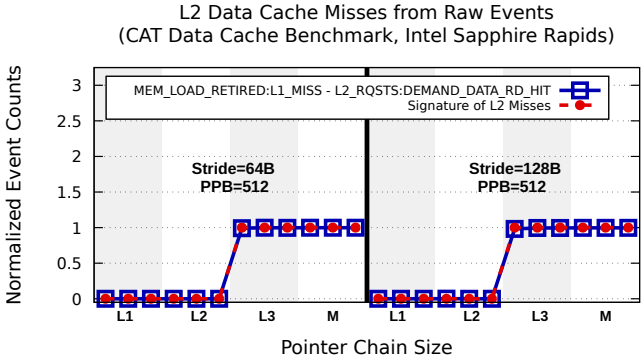
(b) L1 Misses.



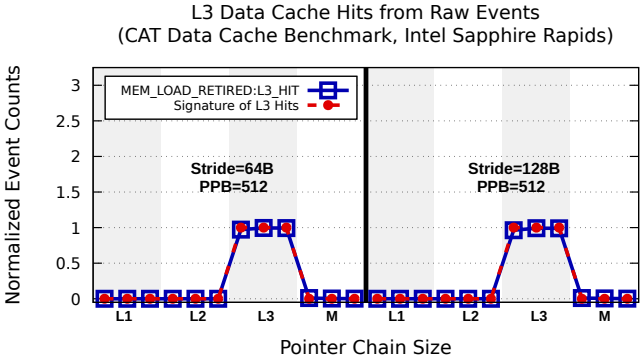
(c) L1 Reads.



(d) L2 Hits.



(e) L2 Misses.



(f) L3 Hits.

Fig. 3: Various data cache metric approximations from least squares.

have demonstrated the use of this analysis with both CPU and GPU floating-point units, branching units, and the memory subsystem, by coupling the analysis with the CAT benchmarks on the Intel Sapphire Rapids CPU and the AMD MI250X GPU architectures.

We are able to account for high levels of noise by collecting event measurements multiple times and excluding events that have high run-to-run variability across measurements. We quantify this noise using the maximum RNMSE between two measurements. For the data cache benchmark, we use multiple threads for our experiments, and minimize the noise by keeping the median reading across all threads.

These strategies sufficiently precondition measurement data prior to the QR factorization. Furthermore, our specialized QR factorization allows us to account for small noise, making it useful for data analysis of practical counter readings. We observed that branching and FLOPs (both CPU and GPU) events exhibit relatively low amounts of noise when executing the CAT benchmarks; however, the measurements of events corresponding to the memory subsystem are noisier.

TABLE VIII: Data Cache Metrics

Metric	Combination of Raw Events	Error
L1 Misses.	2.56e-3×MEM_LOAD_RETIRED:L3_HIT + 3.50e-4×L2_RQSTS:DEMAND_DATA_RD_HIT + 1.00001×MEM_LOAD_RETIRED:L1_MISS - 3.05e-4×MEM_LOAD_RETIRED:L1_HIT	4.07e-16
L1 Hits.	-5.69e-6×MEM_LOAD_RETIRED:L3_HIT - 4.21e-4×L2_RQSTS:DEMAND_DATA_RD_HIT - 4.19e-6×MEM_LOAD_RETIRED:L1_MISS + 0.9996×MEM_LOAD_RETIRED:L1_HIT	9.64e-17
L1 Reads.	2.55e-3×MEM_LOAD_RETIRED:L3_HIT - 7.14e-5×L2_RQSTS:DEMAND_DATA_RD_HIT + 1.00001×MEM_LOAD_RETIRED:L1_MISS + 0.9993×MEM_LOAD_RETIRED:L1_HIT	1.70e-16
L2 Hits.	-5.87e-3×MEM_LOAD_RETIRED:L3_HIT + 1.003×L2_RQSTS:DEMAND_DATA_RD_HIT - 2.39e-3×MEM_LOAD_RETIRED:L1_MISS - 3.11e-4×MEM_LOAD_RETIRED:L1_HIT	2.51e-16
L2 Misses.	8.43e-3×MEM_LOAD_RETIRED:L3_HIT - 1.002×L2_RQSTS:DEMAND_DATA_RD_HIT + 1.002×MEM_LOAD_RETIRED:L1_MISS + 5.74e-6×MEM_LOAD_RETIRED:L1_HIT	2.33e-16
L3 Hits.	1.02×MEM_LOAD_RETIRED:L3_HIT + 5.22e-3×L2_RQSTS:DEMAND_DATA_RD_HIT - 5.26e-3×MEM_LOAD_RETIRED:L1_MISS - 5.88e-6×MEM_LOAD_RETIRED:L1_HIT	2.54e-16

We implemented a specialized QR pivoting strategy that chooses the best sets of events representing our hardware expectation bases. It accomplishes this by prioritizing events which are closest to the individual dimensions of the expectation basis.

After producing the set of linearly independent events from the QR, performing least squares on the resulting matrix successfully provided linear combinations of raw events for the desired metrics, and it correctly indicated through the resulting error the cases where a metric cannot be composed from raw events on a given architecture. This analysis provides a numerical notion of fitness, which aids in validating event combinations. Our analysis defined architecturally available branching, floating-point, and cache metrics.

Even in the case of the cache metrics, where noise is the most prevalent, we demonstrated that rounding the resulting coefficients by just a few percent results in simple combinations of raw events that behave in ways that perfectly match the desired signatures of the performance metrics.

Future work will entail methods to develop different measures to quantify event noise and more rigorously select noise suppression thresholds and pivoting criteria.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their improvement suggestions. This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration; and by the National Science Foundation under award No. 2311707 "SPADE." This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This research also used a pre-production supercomputer with early versions of the Aurora software development kit at the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357.

REFERENCES

[1] S. Atchley *et al.*, "Frontier: Exploring Exascale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3607089

[2] V. G. Vergara Larrea *et al.*, "Scaling the Summit: Deploying the World's Fastest Supercomputer," in *High Performance Computing: ISC High Performance 2019 International Workshops, Frankfurt, Germany, June 16-20, 2019, Revised Selected Papers 34*. Springer, 2019, pp. 330–351.

[3] ALCF, "Aurora Fact Sheet," https://www.alcf.anl.gov/sites/default/files/2022-06/ALCF-Aurora_0.pdf, 2023.

[4] H. Jagode, A. Danalis, H. Anzt, and J. Dongarra, "PAPI Software-Defined Events for in-Depth Performance Analysis," *The International Journal of High Performance Computing Applications*, vol. 33, pp. 1113–1127, 2019.

[5] Advanced Micro Devices, Inc., "MI200 Performance Counters and Metrics," https://rocm.docs.amd.com/en/docs-5.7.1/understand/gpu_arch/mi200_performance_counters.html.

[6] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting Performance Data with PAPI-C," *Tools for High Performance Computing 2009*, pp. 157–173, 2009.

[7] S. S. Shende and A. D. Malony, "The Tau Parallel Performance System," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, May 2006.

[8] M. Schlütter, P. Philippen, L. Morin, M. Geimer, and B. Mohr, "Profiling Hybrid HMPP Applications with Score-P on Heterogeneous Hardware," in *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, ser. Advances in Parallel Computing, vol. 25. IOS Press, 2014, pp. 773 – 782.

[9] H. Brunst and A. Knöpfer, "Vampir," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer US, 2011, pp. 2125–2129.

[10] D. Boehme *et al.*, "Caliper: Performance Introspection for HPC Software Stacks," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 550–560.

[11] D. Barry, A. Danalis, and H. Jagode, "Effortless Monitoring of Arithmetic Intensity with PAPI's Counter Analysis Toolkit," in *Tools for High Performance Computing 2018 / 2019*. Cham: Springer International Publishing, 2021, pp. 195–218.

[12] ALCF, "Getting Started on Aurora," https://docs.alcf.anl.gov/aurora/hardware-overview/machine-overview/, 2023.

[13] OLCF, "Frontier User Guide," https://docs.olcf.ornl.gov/systems/frontier_user_guide.html, 2023.

[14] A. Danalis, H. Jagode, Hanumantharayappa, S. Ragate, and J. Dongarra, "Counter Inspection Toolkit: Making Sense Out of Hardware Performance Events," in *Tools for High Performance Computing 2017*. Cham: Springer International Publishing, 2019, pp. 17–37.

[15] M. Ritter, A. Tarraf, A. Geiß, N. Daoud, B. Mohr, and F. Wolf, "Conquering Noise with Hardware Counters on HPC Systems," in *2022 IEEE/ACM Workshop on Programming and Performance Visualization Tools (ProTools)*, 2022, pp. 1–10.

[16] W. Sid-Lakhdar *et al.*, "PAQR: Pivoting Avoiding QR Factorization," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2023, pp. 322–332.