Collecting hardware event counts is essential to understand program execution behavior. Contemporary systems offer few Performance Monitoring Counters (PMCs), thus only a small fraction of hardware events can be monitored simultaneously. We present new techniques to acquire counts for all available hardware events with high accuracy, by multiplexing PMCs across multiple executions of the same program, then carefully reconciling and merging the multiple profiles into a single, coherent profile. We present a new metric for assessing the similarity of statistical distributions of event counts and show that our execution profiling approach performs significantly better than Hardware Event Multiplexing.

Additional Key Words and Phrases: Hardware event monitoring, performance monitoring counters, hardware event multiplexing, task-parallel performance analysis

1 INTRODUCTION

Hardware performance monitoring during software execution is a key technique for performance analysis, as it provides insight into complex execution time hardware behavior and potential performance bottlenecks. Architectural support for hardware performance monitoring exists in the form of Performance Monitoring Counters (PMCs), a small set of architecture-specific, on-chip registers that can be programmed to count instances of a specified hardware event, selected from a set of hardware events supported on the machine.

Analysis of hardware behavior using PMCs is however significantly limited as the number of available events is generally orders of magnitude higher than the number of PMCs, identified as a key issue facing hardware performance monitoring [12]. Hence, only a small subset of possible events can be monitored simultaneously during program execution. Furthermore, specific hardware events may be incompatible with one or more other hardware events, which means that they cannot be simultaneously monitored even if the number of desired hardware events to be monitored is less than the total number of PMCs on the system.

A common technique to profile more hardware events than there are available PMCs is known as Hardware Event Multiplexing (HEM). This sampling technique over-commits the PMCs with requested hardware events to monitor, causing them to be allocated across the performance counting registers in a time-sharing scheme.

This has several disadvantages. Firstly, application behavior might not be captured accurately: while the number of instances of an event during monitored phases is captured accurately, the count for all unmonitored phases can only be interpolated based on statistical assumptions. If the assumptions are incorrect, the reported number of instances of an event can differ substantially from the actual count. For example, linear interpolation fails to capture spikes in hardware event activity during unmonitored phases and might thus lead to incorrect event counts.

Secondly, the multiplexing period might not be short enough to obtain samples for all events during very short execution intervals, such as fine-grained tasks found in many task-parallel programs. This issue is aggravated with an increasing number of events to be monitored, since the total time required to get samples for all events increases with every additional event.

New paper, not an extension of a conference paper.
Finally, the multiplexing algorithm itself also perturbs the system on each multiplexing period, potentially causing run-time overhead that can significantly impact program performance when over-committing the PMCs. For shorter multiplexing periods this overhead increases, since the instructions that re-allocate the set of events to be monitored to the PMCs are executed more frequently.

In this paper, we present a new approach for obtaining a consistent view of program behavior for a large number of hardware events, without the disadvantages of HEM. The approach can be used for programs with arbitrarily short monitored code regions, such as task-parallel programs. This is achieved by monitoring program behavior multiple times for different sets of hardware events and by combining the resulting execution profiles into a single, consistent profile using statistical methods. Our approach is robust against variations of program behavior across executions, and by operating solely on post-mortem execution profiles, it induces very low run-time profiling overhead compared to that caused by additional sampling code required by on-line techniques such as HEM. We introduce two implementations for execution profile combination: Label Graph Location (LGL) and Behavior Clustering (BC). We demonstrate our approach on programs of the OpenStream task-parallel language [18], which generates hardware event profiles accessible via the \textit{libaftermath-trace} portable trace analysis library [3].

By successfully enabling the generation of execution profiles with reliable monitoring data for larger, more complete sets of hardware events, this work aims to open wider research opportunities in performance and hardware analysis. For example, knowledge of the behavior of each hardware component during each individual region of execution can enable insight into how the components relate to each other at run-time, potentially leading to more comprehensive and effective system performance and power modeling.

This paper makes the following contributions:

1. A new approach for execution profiling that allows users to acquire reliable performance data for all available hardware events at arbitrary granularity (e.g., tasks of arbitrary duration) with two profile combination techniques LGL and BC.

2. A new metric called \textit{Execution Profile Dissimilarity} for assessing the accuracy of task-level hardware event counts recorded in dynamic task-parallel execution profiles using the Earth Mover’s Distance [20]. In this paper, we use the term \textit{accuracy} to be the similarity of statistical distributions of event counts to those observed in reference executions, further defined in Section 4.

3. A quantitative analysis of the performance monitoring data produced by \textit{perf_event} HEM on dynamic task-parallel programs and hence the limitations of this approach.

4. An evaluation of execution profiles generated by our combination techniques against those produced by HEM.

This paper is organized as follows. Section 2 details formal definitions for task-parallel event monitoring on which we base our combination and evaluation techniques. Section 3 introduces the hardware event profiling technique based on task-level combination across multiple profiling executions. Section 4 presents our evaluation method EPD that assesses similarity of monitoring data in hardware event profiles to reference executions, and discusses the main existing evaluation metric: \textit{Correlation Error} [14]. In Section 5, We evaluate execution profiles produced by HEM with different sampling configurations, as well as those produced by our approach with the LGL and BC combination techniques. Related work is then discussed in Section 6 before concluding in Section 7.
We assume that each core has a fixed number of PMCs with identical monitoring capabilities. For simplicity, we do not take into account incompatibilities between events and assume that the events of any subset $E \subseteq \mathcal{E}$ can be monitored simultaneously by any core if $|E| \leq N_{\text{PMC}}$. For the

2 FORMAL DEFINITIONS FOR EVENT MONITORING IN TASK-PARALLEL LANGUAGES

Task-parallel languages are increasingly popular for programming massively parallel, general-purpose architectures [1, 4, 15, 17–19]. Such languages allow programmers to expose large amounts of parallelism in the form of tasks to a run-time system that manages their execution.

Characterizing the behavior of individual tasks is essential for accurate performance analysis. For hardware performance monitoring, it is therefore necessary to obtain per-task event counts to quantify hardware event occurrences for each task. For example, the programmer may be interested in the number of dynamic instructions of a task, in the number of cache misses, or in a derived metric that combines multiple per-task event counts, such as a task’s average cache miss rate. A common technique is to monitor hardware events throughout the execution of the task-parallel program, saving event counts at the beginning and end of each task, then subtracting the two.

Our combination techniques rely on the following formal definitions.

**Tasks and Task Creation Graphs.** A task-parallel program is composed of a statically defined set of task types (i.e., the work functions) that can be instantiated an arbitrary number of times. Let $T_x$ be the set of task instances of an execution $x$. For brevity, task instances are referred to as tasks. We define the task creation graph $G_x = (T_x, D_x)$ of an execution $x$ as a directed tree, where vertices are tasks and the set $D_x \subseteq T_x \times T_x$ contains an edge $(t, t')$ iff task $t$ creates $t'$. We say that $t$ is the parent of $t'$ and $t'$ is a child of $t$. Aside from the root task, each task $t$ has a parent task denoted by $\text{parent}(t)$. We consider programs to be deterministic (i.e., same output for a given input) and that all executions of a given program use the same input. We also assume that task creation is deterministic in the sense that any two executions produce isomorphic task creation graphs.

Figure 1a shows a task-parallel program composed of four task types $\tau_r, \tau_a, \tau_b$, and $\tau_c$, which are instantiated a total of six times. The task graph of a first execution $x$ is given in Figure 1b with the set of tasks $T_x = \{ t^1_x, t_x^2, t^3_x, t_x^4, t_x^5, t_x^6 \}$. The symbols $t^1_x$ to $t^6_x$ have been chosen arbitrarily with the sole purpose of distinguishing different task instances. The task types indicated next to the tasks are not part of the graph and have only been added for clarity. Figure 1c shows the task creation graph of a second execution $y$ with the set of tasks $T_y = \{ t^1_y, t^2_y, t^3_y, t_y^4, t_y^5, t_y^6 \}$. Although the sets of tasks $T_x$ and $T_y$ are different, the parent-child relationships lead to two isomorphic task graphs.

**Events.** Let $\mathcal{E}$ be the set of hardware events available on the system. Each core of the system can monitor a certain number of events simultaneously by assigning events to its core-private PMCs. We assume that each core has a fixed number $N_{\text{PMC}}$ of PMCs with identical monitoring capabilities. For simplicity, we do not take into account incompatibilities between events and assume that the events of any subset $E \subseteq \mathcal{E}$ can be monitored simultaneously by any core if $|E| \leq N_{\text{PMC}}$. For the

---

Fig. 1. Task creation graphs for two executions of a program, where $t_r$ is the root task instantiated by $\text{main}$.
Fig. 2. Combination of two profiles $P_\alpha$ and $P_\beta$ into a global profile $P_\gamma$ containing counts for all events in $E = E_\alpha \cup E_\beta$. An example of task profile combination of two related task profiles is highlighted.

rest of this paper, we assume that the user wishes to obtain counts for a larger set of events than can be monitored at once. We denote the set of events monitored during an execution $x$ as $E_x$.

**Task Profiles and Execution Profiles.** For each task, hardware event monitoring generates an *event count* for each of the monitored events. Let the function $\text{count}(t,e)$ be the function that returns the event count of a task $t$ for an event $e$. We refer to the set of pairs of the form $(e, \text{count}(t,e))$, with one pair for each event $e \in E_x$, as the *task profile* of a task $t$ and use the function $\text{task profile}(t)$ as a shorthand for this set. The set $P_x$ composed of the task profiles of all tasks of an execution $x$ is referred to as the *execution profile*. The goal of the techniques presented in the next sections is to obtain a global execution profile that associates each task with event value pairs for all events $E$, just as if the events had been monitored simultaneously.

**Combinations.** In order to assemble a global profile, the techniques presented in this paper obtain profiles of separate executions, each monitoring only a subset of all available hardware events, and combine these profiles into a global profile containing counts for all events. We refer to a function combining a set of execution profiles into another execution profile as a *combination*. To define the task profiles of the global execution profile, the combination function chooses two or more task profiles from the input execution profiles and combines them by invoking a *task profile combination function*. The combination function repeats the task profile combination until the entire global profile has been assembled. Figure 2 illustrates the terms for two example executions $\alpha$ and $\beta$, each with six tasks and the respective event sets $E_\alpha = \{e^1_\alpha, e^2_\alpha, \ldots\}$ and $E_\beta = \{e^1_\beta, e^2_\beta, \ldots\}$. The highlighted rows in $P_\alpha$ and $P_\beta$ represent an example of two input task execution profiles of the tasks $t^4_\alpha$ and $t^6_\beta$ passed to the task profile combination function to generate the task profile for $t^4_\gamma$ with event counts for all events in $E_\alpha \cup E_\beta$.

The mechanism by which task profiles are selected for combination is crucial for the success of a combination since they define whether the combination produces a meaningful global profile. If the hardware behavior during each selected task was inconsistent, then the resulting combined task profile will have inconsistent event counts, and so incorporating it in any subsequent hardware event analysis will likely decrease the reliability of results. We describe an approach to evaluate this in Section 4. In the next section, we present two strategies for implementing the combination: one which selects tasks according to their position in the task creation graph of the program, and another that takes into account variability during execution by selecting tasks based on similar behavior and interaction with the hardware.
3 EXECUTION PROFILE COMBINATION

Combining tasks across distinct executions of task-parallel programs is non-trivial. The execution is highly dynamic and influenced by a multitude of factors related to the software and hardware environment. Task-parallel programs generally avoid over-constraining the execution order to maximize parallelism, and rely on dynamic scheduling for load-balancing. This leads to different execution orders and different assignments of tasks to cores across executions, which can in turn lead to different utilization of hardware resources, such as memory, caches and interconnects. For example, Figure 3 shows three tasks $t$, $t'$ and $t''$, where task $t$ produces data for the two consumer tasks $t'$ and $t''$. To avoid data races, the program must ensure that $t$ is executed before $t'$ and $t''$, but may not need to order $t'$ and $t''$. Furthermore, the hardware also contributes to variations, e.g., through dynamic frequency scaling or the correction of bit-errors in caches, both leading to different timings for identical sequences of instructions. A naive combination strategy — that assumes that executions of the same program with identical inputs execute the same tasks, on the same cores, and in the same order — may therefore combine tasks with very different execution-time microarchitectural behavior. More sophisticated combination mechanisms are thus necessary.

In this section, we present two combination strategies: Label Graph Location Combination, which combines tasks according to their position in a graph derived from the task creation graph, and Behavior Clustering, which combines tasks based on similar behavior across executions.

3.1 Label Graph Location Combination

The Label Graph Location (LGL) strategy combines tasks with the same position in their respective task creation graphs, which are consistent across dynamic executions for deterministic programs. We identify tasks across executions with a unique label, derived from its path from the graph’s root task and from its rank of creation. The resulting label graph, is constructed as follows.

**Task labels.** Let $\mathcal{L}$ be the set of all labels that can be constructed with the following grammar:

\[
\begin{align*}
\langle "label" \rangle & ::= \langle "num" \rangle \mid \langle "num" \rangle \cdot \langle "label" \rangle \\
\langle "num" \rangle & ::= \langle "digit" \rangle \mid \langle "digit" \rangle \langle "num" \rangle \\
\langle "digit" \rangle & ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid \ldots \mid 9
\end{align*}
\]

Let $label(t)$ be a function that assigns a label to a task $t$ according to the following rules:

\[
label(t) = \begin{cases} 
\emptyset & \text{if } t \text{ is the root task} \\
label(parent(t)) \oplus \text{childrank}(t) & \text{otherwise}
\end{cases}
\]

where $\oplus$ is the concatenation between labels with a dot and $\text{childrank}(t)$ returns the creation rank of a given task by its parent.
Figure 4 illustrates the labeling for the execution $x$ of the example program defined in Figure 1a. The root task $t_x^1$ is a special case and receives the label $\emptyset$ according to the first rule. Since $t_x^2$ is the first task to be created by the root task, $\text{childrank}(t_x^2) = \emptyset$, so $\text{label}(t_x^2)$ is the concatenation of the label of the root task and $\emptyset$, yielding $\emptyset . \emptyset$. Next, as $\text{childrank}(t_x^3) = 1$, the label of $t_x^3$ is $\text{label}(t_x^3) = \emptyset . 1$.

**Label graphs.** Based on the label function, we define the label graph of an execution $x$ as the graph that can be obtained by replacing each node $t \in T_x$ of the task creation graph with label($t$) and each edge $(t, t')$ of the task creation graph with (label(parent($t$)), label($t$)). Since task labeling associates a task with a label that only depends on deterministic task creation, the label graphs of two executions of the same program with the same input are identical.

Algorithm 1 defines the LGL combination mechanism on an unordered set of execution profiles $\{P_1, P_2, \ldots, P_n\}$, given integer subscripts in this section for clarity of description. For each unique label, the LGL combination strategy selects from each profile the single task corresponding to that label. These tasks are then combined into a single task via the function $\text{combine_tasks}_{\text{LGL}}$ for insertion into the resulting combined profile $P_{\text{LGL}}$. The combination function $\text{combine_tasks}_{\text{LGL}}$ is a partially defined function that combines the selected set of tasks to produce a derived task. We assume LGL applies to the minimum number of necessary combinations, where the monitored event sets across the executions are mutually non-overlapping, i.e., the following predicate holds:

$$\forall i, j \in [1, n], i \neq j : E_i \cap E_j = \emptyset$$

The function $\text{combine_tasks}_{\text{LGL}}$ is given in Algorithm 2. The algorithm first calls the function $\text{gentask}$ to generate a new, derived task instance, which is subsequently given the same label as the input tasks. The task profile for each input task is then retrieved and added to the newly generated task. The algorithm results in the returned task $t_{\text{combined}}$ that is associated with monitored event counts for all events in the unioned set $\bigcup_{i=1}^{n} E_i = \mathcal{E}$, recorded during executions of tasks each with the same location in their respective label graphs.

The tasks that are combined by LGL have identical static behavior: they are instances of the same task type, process identical input data and generate identical output data. However, their dynamic behavior can have substantial variation (e.g., dynamic mapping to cores, re-ordering, shared caches), so the resulting combined task profiles may contain inconsistent event counts. We address these issues with a different combination technique that accounts for similarity in micro-architectural behavior, presented in the next section.
3.2 Behavior Clustering

To account for variable dynamic behavior and execution schedule of tasks, we present the Behavior Clustering (BC) combination scheme. We first provide an overview of the combination scheme before detailing our full algorithm.

As with LGL, the set of events to be monitored are partitioned into \( n \) independent event sets to be executed in \( n \) distinct executions. Contrary to LGL, BC operates on a partially ordered sequence of profiles, according to their monitored event sets. The partial order is necessary because BC relies on the events that are common between two profiles to identify tasks with similar behavior, so the intersection must never be empty. In the following, we consistently index corresponding profiles and event sets, meaning that \( E_1 \) corresponds to the event set monitored to produce the first profile \( P_1 \), and so on. In a valid ordering, each event set must have a non-empty intersection with the set of events formed by all of its predecessors, as specified by the following condition:

\[
\forall i \in [2, \ldots, n] : \left( E_i \cap \bigcup_{j=1}^{i-1} E_j \right) = E_{\text{com},i} \neq \emptyset
\]

Let \( \{P_1, P_2, \ldots, P_n\} \) be a valid order for the set of profiles to be combined. BC operates on a series of \( n-1 \) combination steps, where at each step \( i \in [2, \ldots, n] \), the resulting profile of the previous combination step \( P_{\text{combined},i-1} \) is combined with the profile \( P_i \), to produce the next combined profile \( P_{\text{combined},i} \). This resulting profile of the combination step then consists of task profiles with event counts for the event set \( E_{\text{combined},i} = E_{\text{combined},i-1} \cup E_i \). BC is therefore a recursive, accretive process over a series of pairs of profile combinations, starting with \( P_{\text{combined},1} = P_1 \). The final result \( P_{\text{combined},n} \) contains task profiles associated with event counts for all events in \( \bigcup_{i=1}^{n} E_i = E \).

At each combination step, BC combines tasks between the two profiles that have similar counts for their common events \( E_{\text{com},i} \), which we refer to as the overlapping event set of the combination step. To do this, all tasks from both profiles are clustered based on their distribution with respect to these overlapping events. Within each cluster, task pairs are created by mapping each task from one profile to a task from the other profile. Each task pair is then processed to produce a set of combined tasks that are each associated with counts for the events from both profiles, with the counts recorded during tasks that behaved similarly during their respective run-times. This process defines BC conceptually, where the mechanism by which similar tasks are clustered is left to the implementation. In this paper, we present a straightforward clustering approach to demonstrate the BC combination scheme, and leave more sophisticated cluster analysis techniques to future work.

Our BC implementation identifies tasks with similar counts through an iterative clustering scheme: in each iteration, pairs of tasks are selected across the two profiles that satisfy a particular constraint on their event count similarities, with each subsequent iteration requiring a lesser constraint. Thus BC preferentially combines tasks with the most similar event counts between profiles. The process is carried out independently per task type, and in the following descriptions we refer to the set of tasks from the previous step’s combined profile and the set of tasks from the profile to be incorporated in the current step as \( T_{\text{prev}} \) and \( T_{\text{cur}} \) respectively.

The process can be conceptualized into three components: task clustering, cluster processing, and the iterative relaxation process that carries out the task clustering and cluster processing. These are described next in this section and illustrated in Figure 5. The figure shows scatter-plots of tasks, with the shape of the tasks indicating membership of the different profiles: squares represent the task profiles of \( T_{\text{prev}} \), and circles represent task profiles of \( T_{\text{cur}} \). The type of tasks is indicated by the color, where there are two task types in the example: \( \tau_a \) and \( \tau_b \). The axes of the figure each represent an event of the overlapping set. For illustrative purposes only two overlapping events are used, resulting in two axes in the figure.
Task Clustering. At each iteration, a regular-shaped grid is defined by dividing the bounds of the hardware event value space across the unioned task set $T_{prev} \cup T_{cur}$ by a grid division parameter $d$ that for the first iteration is initially set to a maximum value $d_{max}$. The specification of this parameter is detailed further below in this section. The resulting grid is illustrated in Figures 5b to 5f as the dotted lines. We refer to each cell of this grid as a cluster. In our implementation of BC, each cluster is solely defined according to the dimensions of the static grid at each step, meaning that our use of the term cluster is distinct from the more general clustering mechanisms of fields such as unsupervised data mining. The grid division parameter $d$ controls the sizes of the clusters: higher values lead to greater constraints on the similarity of tasks within each cluster with respect to the overlapping events.

Cluster Processing. To create a combined task with counts from both profiles, it is necessary to associate a pair of tasks, where each task in the pair is sourced from a different profile. For each cluster, the tasks within it are defined by the clustering process as being similar with respect to the overlapping hardware events. Therefore any selected pair of tasks within the cluster, where one of the tasks is from $T_{prev}$ and the other from $T_{cur}$, is a conceptually valid selection according to BC’s goal of combining the tasks which behaved similarly at run-time. We propose two strategies to select the task pairs. In both, the tasks within the cluster are separated by their source execution profile into two lists, and pair selections made according to corresponding list positions. For the first strategy, which we simply refer to as BC, each list is sorted according to the tasks’ positions in the profile’s label graph. The ordering is that of their order of visitation in a depth-first search of the label graph, with siblings ordered by their label rank with respect to their parent, equivalent to their order of creation in the program. The label ordering therefore influences the order of combination to ensure that BC favors combining tasks within each cluster that have the same labels or the same parent labels. We refer to the second selection strategy as BC (Unlabeled), in which the two lists are randomly shuffled via the C++ standard library `std::random_shuffle` algorithm, instead of ordered according to labels. In Section 5 we present a comparison of these two BC selection strategies, in order to investigate the necessity of assigning label graph locations to tasks for the purposes of BC combination.

Each selected pair of tasks is then combined to produce a new, derived task that is associated with all the event counts from the task of the previously combined profile, and the counts for the non-overlapping, ‘new’ events being added to it ($E_{cur} \setminus E_{prev}$) from the task of the incoming profile.
in the current combination step. The counts for the overlapping events are therefore those that were first included in the ordered set of combinations and are not averaged. This ensures that all event counts in a combined task profile are true values recorded during a single monitored execution. The resulting task’s label is determined as the first common ancestor in their respective label graphs. This means that tasks of combined profiles produced via BC combination do not necessarily have unique labels, as they can be generated from tasks with different unique labels in their original profiles. After combination, the selected tasks are removed from their original respective task sets $T_{prev}$ and $T_{cur}$, as can be seen in Figure 5c. Once all cross-profile pairs of tasks have been combined, such that at least one of the lists in each cluster is exhausted, the process continues to the next iteration as follows.

**Iterative Similarity Constraint Relaxation.** After processing the clusters in an iteration, the remaining tasks are those with which no pair could be formed due to the two task sets within their clusters not being of equal size, e.g., as in Figure 5c). To include these tasks into the combined profile, after each iteration the intra-cluster task similarity constraint is relaxed. This is done by reducing the grid division parameter and then redefining the grid, as shown in Figure 5d in which the grid division was reduced by 1. A reduced grid division parameter results in larger cluster sizes, meaning that the populated tasks within each cluster may have less similar event counts than those that populated clusters in previous iterations.

The first iteration occurs with the grid division parameter set to $d_{max}$, which is analytically determined from the tasks’ event counts. This is calculated by determining the smallest difference in count between any pair of combinable tasks across the profiles, with respect to each individual overlapping hardware event. The parameter $d_{max}$ is then set to the minimum corresponding grid division calculated from each of the minimum distances across the overlapping events. $d_{max}$ therefore represents an upper-bound for the grid division necessary for the most similar tasks to be combined, since two tasks may be the closest with respect to a single event but they may not be close with respect to another overlapping event.

A similar process is used at the end of each iteration in order to determine the next reduced grid division. However, as the remaining tasks at the end of an iteration are already allocated to clusters as part of the task combination process, the closest sets of tasks are readily available by finding the closest clusters. The calculation for the next grid division update is therefore made more precise by finding the nearest individual combinable pair of tasks within these closest sets of tasks, with their proximity calculated with respect to all overlapping events.

To do this, the closest pairs of clusters are found using the Euclidean distance between their cluster coordinates, where each coordinate component corresponds to a dimension of the grid and is given as the integer index of the cluster along that dimension. Next, the closest cross-profile pairs of tasks are found between each of these pairs of clusters. The distance between two tasks is similarly given as the Euclidean distance, but calculated from their specific event count differences for each hardware event dimension, with each count difference represented as a fractional number of grid divisions. This normalization of event counts to grid divisions is done in order to eliminate bias when calculating the aggregate Euclidean distances across the dimensions that correspond to events with different units (e.g., CPU cycles versus cache misses). The integer grid division parameter is then updated according to the minimum of these differences, to make the size of the clusters produced by the updated grid division equivalent to the current cluster size plus the desired cluster size. In the optimistic case, this relaxes the similarity constraint sufficient to ensure the next iteration’s cluster sizes are large enough to accommodate the closest remaining tasks from the current iteration. This is not guaranteed, however, as the boundaries of each cluster are defined relative to the minimum and maximum value for each event, thus the two closest tasks may
Algorithm 3: combination_{BC}(T, P_1, P_2, \ldots, P_n):

1. \( P_{\text{combined}} = P_1; \quad E_{\text{combined}} = E_1 \)
2. for \( i \in [2, n] \) do
3. \( P_{\text{prev}} = P_{\text{combined}}; \quad E_{\text{prev}} = E_{\text{combined}} \)
4. \( P_{\text{cur}} = P_i; \quad E_{\text{cur}} = E_i \)
5. \( P_{\text{combined}} = \emptyset; \quad E_{\text{combined}} = E_{\text{prev}} \cup E_{\text{cur}} \)
6. \( E_{\text{com},i} = E_{\text{prev}} \cap E_{\text{cur}} \)
7. foreach type \( \tau \in T \) do
8. \( T_{\text{prev}} = \text{get_tasks}(P_{\text{prev}}, \tau); \quad T_{\text{cur}} = \text{get_tasks}(P_{\text{cur}}, \tau) \)
9. mins = \( \emptyset; \quad \text{maxes} = \emptyset \)
10. foreach event \( e \in E_{\text{com},i} \) do
11. \( \text{min}[e] = \min(\text{val}(e, T_{\text{prev}} \cup T_{\text{cur}})) \)
12. \( \text{max}[e] = \max(\text{val}(e, T_{\text{prev}} \cup T_{\text{cur}})) \)
13. end
14. grid_division \( d = \text{calc}_{\text{max div}}(T_{\text{prev}}, T_{\text{cur}}, E_{\text{com},i}) \)
15. while \( d \geq 1 \) do
16. foreach event \( e \in E_{\text{com},i} \) do
17. \( \text{cluster sizes}[e] = \frac{\text{min}[e] - \text{max}[e]}{d} \)
18. end
19. foreach cluster \( c \in \{0, \ldots, d-1\} \) do
20. \( T_{\text{prev}} = \text{sort}_{\text{or shuffle}}(\text{tasks in cluster}(c, T_{\text{prev}}, \text{cluster sizes})) \)
21. \( T_{\text{cur}} = \text{sort}_{\text{or shuffle}}(\text{tasks in cluster}(c, T_{\text{cur}}, \text{cluster sizes})) \)
22. \( N_{\text{prev}} = \min(|T_{\text{prev}}|, |T_{\text{cur}}|) \)
23. for \( j = 0 \) to \( N_{\text{prev}} \) do
24. \( t_{\text{prev}} = T_{\text{prev}}[j] \)
25. \( t_{\text{cur}} = T_{\text{cur}}[j] \)
26. \( t_{\text{combined}} = \text{combine}\text{\_tasks}_{\text{BC}}(t_{\text{prev}}, t_{\text{cur}}, E_{\text{prev}}, E_{\text{cur}}) \)
27. \( P_{\text{combined}}.\text{add}(t_{\text{combined}}) \)
28. \( T_{\text{prev}}.\text{remove}(t_{\text{prev}}) \)
29. \( T_{\text{cur}}.\text{remove}(t_{\text{cur}}) \)
30. end
31. \( d = \text{calc}_{\text{reduce div}}(T_{\text{prev}}, T_{\text{cur}}, E_{\text{com},i}, d) \)
32. end
33. end
34. end
35. return \( P_{\text{combined}} \)

Algorithm 4: combine\_tasks_{BC}(t_{\text{prev}}, t_{\text{cur}}, E_{\text{prev}}, E_{\text{cur}}):

1. \( t_{\text{combined}} = \text{gentask()} \)
2. \( t_{\text{combined}}.\text{add}(\text{task}\_\text{profile}(t_{\text{prev}}, E_{\text{prev}})) \)
3. \( t_{\text{combined}}.\text{add}(\text{task}\_\text{profile}(t_{\text{cur}}, E_{\text{cur}} \setminus E_{\text{prev}})) \)
4. \( t_{\text{combined}}.\text{set}\_\text{label}(\text{label}(t_{\text{prev}}) \cup \text{label}(t_{\text{cur}})) \)
5. return \( t_{\text{combined}} \)

continue to be separated by a cluster boundary in the new grid definition. The relaxation process is therefore an approximation to achieve iteratively looser bounds on similarity without attempting every possible grid division lower than the upper bound \( d_{\text{max}} \).

Once the grid division parameter has been reduced the next iteration is processed, leading to the remaining tasks shown in Figure 5e. The iterative process continues until there are no remaining tasks that require combination, which occurs at the latest at a grid division of 1, given that both input profiles have the same original number of tasks.

Behavior Clustering Algorithm. Algorithm 3 shows the full BC combination algorithm. Each recursive combination step spans Lines 2 to 35, with each step combining \( P_{\text{cur}} \) with \( P_{\text{prev}} \) to produce \( P_{\text{combined}} \). The process proceeds for each task type at Line 7, where first the bounds of each overlapping hardware event are calculated using the functions \( \text{minval} \) and \( \text{maxval} \). The initial value for \( d \) is calculated at Line 14, where \( d \) then defines the distance between two grid lines along the dimension.
for an event $e$, referred to as $\text{cluster} \_\text{size}[e]$ (Line 17). Each cluster is processed from Lines 19 to 31. Let $c$ be the cluster coordinates and $c[e]$ the coordinate component corresponding to the event $e$. The tasks that belong to a cluster are returned by the function $\text{tasks} \_\text{in} \_\text{cluster} (c, T, \text{cluster} \_\text{sizes})$, where event values are between $(\text{mins}[e] + (\text{cluster} \_\text{sizes} \cdot c[e]))$ and $(\text{mins}[e] + (\text{cluster} \_\text{sizes} \cdot (c[e] + 1)))$, for each event in the overlapping set. The tasks are subsequently either sorted according to their labels or randomly shuffled, via the function $\text{sort} \_\text{or} \_\text{shuffle}$, depending on the selection strategy.

Next, the algorithm selects pairs of tasks between the two task sets within the cluster, according to corresponding positions $j$ in their respective label orders. The two selected tasks are then combined to produce a new, derived task denoted $t\_\text{combined}$ via the function $\text{combine} \_\text{tasks}_{\text{BC}}$ at Line 26. This function associates the counts for $E\_\text{prev}$ from $t\_\text{prev}$ to $t\_\text{combined}$, then associates the counts for the remaining events $E\_\text{cur} \setminus E\_\text{prev}$ from $t\_\text{cur}$, then as previously described, sets the label of $t\_\text{combined}$ to their common ancestor in the label graph. The combined task is added to the combined profile, and its two input tasks are removed from their respective task sets. After all clusters have been processed, the updated grid division parameter is then calculated via the function $\text{calc} \_\text{reduced} \_\text{div}$ at Line 14 for the next iteration.

The function $\text{combine} \_\text{tasks}_{\text{BC}}$ is defined in Algorithm 4. A new task is first created via the function $\text{gentask}$. All event counts from the task $t\_\text{prev}$ of the previously combined profile are added, and only the counts for the `new` events contributed by $t\_\text{cur}$ are combined. The counts for the overlapping events are therefore those which were first included in the combination steps and are not averaged. This ensures that all event counts in a combined task were recorded during a single monitored execution. The resulting task’s label is determined as the first common ancestor in their respective label graphs.

The specification of the overlapping hardware events at each step $i$ used to identify task similarity is an important consideration. To select task profiles for combination, that have event counts relevant to each other, there should be a predictive relationship between the tasks’ values for $E\_\text{com}$ and their values for $(E_i \setminus E\_\text{com})$. As a consequence of BC’s requirement for overlapping event sets, there are more executions necessary for BC to monitor all events. Therefore compared to LGL, for which each individual event is monitored only during a single execution, the resulting combined profile for BC has fewer pairs of simultaneously monitored events. Whilst including more overlapping events in each profile may result in combined task profiles sourced from tasks that exhibited greater hardware behavior similarity between executions, doing so further reduces the number of simultaneously monitored event pairs. The process we used to define such overlapping event sets is discussed in the experimental evaluation at Section 5, where in future work we aim to investigate approaches to design optimal event sets to combine profiles on any given architecture and benchmark.

### 3.3 Combination Example

In this section, we provide a short experimental evaluation of the LGL and BC combination strategies, showcasing the benefits of accounting for execution variability with BC compared to the assumption of constant behavior per label as in LGL. In the experiment below, we first collect performance data in two distinct executions of the same benchmark, both monitoring the same hardware event: Intel’s Sandy Bridge L1D:REPLACEMENT, which corresponds to a cache line replacement occurring in the L1 data cache. For the clarity in this experimental example, we refer to the hardware event L1D:REPLACEMENT differently depending on which profile the count is sourced from. To do this, we name them according to the event’s dual definition in the PAPI project [13] `preset’ hardware event list, to be PAPI_L1_DCM (L1 data cache misses) for one profile, and PAPI_L2_DCA (L2 data cache accesses) for the other. Since the data for the ‘two’ events are collected in two different runs, only separate per-task counts for each event are available. In addition, across the profiles there might...
Fig. 6. Example combinations. Dashed lines represent the relationship observed for simultaneous monitoring.

not be identical counts for PAPI_L1_DCM and PAPI_L2_DCA due to execution variability. However, given a sufficiently large number of tasks, there are tasks with very similar micro-architectural behavior in both executions. This property is used in the experimental analysis, where the two profiles are combined via LGL and BC combination, and the resulting combined tasks examined with respect to their counts for PAPI_L1_DCM and PAPI_L2_DCA. A sound combination scheme should generate a profile in which the per-task counts differ only slightly. Any deviation from this ideal reveals the potential shortcomings and invalid assumptions of the strategy.

The experimental system used for evaluation has two Intel Xeon E5-2690 processors with a total of 32 hardware threads (Hyper-Threading enabled) running at a frequency of 2.90 GHz. The machine’s 396 GB of main memory are distributed over two NUMA nodes. For the remainder of this paper, we refer to this machine as Xeon-32. We used version PAPI 5.4.1 [13] to monitor the PAPI preset hardware events shown in Figure 6a, in which the identically defined PAPI events PAPI_L1_DCM and PAPI_L2_DCA were monitored alongside three overlapping events for use in the BC combination, each indicated with an asterisk. The executed benchmark is an OpenStream implementation of Cholesky matrix decomposition, configured to operate on square matrices of $2^{13} \times 2^{13}$ elements using a block size of $2^8 \times 2^8$ elements. We refer to this benchmark as cholesky_b8. The benchmark creates a total of 8080 tasks instantiated from 18 task types defined in the program.

Scatter plots of the 8080 resulting tasks in the combined profile are shown in Figure 6. Values for the Task Mover’s Distance are also provided. This metric will be defined in Section 4. As PAPI_L1_DCM and PAPI_L2_DCA refer to the same native hardware event on the machine, a single execution and thus the ideal combination of two executions would produce all points on a straight line with slope 1, as indicated with the dashed line. However, the LGL results in Figure 6b show that the behavior of tasks with the same label significantly varied across the two executions, leading to many inconsistent tasks that are associated with vastly more L1 data cache misses than L2 data cache accesses and vice versa. The BC results in Figure 6c show a much improved distribution with the relationship more closely approximating the ideal. Whilst there are some tasks with inconsistent event counts, the task distribution in Figure 6c better represents the distribution of the ideal combination than that of Figure 6b. By accounting for execution variability, BC therefore results in a more successful combination with respect to the two example events.

As evaluation of a combination with respect to all of the events in a combined profile is non-trivial due to the complex relationships between events, the visual approach used in this section can only serve as a motivating example. In the next section, we present our profile evaluation metric that allows for quantitative evaluation.
4 EVALUATING EXECUTION PROFILES

An ideal combination or sampling technique produces event counts for each task that are identical to the event counts of an ideal reference execution, in which events are monitored simultaneously during the entire execution. However, since only a subset of the all events can be monitored at the same time, the event counts of the reference execution remain unknown and therefore cannot be used to evaluate a combination. Furthermore, evaluating each task’s event counts with a corresponding task in a reference profile is impracticable due to variability between executions.

To overcome these issues, we present a statistical evaluation approach, Execution Profile Dissimilarity (EPD), that considers the distribution of tasks in an execution profile with respect to its associated set of hardware events. In this section, we first present EPD before describing a further statistical evaluation approach from the literature called Correlation Error [14]. The two approaches are then compared in the experimental evaluation of Section 5.

4.1 Execution Profile Dissimilarity

We consider a successful combination or sampling technique to result in a task distribution similar to the task distribution of a reference execution, throughout which the events are simultaneously monitored. In this paper, we use the term accuracy to refer to this similarity: a highly accurate execution profile closely resembles the reference task distribution, whereas an inaccurate profile is very dissimilar. EPD aims to provide a quantitative evaluation of a profile’s accuracy.

As not all events can be profiled during a single reference execution, EPD is an aggregate metric over a set of calculated similarities, each evaluating the profile with respect to a different subset of the associated hardware events and each with a corresponding reference execution. In the following descriptions we call the profile that is undergoing evaluation the target profile, to distinguish it from the reference profiles.

To calculate the dissimilarity between two task distributions, EPD employs our customization of the non-parametric statistical Earth Mover’s Distance (EMD) [20] metric. For clarity, we refer to our specification of EMD for application to similarity analysis between task profiles as the Task Mover’s Distance (TMD), which quantifies the dissimilarity between task distributions by calculating a statistical distance. By using TMD, EPD accounts for hardware behavior variability between executions and assesses not only the associated absolute counts for each event in the profile, but also evaluates the relationships between the counts of different events in the profile.

To fairly evaluate accuracy of the target global profile with respect to all hardware event relationships, a reference profile is defined for every pair-combination of the included hardware events. For each reference pair, the events are monitored in multiple repeated reference executions, and the TMD is calculated between a projection of the target profile with respect to the event-pair and each of the reference execution repetitions. This produces a set of TMD values, the median of which is then taken as the resulting TMD for the profile with respect to that particular pair of events. These repetitions are carried out in order to ascertain the expected variability in the reference task distributions, which is used to calibrate each TMD result against the event pair, a process described in further detail in Section 4.3. Finally, the target profile’s overall EPD result is given as the geometric mean of the TMDs calculated across all of the hardware event pair combinations.

In the next Section, a simplified overview of EMD as presented in [20] is given, before TMD is detailed in Section 4.3.

4.2 The Earth Mover’s Distance

EMD was introduced to computer science in [20] and has found much use in computer vision, information retrieval and image processing. In the form used in this paper, it is equivalent to the 1st
Mallows Distance or the 1st Wasserstein Distance in mathematics [8]. The following is a simplified description of EMD, as presented in [20].

EMD represents the minimum amount of ‘work’ required to transform the bins of one multi-dimensional histogram \( p \), such that they are equivalent to the bins of another histogram \( q \). Each bin is represented by two parameters: its weight \( \omega \) and its location \( \lambda \), where the specification of bin locations is application-dependent and serves to inform the EMD algorithm of the distance between any two bins. In this paper, we refer to a bin location as a coordinate with respect to the dimensions of the histogram, described further in Section 4.3. The transformation of weight from a bin \( i \in p \) to a bin \( j \in q \) is called a flow \( f_{ij} \), bounded according to the available weight in the source bin: \( f_{ij} \leq \omega_i \), as well as the weight requirement of the target bin: \( f_{ij} \leq \omega_j \). The work required to do this transformation is defined as \( W_{ij} = f_{ij} \cdot \text{dist}(\lambda_i, \lambda_j) \), where the distance function \( \text{dist} \) defines the inter-bin distance based on the locations of \( i \) and \( j \). EMD finds the total set of flows \( F \) that minimizes overall work. It is therefore a formulation of the transportation problem with the bins of \( p \) representing suppliers and the bins of \( q \) representing consumers.

4.3 The Task Mover’s Distance

Task Mover’s Distance (TMD) quantifies the dissimilarity between the task distribution of the target profile and the task distribution of a reference profile, with respect to a specific subset of hardware events, referred to in this section as \( E \). TMD is a specification of EMD that defines the weight and location of each histogram bin, together with the distance function between bin locations. It then solves for EMD accordingly to produce a dissimilarity value between the two distributions. TMD includes a calibration step to account for execution variability observed in repeated reference executions, in order to produce the final TMD value between the two distributions.

We now present TMD, with Figure 7 showing an example TMD calculation between target and reference tasks distributed with respect to two events.

Histogram Binning. The target and reference task distributions are each represented as histograms with dimensions corresponding to the hardware events in \( E \). The bins of the two histograms are defined equivalently, with bounds of each dimension in the histogram corresponding to the minimum and maximum value found for the associated event across all reference executions. Each dimension is then partitioned into a number of equal-sized intervals corresponding to the desired granularity of analysis; a higher number of intervals results in a greater histogram resolution, which allows for a more precise calculation of TMD. The target profile may have tasks with event values outside the reference range. The bin count for each dimension is therefore increased by two for two additional bins receiving tasks lower than the reference value range and tasks greater than the reference value range, respectively. Each histogram is then populated, with the tasks of the corresponding profile allocated to the bins based on their recorded hardware event values. Letting \( n_i \) be the resulting task count within bin \( i \), the bin’s weight in the EMD calculation is given as \( \omega_i = \frac{n_i}{|T|} \), where \(|T|\) corresponds to the total number of tasks in the profile. Each bin’s weight therefore represents the proportion of tasks held in the bin relative to the total task count, and the calculated dissimilarity is conceptually the amount of work necessary to rearrange the proportions to match those of the reference distribution.

A histogram representation of two example task distributions is given in Figure 7a, where both task distributions are shown in the same histogram for simplicity, with reference tasks illustrated by circles and the tasks of the target evaluation profile illustrated by crosses. In the example, one task in the target task distribution has event values outside the reference range and is therefore placed in one of the external bins, which are indicated by dashed lines on a gray background.
Bin Locations and the Distance Function. The location of a bin is defined as the Cartesian coordinate vector within the multi-dimensional histogram that corresponds to the centroid of the bin’s populated tasks. This centroid is calculated from the bin’s populated tasks’ mean count for each event and is thus calculated after all tasks have been allocated to bins. Each coordinate component for an event corresponds to the centroid’s fractional position along the dimension corresponding to that event. A bin location is therefore equivalent to a Cartesian coordinate in a continuous |E|-dimensional Euclidean space with unit-length equal to one bin. We define the distance function between bins as the Euclidean distance between their coordinates.

As each event’s value range is initially divided into an equal number of bins, distance is then a relative measure across the event space that is independent of the units of the set of hardware events. This is especially important to ensure that all events in an event set contribute equally even if their counts usually have different orders of magnitude. As the bin separation represents a relative change in event count, normalizing the calculation to bin-distance ensures that the calculation considers, for example, a 10% dissimilarity in L1 data cache misses the same as a 10% dissimilarity in TLB misses, even if the absolute count differences are not the same. This enables reasoning about distribution dissimilarities with respect to multiple events, where if both of these example 10% variations are observed then it is valid to calculate their resulting joint dissimilarity using Euclidean distance.

The bin locations are shown in Figure 7b, where the distance function dist applies between the coordinates of the bin in the target evaluation profile and the coordinates of the associated bin in the reference.

Calibration. To introduce the concept of TMD calibration, we refer to the result of solving EMD with the above specifications as the uncalibrated TMD. The basic units of uncalibrated TMD are abstract: the proportion of tasks is multiplied by the bin distance, which is relative to the chosen bin count. Conceptually, an uncalibrated result of 1.0 is equivalent to the entire task set moving a distance of one bin separation, or half the task set moving a distance of two bin separations, etc.

Due to the variability at execution time, the reference distribution itself varies and the uncalibrated TMD between repeated reference execution profiles is therefore non-zero. To conceptualize the quantification of dissimilarity and account for reference distribution execution variability, we normalize the uncalibrated TMD results to the median result calculated between every unique pair of profiles from the set of repeated monitored reference executions, with respect to the hardware.

![Figure 7](image-url)
event set \( E \). We term the median value of the uncalibrated TMD between all pairs of reference profiles as the \textit{Calibration TMD}, specific to the configured bin count. Therefore a TMD result of 1.0 approximates the median dissimilarity one would expect if the target profile was simply an execution during which the events were constantly and simultaneously monitored. Higher TMD values then represent task distributions with greater than mean expected dissimilarities. As EPD is an average across TMDs, the result of EPD is then also relative to a \textit{Calibration EPD} defined as 1.0.

Figure 7 includes the TMD results for the combined task distributions of the combination example presented in Section 3.3. The histograms were generated with a bin count of ten for each of the two hardware events, and the Calibration TMD was calculated using five repeats of the reference execution. The median TMD calculated against each of the reference task distributions for LGL is 4.00 ± 0.08, whereas BC’s TMD is 1.05 ± 0.41, with error given as the maximum variation from the median across the reference execution repetitions. These results suggest that TMD can appropriately capture the dissimilarity between task distributions, and therefore that EPD can evaluate the accuracy of profiles produced by a combination or sampling technique.

### 4.4 Correlation Error

Mytkowicz et al. [14] presented a statistical evaluation technique called \textit{Correlation Error}. This metric calculates Spearman’s correlation coefficient between each pair combination of included hardware events, comparing observed coefficients to those determined from reference executions, in which the event pairs are each simultaneously monitored throughout. An aggregate evaluation is then made across all hardware event pairs by plotting the difference between the observed and reference correlation coefficients as a cumulative distribution function (CDF), with the final evaluation result given as the area between the observed and ideal CDF. As Spearman’s correlation coefficient is within the interval \([-1.0; 1.0]\), the final result is bounded between 0.0 and 2.0.

We compare our EPD metric to Correlation Error in the Experimental Evaluation in Section 5.3.

### 5 EXPERIMENTAL EVALUATION

In this section, we first discuss and experimentally evaluate the accuracy and limitations of HEM for monitoring OpenStream task-parallel programs using EPD. We then compare HEM to the relative accuracy achieved by our combination approach on three task-parallel OpenStream benchmarks. We also experimentally compare the two evaluation metrics using Correlation Error.

#### 5.1 Hardware Event Multiplexing

HEM is a sampling method that time-shares hardware events onto over-committed PMCs. The Linux kernel has handled this scheduling in the \texttt{perf_event} subsystem since version 2.6.31. We briefly outline the multiplexing algorithm and refer to [2] for an in-depth discussion of the scheduling strategy.
The perf_event multiplexing algorithm schedules groups of compatible events to the PMCs. It organizes these groups in a circular, linked list and initializes a pointer to the first position from the list. At the beginning of each multiplexing period with a duration \( \pi \), the algorithm starts by selecting the events in the group referenced by the list pointer. It then tries to select the events of each subsequent group in the list, until an incompatibility is detected between the already assigned events and a newly selected event. The compatible groups are scheduled, and the algorithm advances the list pointer by a single element and waits for its next invocation. In our experimental setup using PAPI to interface with perf_event, the groups are configured to contain only a single event. Let \( E_{\text{HEM}} = \{ e_1, \ldots, e_n \} \) be the set of events that are to be monitored using HEM, leading to the creation of \( n \) groups each containing a single event, and let \( E_{\text{mon}}^i \) be the set of events monitored at the \( i \)-th multiplexing period. Figure 8 illustrates an example, where \( N_{\text{PMC}} = 3 \) and \( E_{\text{HEM}} = \{ e_1, \ldots, e_6 \} \) and where all events are compatible. We define the duration during which a given event is consecutively monitored as the sample duration \( \pi_{\text{sample}} \) and the minimal time that is necessary to obtain at least one sample for every event as \( \pi_{\text{all}} \).

The example represents the best case for accuracy, where an absence of monitoring incompatibilities between the events allows \( N_{\text{PMC}} \) events to be monitored simultaneously during each multiplexing period. In this case, \( \pi_{\text{sample}} = \pi \cdot N_{\text{PMC}} \) and \( \pi_{\text{all}} = \pi \cdot (|E_{\text{HEM}}| - N_{\text{PMC}} + 1) \). In the worst case, the events are all incompatible and every event has to be monitored separately, leading to a sample duration that only spans a single multiplexing period \( \pi_{\text{sample}} = \pi \), and the time to acquire a sample for all events \( \pi_{\text{all}} = \pi \cdot |E_{\text{HEM}}| \).

These factors are important for the accuracy of task profiles. For example, when a task is shorter than \( \pi_{\text{all}} \), then no samples are collected for certain events during a task, such as for \( \pi^2 \) in Figure 8. The accuracy of a task profile therefore increases with: (1) a greater number of PMCs; (2) a longer task duration; (3) fewer multiplexed events; or (4) a shorter multiplexing period. However, the number of PMCs is fixed by the hardware, the task duration is fixed by the task-parallel program, and the number of events to be monitored is fixed by the user. The multiplexing period is set by the operating system, but cannot be reduced arbitrarily since this increases multiplexing overhead due to frequent invocations of the event scheduler.

Previous works have aimed to evaluate the true accuracy of performance monitoring counters [7, 21, 22]. However, little work has been done to understand the accuracy and limitations of sampling techniques such as HEM across different configurations and profiling targets, pointed out as a key issue facing hardware performance monitoring by Moseley et al. [12]. Below, we present what we believe is the first quantitative study on the accuracy of event counts generated from executions monitored via HEM, specifically with respect to three impacting variables: \( |E_{\text{HEM}}| \), \( \pi \) and the average task duration. We examine their effect on accuracy at a constant number of PMCs, in order enable greater understanding of the limitations of the sampling approach.

**Experiment Overview.** To investigate the factors for HEM profile accuracy, we profiled executions of different average task durations, monitoring event sets varying in size from 2 to \( |E| \), multiplexing periods from 0.01 ms up to 4.00 ms, and evaluated the results using EPD, where we used FastEMD 3.1 [16] to solve EMD for TMD. We profiled the event sets on the Xeon-32 experimental system described in Section 3.3, running PAPI 5.4.1, which reports 33 available hardware events. In order to vary \( \pi \), we provided the Linux kernel version 4.9.0 with a boot parameter for the multiplexing period that can be set to values lower than the kernel’s default period of 1 ms. In the experiment, two instances of the OpenStream Cholesky matrix decomposition benchmark were profiled: cholesky_b8, as described in Section 3.3, and the same benchmark configured with parallel block size of \( 2^7 \times 2^7 \) elements, referred to as cholesky_b7. This configuration of the program
produces finer-grained parallelism due to the smaller task block size, resulting in 49440 executed
tasks from 18 task types, and a shorter average task duration.

Event sets of different sizes were specified for monitoring during the experiment. To avoid
bias in the selection of events in each set resulting from the potential for particular monitoring
incompatibilities between events, for each event set size 30 random sets of events were selected
for monitoring. This was done by shuffling the total set of events via the C++ standard library
std::random_shuffle, and selecting the first set of events in this list sufficient to retrieve the
required size. At each event set size, these same 30 random event sets were profiled for each
multiplexing period, to ensure the accuracy results can be fairly directly compared across periods.

Three repeated reference executions were carried out for each reference event-pair. The bin
count for the EPD evaluation was set to ten, meaning that after including two external bins for
both dimensions, the task distributions were divided into a maximum of 712 bins, although orders
of magnitude fewer bins were actually populated.

**Results.** The results of the analysis are shown in Figure 9, where each data point represents the
median EPD found across the 30 monitored event sets, with error bars given as the minimum and
maximum EPD. We present the results according to the three aforementioned factors, as follows.

**Varying the number of multiplexed events.** EPD increases with the number of multiplexed events,
agreeing with the expectation that overall accuracy decreases with |EHem| at a constant NPMC. As
particular hardware event incompatibilities existed on the experimental system and the event set
selection was randomized, some of the event sets of size five were necessarily monitored with HEM
while others were not. This produced a large variance in EPD results across the 30 monitored
executions. Incompatible event sets of size five meant that a significant proportion of multiplexing
periods were spent monitoring a small number of events, compared to larger incompatible event
sets where the scheduling algorithm is more likely to find additional, compatible events to be
incorporated in the monitoring during each multiplexing period. The variance thus decreases as
the size of event set increases, as the scheduling strategy is more likely able to make use of more
PMCs during each multiplex period. In addition, less variance is observed for higher event set sizes
as randomized event selection more likely results in more similar event sets, until all event sets are
defined equally at |EHem| = 33.

**Varying the average task duration.** The results of Figure 9 indicate that more accurate execution
profiles can be achieved for benchmarks consisting of tasks with generally longer durations, as
cholesky_b8 achieves lower overall EPD than cholesky_b7 at each equivalent multiplexing period
and multiplexed event set. This agrees with the expectation that the accuracy should increase as
more event counts are sampled during each task.

**Varying the multiplexing period.** The results indicate that accuracy of HEM execution profiles
generally increases as π decreases, since shorter periods increase the likelihood for all hardware
events to be allocated monitoring time during a given task. However, the experiment indicated a
lower-limit to the multiplexing period as a significant run-time overhead was observed at the lower
multiplexing periods. This overhead results from the HEM scheduling code, which is necessarily
executed more often at shorter periods (e.g., twice as often for π = 0.01 ms than for π = 0.02 ms).
This overhead is shown in Figure 10 which, presents the average execution times relative to unin-
strumented OpenStream, of the non-multiplexed executions of event set size 2, and the multiplexed
executions of event set sizes 12 to 33, for each of the multiplexing periods ranging from 4.00 ms
to 0.01 ms. As PAPI reported a maximum NPMC of 11 on Xeon-32, the range 12 to 33 was used to
ensure the event sets were certainly multiplexed even if no incompatibilities were present. For
Fig. 9. EPD Results for `cholesky_b7` and `cholesky_b8` at various multiplexing periods, with slight offsets on the x-axis to ensure error bar visibility. Data points are the median EPD of 30 profiles of randomly selected event sets, with error bars as minimum and maximum.

Fig. 10. Total execution times relative to uninstrumented OpenStream. Data points represent the median either across all non-multiplexed executions of event set size 2, or across all multiplexed executions of event set sizes 15-33, with error bars as minimum and maximum.

each value for the multiplexing period, the average was calculated across all monitored event sets of size 12 to 33, since the HEM scheduling algorithm is bound by $N_{PMC}$, at most making $N_{PMC}$ attempts to schedule events. This means that the HEM scheduling overhead in the experiments is approximately constant for all multiplexed event sets of these sizes at a constant $\pi$. The figure shows that HEM sampling overhead remains low for the relatively short multiplexing periods, with the median relative execution times varying from 1.13 to 1.26 as $\pi$ was reduced from 4.00 ms to 0.10 ms in `cholesky_b7`, and from 1.07 to 1.18 for `cholesky_b8` at the same periods. However, the overhead begins to increase dramatically for periods shorter than 0.10 ms, where the median relative execution time at the multiplexing period 0.01 ms is 3.58 and 3.18 for `cholesky_b7` and `cholesky_b8` respectively. The increasingly large HEM overheads severely limit the reduction of the multiplexing period. In addition to the increased execution time, instrumentation with shorter periods increasingly perturbs the execution, resulting in in significantly different micro-architectural behavior compared to non-instrumented execution.

The EPD results show that when monitoring hardware behavior during short periods of execution such as those of fine-grained task-parallel programs, HEM results in very inaccurate profiles. The experiment shows that this is the case even when only multiplexing a small number of additional hardware events greater that the system is capable of simultaneously monitoring, meaning that
HEM is unable to monitor even relatively small subsets of hardware events. In the next section, we evaluate HEM at a fixed $\pi$ against our execution profile combination approach, where we show that both of the combination strategies presented in Section 3 achieve significantly more accurate profiles than HEM.

5.2 Combination Approaches
To evaluate the accuracy of global profiles generated with our execution profile combination approach, in this section we present experiments in which the BC and LGL combination strategies are applied to executions of three benchmarks across two experimental machines. The resulting accuracies are then compared to profiles generated from executions monitored with HEM, where we show that both BC and LGL combination produces profiles that are significantly more accurate than those produced by HEM.

**Experimental setup.** To investigate the effects of a larger system with more dynamic execution behavior, we evaluated our combination approaches on a 192-core system referred to as Xeon-192, in addition to the 32-core system Xeon-32 used for the previous experiments. Xeon-192 is an Intel Xeon E5-4640 machine with 756 GB main memory distributed over 24 NUMA nodes, running Linux 3.11.0, and also using version 5.4.1 of the PAPI library. Three benchmarks were used in the evaluation: cholesky_b7 from the previous experiments; seidel, an OpenStream implementation of the Gauss-Seidel linear equation solution, consisting of 63745 tasks from 30 task types; and kmeans, an OpenStream K-means clustering benchmark consisting of 32728 tasks from 8 task types.

For LGL combination, the event sets were defined so that they contained no overlapping events. This was done in a greedy process which started with an initially empty event set. To this set, each event of the total set of available events, as reported by PAPI’s utility function $papi\_avail$, was added one at a time until it was no longer simultaneously compatible. When an incompatibility was found, the event was placed into a new event set, and the iteration continued. This resulted in seven event sets on Xeon-32 and four event sets on Xeon-192.

The requirement of BC for overlapping event sets results in a higher number of necessary profiles. The exact number of event sets depends on how the set of all available events is partitioned into sets of events to be added at each combination step, and the set of overlapping events to add to each partition for combination purposes. We leave the specification of an algorithm producing a minimal number of event sets for future work and generated the sets based on a heuristic.

The heuristic begins with the largest event set that includes a variety of events: data cache events, instruction cache events, branch events, and CPU cycle events. Next, using the reference executions of the cholesky_b7 benchmark, the Pearson correlation coefficient for each pair combination of available PAPI events is calculated as an indicator for the strength of the predictive relationship between events. From the remaining events, the heuristic then selects those with the largest number of high correlation coefficients with the already-selected events. Across each pair of relatively highly correlated events, the already previously included events form the overlapping event set for the new profile, and the associated non-overlapping events of each pair are then added to the profile’s event set, until it is no longer simultaneously compatible. This process is repeated until there are no more remaining events. For the 33 events in the experiments, this led to a total of 15 event sets for combination with BC, shown in Table 1.

The multiplexing period $\pi$ on Xeon-32 for the HEM executions was set to 0.10 ms which incurs only a relatively small run-time overhead on the system during the experiments detailed in Section 5.1. On Xeon-192 the experiments were limited to the default multiplexing period of 4.00 ms due to the machine’s requirement for a default, vendor-specific kernel.
On Xeon-32 each of the 528 pair combinations of 33 PAPI events was executed 5 times in order to generate the reference profiles and calculate the calibration TMDs. The BC, BC (Unlabeled), LGL and HEM approaches were repeated 30 times to each generate 30 global profiles for accuracy evaluation via EPD. Access and experimental use of Xeon-192 was more limited, and each reference pair was instead monitored in 3 reference executions, with each of the approaches repeated 3 times to generate 3 global profiles per approach. On both machines, the bin count used for TMD was set to 10 for each dimension, leading to $10^2$ total histogram bins for each TMD calculation. The results of the experiments are presented in the next section.

**Results.** Figures 11 and 12 present the results for Xeon-32 and Xeon-192, respectively. For the Xeon-32, the height of the bars indicates the mean EPD across the 30 generated global profiles and errors represent a 99% confidence interval. Since three global profiles were generated for each approach on Xeon-192, the bar height for this platform indicates the median calculated EPD from the three global profiles, with errors showing the minimum and maximum EPD values.

The significantly higher EPD values for HEM in the figures agree with the previous multiplexing analysis, where multiplexing all 33 PAPI events produces very inaccurate profiles on both experimental systems and on all three benchmarks. On both Xeon-32 and Xeon-192, the results show that our multi-execution combination approaches produce significantly more accurate global profiles than HEM, with EPD values close to the Calibration EPD. On Xeon-32 the 99% confidence
interval EPD results for profiles generated by BC were [1.55,1.63] for cholesky_b7, [2.92,3.15] for seidel, and [2.04,2.13] for kmeans, whereas LGL produced 99% confidence interval EPD results of [1.60,1.68], [2.72,2.95], and [1.85,2.00] for the same benchmarks, respectively. Therefore both BC and LGL produced profiles of very similar accuracies on Xeon-32. These results indicate that the multi-execution combination approaches are capable of generating profiles with hardware event counts that closely approximate what would be recorded if all included events were simultaneously monitored throughout a single execution.

On Xeon-192 the relative accuracy differences between the strategies were pronounced, with BC producing profiles with median EPD equal to 2.15 ± 0.24 for cholesky_b7, 2.82 ± 0.10 for seidel, and 2.32 ± 0.23 for kmeans, with error given as maximum variation from the median. The corresponding LGL EPD results on Xeon-192 were 3.16 ± 0.31, 4.60 ± 0.18, and 7.96 ± 0.32 for the three benchmarks. On this machine, the results show that significantly more accurate profiles are produced by BC combination than by the LGL approach in all three benchmarks. These results suggest that the relative success of BC combination over the more naive LGL combination increases with more execution variability, aligning conceptually with the motivation behind the BC approach.

The experimental results show negligible differences between the EPD of profiles generated via BC combination in which tasks are selected for combination within clusters according to a label-order, and BC (Unlabeled) which uses no strict order for tasks combinations within each cluster. These results suggest that it is not necessary for tasks to be initially labeled to achieve accuracy via BC combination, meaning that BC is likely to be applicable to other run-times where a label graph is not producible.

Therefore the results show that our combination approach is capable of producing significantly more accurate global profiles than the current standard technique of HEM. It avoids the run-time overheads seen in Figure 10 by moving the on-line multiplexing overhead of HEM to an off-line, post-mortem profile combination stage, thus ensuring that the counts collected during each profile are relatively accurate representations of what would occur in a non-instrumented execution. Furthermore, a more sophisticated mechanism to generate the set of hardware events profiled at each BC combination step could both increase the number of simultaneously monitored events in the resulting global profile, and enable better selection of tasks between profiles via more predictive overlapping event sets, to potentially produce global profiles of even greater accuracy.

5.3 Evaluation with Correlation Error

In this section, we further evaluate the combination and HEM global profiles generated during the experiments of the previous section using the Correlation Error metric by Mytkowicz et al. [14], as described in Section 4.4. The results of Correlation Error are then compared to those produced by EPD in order to investigate whether or not correlation analysis is sufficient to evaluate profile accuracy. We show that the numerical difference in correlation does not always accurately reflect the dissimilarity between task distributions, and that TMD (and thus EPD) is better able to do so.

The results of Correlation Error evaluation are given in Figures 13 and 14, for Xeon-32 and Xeon-192 respectively. Comparing these results to the EPD results of Figures 11 and 12, aggregate Correlation Error generally agrees with the EPD evaluation. In our work, we initially explored similar profile evaluation based on Spearman correlation as a non-parametric measure for the strength of monotonic relationship between two variables. However, we found that the complex, potentially multi-modal relationships between hardware events were often inadequately represented, where similar correlation coefficients could be calculated from otherwise very dissimilar task distributions. An illustration of this observation is given in Figure 15. The figure shows the specific task distributions and evaluation results with respect to L1 data cache misses (PAPI_L1_DCM) and store
instructions (PAPI_SR_INS), produced by LGL and BC combinations of cholesky_b7 on Xeon-192 during the experiment presented in Section 5.2. Figure 15a shows the task distribution of an LGL combined profile, for which the Correlation Error is very low despite the distribution itself being very dissimilar from that of the reference in Figure 15c. For this LGL combined profile, the calibrated TMD, as used in our EPD approach, gives a very high dissimilarity result as would be expected from the high dissimilarity shown between Figures 15a and 15c. Figure 15b shows the task distribution of a BC combined profile, which much more closely resembles that of the reference execution and results in a significantly lower TMD, as expected. However, the Correlation Error of the BC combined distribution is instead slightly larger than that calculated for the LGL combined profile. The example distribution comparisons indicate that TMD, and thus EPD, can more precisely capture accuracy of a target hardware event profile, assessing the distribution of tasks directly, rather than the strength of the monotonic relationship between events. Furthermore, unlike Correlation Error which does not account for variation in the reference executions themselves, EPD does consider the expected variability by calibrating the results.

6 RELATED WORK

Previous works have attempted to enable generation of global profiles in the case of limited PMCs. In [10, 11] Mathur and Cook investigate ways to enhance HEM by improving hardware event count interpolation of the PAPI software multiplexing implementation. This user-space library has been superseded by the kernel-space perf_event subsystem since Linux 2.6.31. The authors consider the total event counts of each hardware event measured as the end of single-thread
benchmarks, and compare them to reference total counts. The technique therefore does not attempt to produce or evaluate individual task profiles, as we do. In [9] Lim et al. target the perf_event HEM implementation of Linux kernel 3.11.3. The authors improve interpolation by prioritizing monitoring time-slots to events with less reliably linearly predictable values. Their work also focuses on event sampling on a single thread, and is evaluated by measuring error from a set of known, expected event samples recorded during reference non-HEM execution profiles. In [2], Dimakopoulou et al. identify non-optimal multiplexing event scheduling for constrained hardware events in the Linux perf_event subsystem. By implementing an improved scheduling algorithm, the authors optimize PMC utilization on an Intel architecture simulator. Whilst these works attempt to optimize HEM, the EPD results presented in this paper show that even a small number of additional events drastically reduces the accuracy of hardware event counts profiled during fine-grained task-parallel programs.

Other works have focused on a form of execution trace combination in a technique called trace alignment, introduced by Hauswirth et al. in [6]. Here, single-threaded programs are repeatedly profiled with each profile monitoring a different set of hardware events. The objective of trace alignment is to capture hardware event counts that occurred in the same execution phase across the multiple executions of the program. To do this, the authors apply a technique based on Dynamic Time Warping (DTW) to align the traces in time according to a performance metric common to all the traces. In the paper, a qualitative, visual evaluation is applied. Mytkowicz et al. expand on the DTW-based trace alignment in [14] and introduce the quantitative evaluation technique for generated execution profiles called Correlation Error. We compared the results of EPD with this evaluation approach in Section 5.3 and found that it suffers from drawbacks through its use of Spearman’s correlation coefficient, which potentially imprecisely quantifies the dissimilarity between distributions. The trace alignment approaches attempt to relate the sequentially ordered phases of execution across separate runs of single-threaded programs. They therefore do not account for a multi-threaded execution environment with dynamically scheduled execution instances, where execution phase similarity is not guaranteed per thread or collectively. Our work targets this environment directly by constraining the assembly and subsequent reasoning of executed regions of the distinct programs to those which had similar run-time execution (BC), or precise definition in the program structure (LGL).

In [5], Gonzalez et al. apply a clustering technique based on a single set of common hardware events to identify similarities between instances of code-regions in parallel executions. Event sets that include these events are multiplexed in a single execution. By clustering code-regions on common events, an averaged summary of the program’s behavior is generated by extrapolating per-cluster event counts. The authors compare the extrapolated results for each cluster to those found when clustering on reference executions. The authors produce a set of event counts for each cluster, and therefore do not produce or evaluate a global profile where each individually executed instance of a code-region is associated with hardware event counts, for all events.

7 CONCLUSION

We presented new techniques for generating accurate and complete execution profiles of dynamic task-parallel programs, by assembling combined profiles containing monitored counts for all available PAPI hardware events. We detailed and formally defined our two combination techniques: Label Graph Location (LGL), which combines tasks according to task creation relationships; and Behavior Clustering, (BC) which combines tasks based on similar microarchitectural behavior. We further presented a metric based on the Earth Mover’s Distance that quantitatively evaluates the accuracy of execution profiles by calculating the similarity between its event counts statistical distribution to those of reference executions. We applied this metric to provide a quantitative
analysis of Hardware Event Multiplexing (HEM) against our combination techniques. In our experiments, we found that profiles generated using HEM suffer from a very high degree of inaccuracy. The results showed that our approach produced execution profiles with complete and vastly more accurate hardware event counts.

REFERENCES


