Adaptive Parallel Compressed Event Matching

Mohammad Sadoghi [#], Hans-Arno Jacobsen [*]
[#] *IBM T.J. Watson Research Center*
[*] *Middleware Systems Research Group, University of Toronto*

***Abstract*—The efficient processing of large collections of patterns expressed as Boolean expressions over event streams plays a central role in major data intensive applications ranging from user-centric processing and personalization to real-time data analysis. On the one hand, emerging user-centric applications, including computational advertising and selective information dissemination, demand determining and presenting to an end-user the relevant content as it is published. On the other hand, applications in real-time data analysis, including push-based multi-query optimization, computational finance and intrusion detection, demand meeting stringent subsecond processing requirements and providing high-frequency event processing. We achieve these event processing requirements by exploiting the shift towards multi-core architectures by proposing novel adaptive parallel compressed event matching algorithm (A-PCM) and online event stream re-ordering technique (OSR) that unleash an unprecedented degree of parallelism amenable for highly parallel event processing. In our comprehensive evaluation, we demonstrate the efficiency of our proposed techniques. We show that the adaptive parallel compressed event matching algorithm can sustain an event rate of up to 233,863 events/second while state-of-the-art sequential event matching algorithms sustains only 36 events/second when processing up to five million Boolean expressions.**

I. Introduction

Efficient event processing (i.e., event matching) is an integral part of a growing number of web and data management technologies ranging from user-centric processing and personalization to real-time data analysis. In user-centric processing applications, there are computational advertising [25] and location-based services for emerging applications in cospaces [1]. Common to all of them are patterns and specifications (e.g., advertising campaigns, job profiles, service descriptions) modeled as Boolean expressions, XPath expressions, or SQL queries and incoming user information (e.g., user profiles and preferences) modeled as events using attribute-value pairs, XML documents, or relational tuples. In the real-time analysis domain, there are (complex) event processing [2], [6], [25], [21], [22], XML filtering [4], [19], intrusion detection [24], [8], and computational finance [23]; Similarly, common among these applications are predefined sets of patterns (e.g., queries and attack specifications) modeled as subscriptions and streams of incoming data (e.g., relational tuples, data packets) modeled as events.

Unique to user-centric processing and personalization are strict requirements to determine only the relevant content (e.g., ads) with respect to user demographics and interests [25]. Furthermore, user-centric processing demands scaling to millions of patterns and specifications (e.g., advertising campaigns) and millions of users (i.e., consumers) while meeting processing latency constraints in the subsecond range, to achieve an acceptable service-level agreement. Noteworthy, users often within certain demographic proximity are potentially interested in similar content and demand receiving the same contents (e.g., following popular local news, songs, or products).

We argue that in order to meet these scaling requirements, it is essential to develop efficient parallel matching algorithms that are (1) compatible with state-of-art high-dimensional indexing structures [21], [22], [20] and (2) capable of exploiting and extracting similarity among incoming events. In other words, matching algorithms that are able to efficiently identify and aggregate user profiles with similar interests and to amortize the cost of finding user-relevant content over many users.

Unique to real-time data analysis applications are critical requirements to meet the ever growing demands in processing large volumes of data at predictably high-throughput and low latencies across many application scenarios [12], [3]. The need for increased processing bandwidth is the key ingredient in high-throughput real-time data analysis that enables processing, analyzing, and extracting relevant information from streams of incoming data. We argue that in order deliver real-time event processing in presence of continuous proliferation of data and bandwidth, it is inevitable to expand the research horizon beyond the conventional sequential matching algorithms and adopt other key enabling technologies such as multi-core architectures. There is a mounting evidence that the number of cores is predicted to grow exponentially in the coming years [14], [18], and that multi-cores are successfully being used to accelerate many data management applications [15]. Therefore, the next generation of real-time event processing must exploit parallel hardware such as multi-core architecture to sustain the expected growth of demands.

These requirements constitute challenges of paramount importance for applications that rely on event processing (with our primary focus on patterns defined as Boolean expressions). To address these challenges, first and foremost, we develop a parallel event matching algorithm over an exiting state-of-the-art Boolean expression index structure [21], [22], [20]. Our parallel matching algorithm exploits multi-threading and the shared-memory architecture of modern multi-core processors in order to scale to millions of expressions and to meet the high-throughput demands. Such parallel matching algorithms have received little attention by prior Boolean expression matching approaches (e.g., [2], [6], [25], [21], [22]). Second, we introduce a stream-aware adaptive parallel algorithm that can identify and utilize the overlap (or the similarity) within the event stream. Such an effective online event stream reordering technique have also been largely ignored by the prior-art in matching algorithms (e.g., [2], [6], [25], [21], [22]).

In short, the contributions of this paper are summarized

as follows (i) A novel **P**arallel **C**ompressed event **M**atching (PCM) algorithm over a bitmap-based event encoding detailed in Section V; (ii) an efficient **O**nline **S**tream **R**e-ordering (OSR) technique presented in Section VI; (iii) an **A**daptive **P**arallel **C**ompressed event **M**atching (A-PCM) algorithm that adaptively chooses between matching over compressed or uncompressed events depending on stream similarity discussed in Section VI; and (iv) a comprehensive experimental evaluations that establishes the effectiveness of our approach in Section VII.

In what follows, we use the terms "event stream" and "stream" interchangeability. Furthermore, we formulate our "event processing" or "event matching" problem using the well-known publish/subscribe (pub/sub) matching problem [2], [6], [25], [7], [21], [22] described in Section II.

II. Expression Language and Data Model

In our pub/sub matching model, the input is a set of subscriptions (a conjunction of Boolean predicates) and an event (an assignment of a value to each attribute), and the output is a subset of subscriptions satisfied by the event. In fact, we also model our events as Boolean expression to enable a more powerful event language.

In short, we define a Boolean expression as a conjunction of Boolean predicates. A predicate is a triple, consisting of an attribute uniquely representing a dimension in n-dimensional space, an operator, and a set of values, denoted by $P^{\texttt{attr},\texttt{opt},\texttt{val}}(x)$, or more concisely as $P(x)$. A predicate either accepts or rejects an input x such that $P^{\texttt{attr},\texttt{opt},\texttt{val}}(x) : x \longrightarrow \{\texttt{True}, \texttt{False}\}$, where $x \in \texttt{Dom}(P^{\texttt{attr}})$ and $P^{\texttt{attr}}$ is the predicate's attribute. Formally, a Boolean expression be is defined over an n-dimensional[1] space as follows:

$$\{P_1^{\texttt{attr},\texttt{opt},\texttt{val}}(x) \wedge \cdots \wedge P_k^{\texttt{attr},\texttt{opt},\texttt{val}}(x)\},$$
$$\textit{where } k \leq n; \quad i,j \leq k, \; P_i^{\texttt{attr}} = P_j^{\texttt{attr}} \quad \textit{iff} \quad i = j.$$

We support an expressive set of operators for the most common data types: Relational operators ($<$, $\leq$, $=$, $\neq$, $\geq$, $>$), set operators ($\in$, $\notin$), and the SQL BETWEEN operator.

The expressiveness of our subscription and event language enables supporting a wide range of matching semantics. In this paper, we focus primarily on the classical pub/sub matching problem: *Given an event ϵ and a set of subscriptions $\mathbf{\Sigma}$, find all subscriptions $\sigma_i \in \mathbf{\Sigma}$ satisfied by ϵ*. We refer to this problem as *stabbing subscription* $\texttt{SQ}(\epsilon)$ given by:

$$\texttt{SQ}(\epsilon) = \{\sigma_i | \; \forall P_q^{\texttt{attr},\texttt{opt},\texttt{val}}(x) \in \sigma_i, \; \exists P_o^{\texttt{attr},\texttt{opt},\texttt{val}}(x) \in \epsilon, \; P_q^{\texttt{attr}} = P_o^{\texttt{attr}}, \exists x \in \texttt{Dom}(P_q^{\texttt{attr}}), \; P_q(x) \wedge P_o(x)\}$$

III. Boolean Expression Indexing

We provide an overview of BE-Tree before presenting our proposed adaptive parallel compressed matching (A-PCM) and online stream re-ordering (OSR) algorithms.

BE-Tree is a an index structure for indexing a large collection of Boolean expressions (i.e., subscriptions) and for efficiently retrieving the most relevant matching expressions given a stream of incoming events [21], [20]. In general, BE-Tree is an n-ary tree structure in which a leaf node contains the actual data (Boolean expressions) and an internal node contains partial information about data (e.g., an attribute and a range of values) in its descendant leaf nodes. The tree consists of three classes of nodes: A p-node (partition node) for storing the partitioning information, i.e., an attribute. A c-node (cluster node) for storing the clustering information, i.e., a range of values. And an l-node (leaf node), being at the lowest level of the tree, for storing the actual data. Moreover, p-nodes and c-nodes are logically organized in a special directory structure for fast tree traversal and search space pruning. The overall tree structure is depicted in Figure 1.

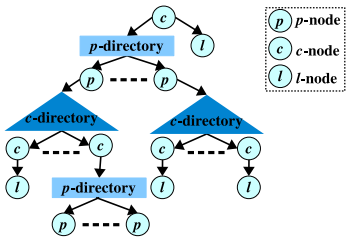


Fig. 1. An overview of BE-Tree data structure.

IV. Bitmap Event Encoding

In this section, we focus on an execution model that exploits opportunities to further accelerate the matching algorithm computation that relies on a bitmap-based event encoding [20]. The main advantage of the bitmap-based technique is to ensure exactly-once evaluation of every distinct predicate. At a high-level, this technique exploits predicate inter-relationships (i.e., predicate covering) and guarantees exactly-once evaluation of distinct predicates. At a low-level, it minimizes the memory footprint through an efficient bitmap representation that also speeds up the computation using low-level bitwise operations to preserve cache locality.

One of the matching algorithm properties, in addition to search space pruning, is minimizing the true candidate computations, i.e., the evaluation (and the encounter) of common predicates exactly once. The scope of this objective is generalized by also ensuring that each distinct predicate is always evaluated exactly once. Conceptually, in this paradigm, as the matching algorithm examines an incoming event, an efficient structure is maintained (with respect to both time and space) to store the evaluation result (`True` or `False`) of each distinct predicate in the subscription workload. This structure is represented as a bit-array, in which each bit indicates whether or not a distinct predicate has been evaluated to `True` (or `False`). Exploiting a bit-array not only provides fast read/write access to predicate evaluation results, but also its compact encoding could potentially be resident entirely in the cache of a modern processor, which significantly reduces the number of cache-misses.

Therefore, the *bitmap-based event encoding*, pushes the limit of matching algorithms' standard predicate evaluation by also enabling exactly-once evaluation and incorporating the predicate inter-relationships (e.g., predicate covering) through precomputation and storing of predicate coverings, which is achieved partly due to the exploitation of the discrete and

[1]Without loss of generality, we assume each dimension has a domain cardinality of size N.

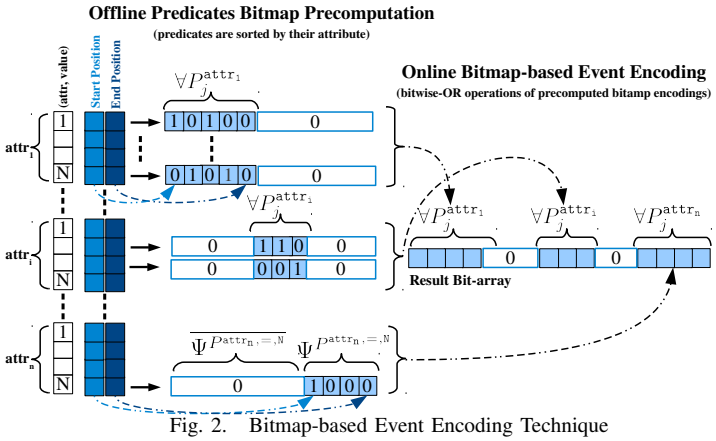


Fig. 2. Bitmap-based Event Encoding Technique

finite domain properties. The bitmap structure is over the set of all distinct predicates such that for any given attribute-value pair, i.e., the equality predicate $P^{\texttt{attr},=,\texttt{value}}$, all distinct predicates that are relevant for P are precomputed and stored in the bitmap. Therefore, instead of individually evaluating every relevant distinct predicate for a given event's equality predicate, the evaluation results are precomputed for every possible distinct predicate that is affected by any given event's predicate. The set of affected predicates by the equality predicate P, denoted by Ψ^P, is defined as follows

$$\Psi^P = \{P_i |\ \forall P_i^{\texttt{attr},\texttt{opt},\texttt{val}}(x) \in \mathbf{\Psi}, \\ P^{\texttt{attr}} = P_i^{\texttt{attr}}, \exists x \in \texttt{Dom}(P^{\texttt{attr}})\},$$

where $\mathbf{\Psi}$ is the set of all distinct predicates.

Consequently, the set of all distinct predicates that are not affected by P are given by $\overline{\Psi^P} = \mathbf{\Psi} - \Psi^P$. The bitmap is constructed by determining the sets Ψ^P and $\overline{\Psi^P}$ for each P, i.e., P is formed by enumerating over the discrete values of each attribute (a dimension in the space) in order to construct an equality predicate P. Next, each predicate is evaluated in Ψ^P for a given P and stored as the result in the bitmap. Also, the set $\overline{\Psi^P}$ is automatically filled with zeros because none of the predicates in the set $\overline{\Psi^P}$ are affected by P. The overall structure of the bitmap and its organization of Ψ^P and $\overline{\Psi^P}$ are illustrated in Figure 2.

The most striking feature of the bitmap structure is its highly sparse matrix structure because the set of bits represented by $\overline{\Psi^P}$ are all zero, and given the high-dimensionality of our problem space, the bitmap is space-efficient because $|\Psi^P| \ll |\overline{\Psi^P}|$. In particular, if the corresponding bits are re-ordered by clustering Ψ^P and $\overline{\Psi^P}$ (as shown in Figure 2), an effective space-reduction can be achieved for the bitmap structure. The bitmap space saving ratio $\frac{|\overline{\Psi^P}|}{|\Psi^P|}$, through reordering of the bits in the bitmap, is directly proportional to the number of attributes l if the predicates are sorted based on their attributes and given by

$$l \propto \frac{|\overline{\Psi^P}|}{|\Psi^P|}.$$

This ratio is further influenced by the distribution of predicates over each attribute. For instance, for certain domain values $value$ over attribute $attr$, there may be no predicate such that the $value$ falls in any predicate's range of values; thereby, implying that the set $\Psi^{P^{\texttt{attr},=,\texttt{x}}}$ consists of only zero

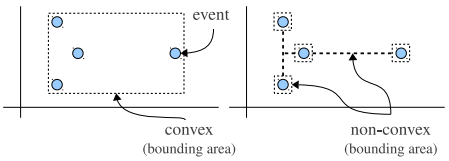


Fig. 3. Convex vs. non-convex minimum bounding area

bits. Such a distribution of predicates results in further space reduction because neither $\Psi^{P^{\texttt{attr},=,\texttt{x}}}$ nor $\overline{\Psi^{P^{\texttt{attr},=,\texttt{x}}}}$ need to be stored explicitly in the bitmap. This reduction in space also improves, as a byproduct, the bitwise operations during the matching process. In addition, there are potential research opportunities to develop more effective bit reordering techniques (i.e., a predicate topological sort order) to further improve the space saving ratio. In general, the minimum number of predicates that must be maintained for each P (a lower-bound on size of the set Ψ^P) are those distinct predicates that are satisfied by P. This minimum set is defined as follows:

$$\Psi^P_{\texttt{min}} = \{P_i |\ \forall P_i^{\texttt{attr},\texttt{opt},\texttt{val}}(x) \in \mathbf{\Psi}, P^{\texttt{attr}} = \\ P_i^{\texttt{attr}}, \exists x \in \texttt{Dom}(P^{\texttt{attr}}),\ P(x) \wedge P_i(x)\},$$

During the matching process, upon arrival of a new event e, the precomputed bitmap index is utilized to efficiently compute all distinct predicates that are satisfied by the incoming event. This is carried out by a bit-wise OR-operation of relevant rows in the bitmap index in order to fully construct the *Result Bit-array*: A bit-array in which each bit corresponds to a distinct predicate, where a bit with value 1 signifies that the corresponding predicate is `True`, otherwise `False`. The Result Bit-array is constructed as follows:

$$\textit{Result Bit-array} = \bigcup_{P_i \in e} \{\Psi^{P_i}, \overline{\Psi^{P_i}}\},$$

where no actual operation is required to account for $\overline{\Psi^{P_i}}$ sets.

The Result Bit-array, namely, the *bitmap-based event encoding*, can entirely fit in the cache as long as the subscription workload contains only on the order of a few million of distinct predicates appears the active and hot set, for which only a few mega bytes of cache is sufficient[2]. However, the potential source of cache misses is not limited to Result Bit-array accesses, in fact, another internal data structure that generates cache misses (in addition, to general pointer chasing of any tree structure) is the representation Boolean expressions (subscriptions).

In the bitmap-based evaluation, since each subscription requires only to keep an array of references to Result Bit-array, it is feasible to store all subscriptions in a as cache-conscious blocks of 2-dimensional arrays of references. Moreover, clusters of subscriptions could be summarized using *2-dimensional subscription representation* and be fitted in the processor cache; thus, substantially reducing the number of cache misses during subscription evaluations in order to improve the overall matching time.

The bitmap maintenance, the removal of predicates, is done by marking predicates as pseudo deleted and a background process periodically re-claims the space. However, adding new

[2]The processor used in our experiment has two shared-cache blocks of size 6144KB.

distinct predicates[3] is handled by storing them in an overflow-bitmap data structure, which is periodically merged with the main bitmap. Notably during the merge process, the matching can continue by combination of using the old bitmap and/or simply re-evaluating each predicate on-the-fly as needed.

V. Matching on Compressed Events

By exploiting the bitmap-based encoding from Section IV, we develop a parallel matching algorithm (PCM) that carries out subscription matching in parallel over compressed events. The compressed events are created by coalescing multiple bitmap-based event encodings into one. Our event compression technique, unlike convex-shaped minimum bounding area techniques, prevalent in the database multi-dimensional indexing literature [10], geometrically represents many events using a non-convex minimum bounding area that avoids including in the bounding area any empty space, i.e., dead space, as shown in Figure 3. Inclusion of dead space results in increased chance of false candidates.[4]

A. Parallel Compressed Matching

In a sequential processing setting, as an event arrives, we first compute a bitmap-based encoding of the event, then we use the bitmap encoding for traversing a tree-based index structure (e.g., traversing BE-Tree) and finding the match results. This process is illustrated in Figure 4.

In a parallel processing setting with compression, as part of our PCM algorithm, we leverage the bitmap-based encoding to provide an effective compression algorithm that coalesces a set of incoming events into a single representation, namely, a *compressed event*, and subsequently traverse the index solely using this compressed event. Not only does the compressed event inherit all desired properties of the bitmap-based encoding such as cache-consciousness and exactly-once predicate evaluation, but it also geometrically represent a set of events as a non-convex minimum bounding area with no dead space, an effective way for reducing false candidates.

The efficient construction of a compressed event is achieved by bitwise-OR operations over the bitmap-based encoding of a set of events. The compressed event is then used to index traversal once for all compressed events and identify all leaf nodes with potential matching subscriptions. In the final stage of PCM, we determine for each event the actual matched subscriptions using the individual bitmap-based encoding for each event. The five stages of the PCM algorithm for matching over compressed events is depicted in Figure 5.

Notably, the PCM algorithm can essentially solve the matching problem for a set of events with a single index traversal and with a single pass over all relevant leaf nodes. Thus, our approach amortizes memory accesses and matching cost over a set of events exhibiting a substantial improvement in matching throughput (and arguably in matching latency).

The PCM algorithm is also amenable to a high degree of parallelization. In particular, Stage 1 of our algorithm (as shown in Figure 5) can be implemented using n threads for n events

[3]Note that adding new predicates covered by existing predicates is trivial.

[4]*False candidates* are subscriptions that are not identified falsely as potential matches and must be filtered.

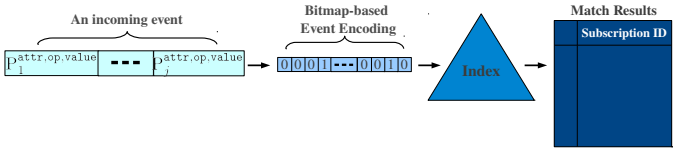


Fig. 4. Overview of Matching Algorithm using Bitmap-based Encoding

such that each thread is responsible to compute the bitmap-based encoding for each event. Similarly, Stage 2 coalesces the bitmap encoding of n events into a single compressed event, which again can be done in parallel if the bitwise-OR computation is horizontally partitioned across n threads. Let the size of the bitmap encoding be given by $\texttt{length}(bitmap)$ bits, then the bitmap-based encoding is horizontally partitioned into chunks such that each thread is assigned one chunk, and a chunk is $\texttt{lenght}(bitmap)/n$ number of consecutive bits (Stage 3). Furthermore, the chunk size is adjusted to be a multiple of a cache-line size in order to avoid *false sharing*, i.e., false sharing occurs when in a shared-memory multi-threaded program, threads' local objects fall within the same cache-line, thereby unnecessaryly triggering a cache-coherency protocol. False sharing is known to substantially reduce the overall performance of multi-threaded applications [16].

The resulting compressed bitmap encoding of all events (the compressed event) is then passed onto to the index structure (Stage 4). During the matching process (within a single event) all matching candidate subscriptions for the compressed event are collected and returned. Finally, in Stage 5 of the PCM algorithm, for each returned set of subscriptions, every subscription residing is matched individually for each event (using its pre-calculated bitmap encoding in Stage 1). Therefore, at this point, all the matching subscriptions can be determined exactly for each event. In addition, each event can carry out the final check within a single thread. Most importantly, all parallelized stages of our PCM algorithm are independent within each stage and require no coordination, which again improves running time considerably. In addition, only limited overall coordination and no data consistency is required across threads. In fact, the only coordination requirement is the *in-order processing of stages*: Each thread in a stage must wait until all running threads have completed before proceeding to the next stage.

Lastly, it is important to note when matching with compressed events using the bitmap-based encoding (Stages 4-5), that all predicate evaluations including predicates associated with index internal predicates (e.g., predicate for describing an index node), are translated into direct lookup in the *Result Bit-array*. As a result, eliminating the need for re-evaluating any predicates for any of the compressed events. The predicate evaluation of the PCM algorithm when applied to BE-Tree is demonstrated in Figure 6.

B. Traversal Unrolling

The first PCM algorithm optimization is the dynamic loop unrolling (i.e., loop unwinding) for index traversal. In the base PCM algorithm, the tree is traversed recursively (resulting in a depth-first search), which continuously switches between leaf node scanning (and parallel matching) and traversing the remainder of the tree. This constant switching introduces

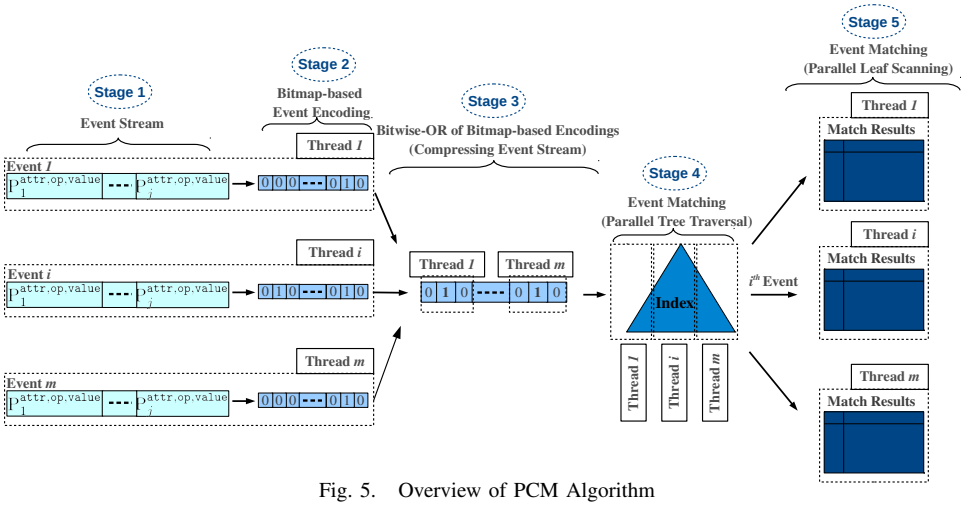


Fig. 5. Overview of PCM Algorithm

additional overhead every time leaf node scanning is initiated: (1) events-to-thread assignment, (2) threads-to-processor or threads-to-core assignment, and (3) thread creation and destruction depending on the thread implementation.[5]

As discussed previously, although in theory, every thread is run independently of other threads, the re-assignment, that occurs after every leaf node scanning, may reshuffle threads and core assignments causing false sharing. For example, each thread keeps track of a number (and actual) of matches for each event and the thread's local variables are brought into non-shared L1 cache of the core that is running the thread. Now suppose, when the first leaf node is examined, the event e_i is assigned to the thread t_j running on the core c_k, but upon a subsequent assignment, suppose that the thread assignment is re-shuffled such that t_j is now assigned to c_{k+1} and t_j continues to be responsible for the event e_i. Thus, the e_i counter is brought into the non-shared L1 cache of both c_k and c_{k+1}. Now, as soon as the e_i matching counter is incremented by t_j, an expensive cache coherency protocol is triggered, invalidating and synchronizing c_k's local cache despite the fact that c_k will never be read and could have simply been ignored (i.e., an instance of false sharing).

The three issues raised above can be addressed, if tree traversal and leaf scanning steps are de-coupled, conceptually, resulting in a dynamic tree traversal unrolling. Thus, we introduce traversal unrolling in PCM as follows. First, traverse the index and collect all candidate leaf nodes (stored in a form of list-based structure), and, second, we iterate through all leaf nodes and scan their contents. In this way, the threads-to-processor and threads-to-core assignments occur only once, which eliminates the problem of false sharing and cache coherency all together. Furthermore, unrolling also avoids unnecessary thread creation and destruction.

C. Parallel Path Traversal

The next PCM algorithm optimization is the acceleration of the matching computation by parallelizing the index traversal. In which conceptually index is divided into sets of regions, and each region is assigned to one thread. Each region can be defined as a set of paths that needs to be examined. Recall

[5]For instance, in certain Open Multiprocessing libraries, the programmer has no direct access to control the underlying thread creation/destruction.

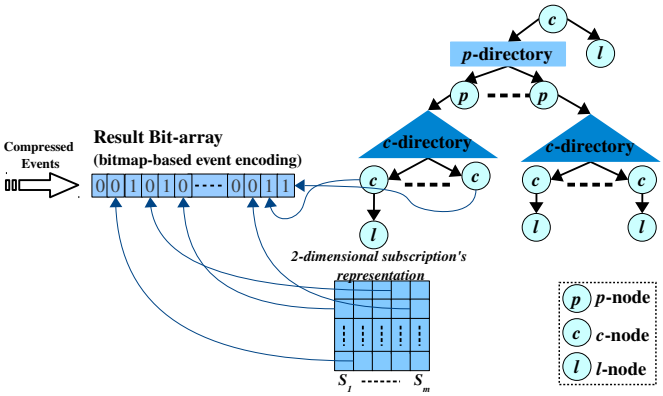


Fig. 6. PCM Predicate Evaluation (Stages 4-5 in Figure 5)

that most high-dimensional tree indexing structure results in a multi-path traversal. To form these regions, different scheduling schemes for paths-to-thread assignment are possible.

Under the static assignment, a round-robin assignment of paths to threads has proven to be effective in our evaluation (and is a default option in all of our experiments). One possible shortcoming of a round-robin assignment is underutilization of processing power for certain workload distributions such that for a given event certain threads remain idle because their assigned region has no path that is relevant for the given event(s). To the contrary, the dynamic assignment relies on the paths-to-thread assignment at run-time after identifying potential paths for a given (compressed) event.

Note that the parallel path traversal optimization does not change the tree traversal and leaf scanning de-coupling of the PCM algorithm that we introduced in the last section.

D. Matching Pipeline

A natural pipeline arises in our PCM proposed compressed matching computation, namely, bitmap-based event encoding, event stream compression, and event matching. This pipeline can be mapped onto the symmetric multiprocessing (SMP) architecture that connects a set of identical processors using shared main memory (spread over several memory banks all connected by a system bus.) The main challenges that arise in exploiting an SMP is limited data bandwidth with respect to the available processing bandwidth. In order to avoid any processor idleness, one must ensure that processors are fed

directly by their closest memory banks, coupling memory and processor. We achieve memory and processor coupling by relying on non-uniform memory access (NUMA) and by replicating the index in the processor's local memory banks.[6]

E. Parallel Implementation

We use the portable Open Multiprocessing (OpenMP) library for implementing our PCM algorithm and our proposed optimization including compression, traversal unrolling, and parallel path traversal techniques. In addition, OpenMP provides both static and dynamic thread scheduling as a tunable parameter, which we utilized in our evaluation.

VI. Online Event Stream Re-ordering

To realize the true power of the PCM algorithm's compressed matching, the compressed events must be similar. A naive solution would simply rely on the raw event order, which may fluctuate and contain random noise (i.e., non-similar events), that deteriorates the effectiveness of the compression technique. Therefore, it is essential to account for noise in the event stream and proactively bring together similar events that are close to each other but not adjacent. This goal is achieved by our online stream re-ordering technique (OSR).

Unlike the exiting online re-ordering algorithms that rely on variation of locality-sensitive hashing, we argue that although these approaches are suitable for re-ordering of data over high-dimensional space (similar to our event space), yet these approaches are inadequate in our setting. Basically, any online re-ordering algorithm that relies on locality-sensitive hashing can assign a set of events into a set of clusters (i.e., bucket), where the number of clusters is much smaller than the number events). Ideally, each cluster will contain similar events. Such a re-ordering algorithm conceptually resembles approximate sorting.

However, there are four major shortcoming of any algorithm that relies solely on locality-sensitive hashing. First, locality-sensitive hashing requires tuning many parameters, a non-trivial and daunting task [5]. Second, there is no efficient technique for reasoning about the similarity among events in each cluster. We refer to this as discovering the degree of *stream heterogeneity*. Third, more importantly, there is no way for controlling the size of these clusters, i.e., the distribution of events across clusters maybe highly skewed. Forth, in locality-sensitive hashing, each hashed data item over the high-dimensional space is assumed to be fully defined over the entire space. However, in our domain, most events, specify values for only small sets of dimensions and provide no values for the remaining dimensions. We refer to this irregularity as *incomplete event data*.

To address these shortcomings, we propose a new online re-ordering technique OSR that solves all the above-mentioned shortcomings. Notably, our online re-ordering exploits BE-Tree to re-order the event stream. As discussed before, BE-Tree is designed for indexing Boolean expressions over a high-dimensional space, coping with the curse of dimensionality is also one of the strength of locality-sensitive hashing.

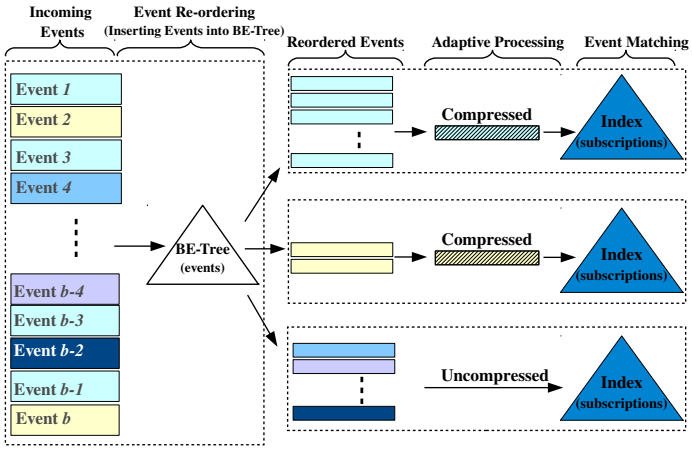


Fig. 7. Overview of Stream Online Re-ordering and Matching

In short, the PCM algorithm operates as follow. In order to re-order the event stream, events are buffered into batches (of size b), then the batched events are inserted into BE-Tree. The resulting tree is a set of clusters that hold similar events together. All events in each event cluster are then compressed and matched in turn.

Since BE-Tree takes as inputs the minimum and maximum cluster size, then by design, it offers control over the size of each cluster. In particular, it is desirable that the minimum size of each event cluster should be larger than the number of available threads in order to achieve the maximum benefit and avoid thread idleness.[7] In addition, BE-Tree also supports incomplete event data by design [21]. More importantly, the heterogeneity of each cluster can efficiently be derived based on the depth of any cluster in the tree. For example, all events assigned to the root cluster do not have predicates on any common attributes, or all events in a cluster at depth two of the tree, have predicates defined on at least two common attributes, where the range values defined by these predicates are also overlapping.

Exploiting the most important feature of the OSR algorithm is the ability to reason about stream heterogeneity and dynamically adapt to similarity among events in the stream. Therefore, we proposed an adaptive parallel compressed matching algorithm (A-PCM) that utilizes both the parallel compressed matching algorithm and the standard parallel matching algorithm.[8]

Our A-PCM algorithm works as follows. As the stream is batched and re-ordered on-the-fly using the OSR technique, for each batch of events, all event clusters below a certain similarity threshold are processed as uncompressed using the standard parallel matching algorithm while event clusters above a certain similarity threshold are processed using the compressed matching technique. All events in each cluster

[6]Using NUMA memory-processor coupling, we have achieved linear scaling of the PCM algorithm when scaling to eight processors of an 8-core Intel Xeon X6550 (results are omitted due to the lack of space).

[7]The effect of the event cluster size is studied in Section VII-B.

[8]For the standard parallel algorithm, we process n events in parallel using n threads, where each event is assigned one thread and all threads operate independently in complete isolation [7].

TABLE I
SYNTHETIC AND REAL WORKLOAD PROPERTIES

	Workload Size	Number of Dimensions	Match Prob	Stream Similarity	Number of Distinct Predicates		Match Prob (DBLP Data)
Size	1M-5M	5M	5M	5M	5M		5M
Number of Dim	128	32-768	128	128	128		677
Cardinality	48	48	48	48	48-3072		26
Number of Sub Pred	7	7	7	7	7		8
Number of Event Pred	15	15	15	15	15		16
Pred Avg. Range Size %	12	12	12	12	12		12
% Equality Pred	0.4	0.4	0.4	0.4	0.4		0.4
Match Prob %	1	1	0.01-9	(≈ 0) or 1	(≈ 0) or 1		0.01-9
Stream Similarity %	70	70	70	10-100	70		70

that satisfy the threshold condition are compressed together. Consequently, the number of compressed clusters of events are proportional to the number of clusters that are above the similarity threshold. The A-PCM algorithm is depicted in Figure 7.

VII. EVALUATIONS

We present a comprehensive evaluation of our PCM, A-PCM, and OSR algorithms using both synthetic and real datasets. The experiments were conducted on a machine with two Quad-core Intel Xeon X5450 processors running at 3.00 GHz with two 6MB of shared L2 cache and 16GB of memory. All algorithms are implemented in C (compiled with version gcc 4.1.2 and O3 optimization level) using OpenMP 2.2 and are extensions of the BE-Tree 1.3 open source project.[9]

A. Experiment Overview

First, we demonstrate the effectiveness of the PCM optimizations and the importance of the A-PCM algorithm when applied to BE-Tree. Second, we compare our adaptive A-PCM algorithm with BE-Tree, which is known to be one of the fastest matching algorithm [21], under controlled experimental conditions, before showing results on real-world data. All workload are generated using the open source Boolean expression generator, BEGen[10]. In particular, in our evaluation, we varied *workload distribution*, *workload size*, *space dimensionality*, *event matching probability*, *number of distinct predicates*, and, most importantly, *event stream similarity*.

The value of each parameter in our synthetic and real-world workloads are summarized in Table I, where Columns 1-5 are synthetic data and Column 7 is a real data. Each column corresponds to a different workload setting while each row corresponds to the actual value of various workload parameters. All workloads are generated by BEGen.

In our evaluation, we generate workloads having a controlled degree of *event matching probability*, which ensures that each event in the event workload matches a certain percentage of all subscriptions. For example, the matching probability of $m\%$ means that each event matches at least $m\%$ of all subscriptions. Similarity, BEGen enables controlling the *event stream similarity*. For instance, an event workload with stream similarity of $s\%$ means that each event has been replicated k times using a Gaussian distribution, in which $(1-s)\%$ of predicates in the replicated events have been replaced using new random predicates, e.g., if $s = 100\%$, then the k replicated events are all identical, and if $s = 0\%$, then all the k replicated events are completely different. As a result, if the initial event workload has a matching probability of $m\%$, then by applying the stream similarity technique, the final event workload may exhibit on average a lower event matching probability.

In our micro experiments, we individually study the effectiveness of each optimization in the PCM algorithm, namely, the unoptimized parallel compressed matching algorithm (C), the parallel compressed algorithm with traversal unrolling (C-U), the parallel compressed algorithm with traversal unrolling and parallel path processing (C-U-PP), the parallel compressed algorithm with traversal unrolling, parallel path processing combined with the OSR technique (C-U-PP-RE), and, finally, our "flagship" algorithm, the adaptive parallel compressed algorithm with traversal unrolling, parallel path processing and OSR technique (A-PCM). Also, in all our experiments, the matching time for C-U-PP-RE and A-PCM also includes the time taken for online stream re-ordering.

In our macro experiments, we consider (1) BE-Tree [21], (2) BE-Tree with a bitmap-based encoding (Bitmap) [20], (3) base parallel BE-Tree, which simply process n events over n threads in parallel (Parallel) [7], (4) the A-PCM algorithm when applied to BE-Tree, where the stream batch size for re-ordering is set to 1024, the event cluster size for stream re-ordering is set to 8, and the stream similarity threshold is set to 2 (A-PCM).

Finally, in all our experiments, the internal parameters of BE-Tree, in particular, the minimum and maximum cluster size, have been assigned based on guidelines provided in [21], e.g., the maximum cluster size is set to 5 and 20 for workloads with matching probability $m < 1\%$ and $1\% \leq m \leq 9\%$, respectively, while the minimum cluster size is fixed at 3.

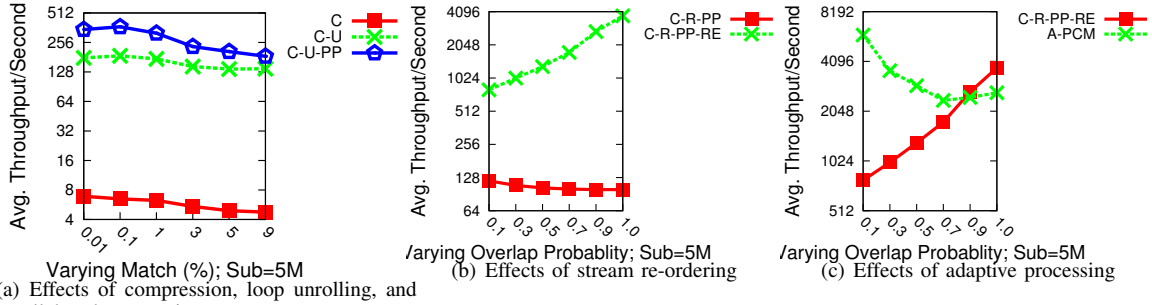
B. Micro Experiments

We establish the importance of the PCM optimizations by varying the degree of both stream similarity and matching probability. We have experimented with both uniform and Zipf distributions and real-world data workload distributions, in which a similar overall trend has been observed in all variations. As a result, in the interest of space, we have included results for uniform distribution unless stated otherwise. The micro experiments for various optimizations are shown in Figure 8 and effects of internal parameters for the adaptive parallel compressed algorithm are presented in Figure 9.

Effects of Traversal Unrolling. One of the main challenges in parallelizing an algorithm is the need to reduce *locking* and avoid *false sharing*, which together could offset any potential gain resulting from parallelism [16]. The parallel compressed algorithm C has already eliminated the need for locking by assigning each event to exactly one concurrent thread. However, without fine control over event-to-thread or

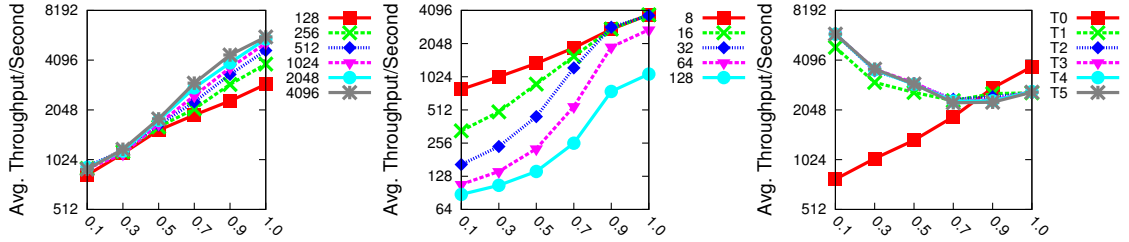
[9] http://msrg.org/project/BE-Tree

[10] http://msrg.org/datasets/BEGen



(a) Effects of compression, loop unrolling, and parallel path processing (b) Effects of stream re-ordering (c) Effects of adaptive processing

Fig. 8. Comparison of various compression algorithm extensions



(a) Effects of event stream batch size (b) Effects of event cluster size (c) Effects of adaptive similarity threshold

Fig. 9. Effects of the compressed matching internal parameters

thread-to-core assignments, the rise of false sharing and cache coherency are inevitable due to the recursive nature of BE-Tree processing. The traversal unrolling C-U solves precisely this problem. As shown in Figure 8(a), the throughput has increased by 29X when applying traversal unrolling. Notably, the C algorithm throughput is actually lower than sequential BE-Tree, as shown in Figure 8(a), this finding confirms the claim in [16] that unless false sharing is resolved no performance gain is obtained even with a theoretically highly parallel algorithm [16].

Effects of Parallel Path Processing. Our second optimization is aimed to systemically eliminate the second obstacle, as formulated by *Amdahl's Law*, in harnessing the true benefits of any parallel algorithm. This law states the performance gain of any parallel algorithm is limited by and inversely proportional to the time needed for the sequential fraction of the algorithm. The biggest sequential component of the C-U algorithm is traversal unrolling, which is solved by the proposed parallel path processing algorithm C-U-PP. As demonstrated in Figure 8(a), after parallelizing the traversal unrolling, the throughput is increased by nearly 51X over our base compressed algorithm C.

Effects of Stream Re-orderings. The last two experiments showcase the effectiveness of the parallel compressed algorithm, but they do now show the practicality and wide-applicability of our approach. The compressed algorithm C-U-PP can successfully utilize similarity in an event stream assuming that the stream is sorted in a sense that similar events are near each other (Figure 8(a)). But in practical setting, although there may be overlap among events, however, the events may be in any random order (in fact, all generated workloads except those in Figure 8(a), follow a random order). Therefore, it is essential to efficiently re-order the event stream on-the-fly as achieved by our online stream re-ordering technique (OSR) included in C-U-PP-RE. In Figure 8(b), we vary stream similarity for an unsorted event stream. For C-U-PP-RE, events are buffered in batches of size one thousand

Datasets	Stream Re-ordering	Bitmap Encoding & Compression	Tree Traversal	Leaf Scanning
Unif	2.57%	0.91%	32.08%	63.62%
Zipf	0.36%	0.18%	28.61%	70.64%
Author	2.18%	0.76%	29.53%	66.97%
Title	1.04%	0.22%	23.20%	75.41%

TABLE II
A-PCM MATCHING TIME BREAKDOWN (%)

(a tunable parameter), and re-ordered before passing into the parallel compressed algorithm. The re-ordering is most effective, as expected, when the stream similarity is high, in which the throughput is increased substantially by a factor of up to 38X. Naturally, as stream similarity is reduced, the benefit is also reduced. Thus, when similarity is only 10%, C-U-PP-RE remain dominant and outperforms C-U-PP by 8X.

Effects of Stream Adaptive Processing. To further exploit the efficient online stream re-ordering included in C-U-PP-RE, the proposed adaptive algorithm A-PCM also recognizes when the parallel compressed algorithm C-U-PP-RE is most effective and when the parallel BE-Tree algorithm, Parallel, is most suitable. This process interweaves with our stream re-ordering algorithm, which clusters events based on their similarities. Thus, all clusters below a certain tunable similarity threshold (2 in our case) is processed using Parallel and others are processed by C-U-PP. This adaptive processing of the event stream amounts to the substantial gain of up to 8X, as shown in Figure 8(c). As expected, the A-PCM algorithm is most effective when the variance in the event stream is high, which is observed when stream similarity reaches 0%. However, when the event stream is at 100% similarity (arguably an unlikely practical scenario), then, as expected, the C-U-PP-RE algorithm has slightly higher throughput than the A-PCM algorithm.

Effects of Adaptive Processing Parameters. The three A-PCM parameters that were assumed in the previous experiments are: The event stream batch size (in stream re-ordering), event cluster size (in stream re-ordering), and the similarity threshold. We observe that as the stream batch size increases, our re-ordering algorithm is able to find a better

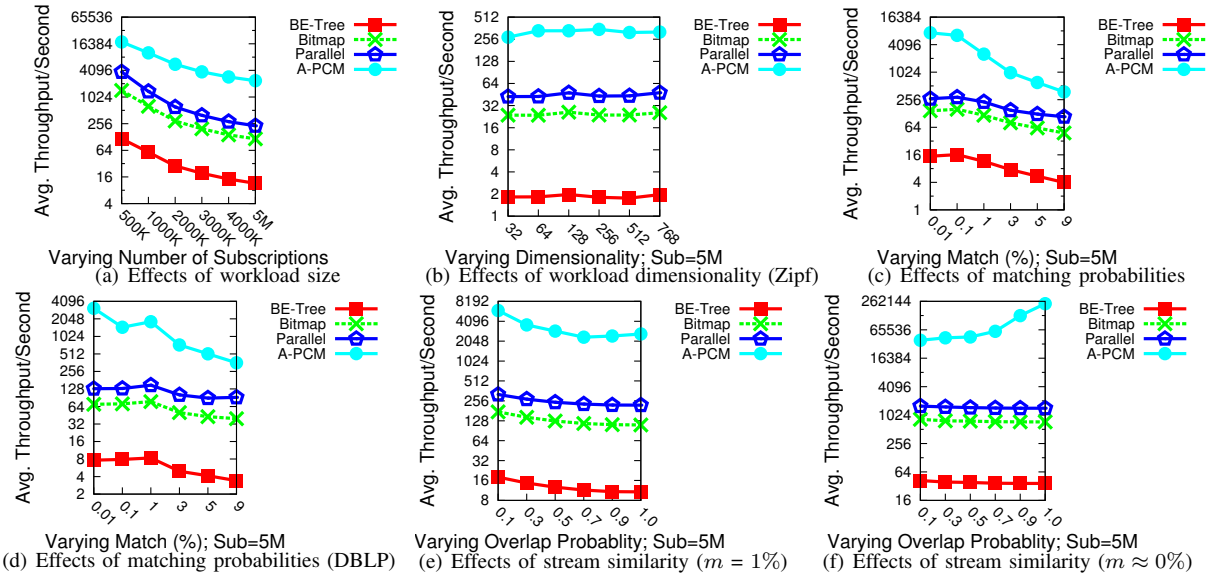


Fig. 10. Comparison of sequential, parallel, and adaptive parallel compressed techniques

clustering, thus, resulting in an improved throughput, as shown in Figure 9(a). In all of our experiments, we fixed our batch size to 1024.

Unlike the effect of the batch size that was rather subtle, we observed that the event cluster size for stream re-ordering is critical in tuning the A-PCM algorithm. As shown in Figure 9(b), as we vary the stream similarity, different values of the event cluster size substantially affect the overall throughput. For example, for the lowest stream similarity, as the event cluster size is decreased from 128 to 8, the throughput is increased by a factor of 9X. This increase in throughput is justified because forcing a large event cluster size over a stream with low similarity threshold results in clusters with many unrelated events; thus, increasing the number of search paths and increasing the false candidate rate. To the contrary, we observed in all our experiments that when the cluster event size is set to the number of available threads (8 in our setting), regardless of the stream similarity threshold, the A-PCM algorithm always achieves the highest throughput. Therefore, simplifying the parameter tuning of the re-ordering technique and demonstrating that our re-ordering technique is robust with respect to the degree of the stream similarity.

The re-ordering technique is robust because when using BE-Tree for clustering events, BE-Tree automatically adapts the cluster size given the workload, and the value of the event cluster size is simply taken as an initial value. This parameter is tuned adaptively based on the size of the discovered clusters. This fact is also evident in Figure 9(b), which shows that the throughput of lower and higher cluster values converges.

The results for varying the similarity threshold parameter are shown in Figure 9(c), which is the reminiscence of the pattern observed in Figure 8(c). The higher the similarity threshold (and the A-PCM algorithm itself) is, the better suited it is for streams with a lower overlap among events. By using higher a similarity threshold, we ensure that only events with predicates on many common attributes are compressed together. Overall, we observe that setting similarity threshold at around 2 is most robust to workloads with different degrees of stream similarity. Thus, in all of our experiments, we choose similarity threshold of 2.

Matching Time Breakdown. We conclude our micro experiments by analyzing the A-PCM matching time for our default datasets with a 1% matching probability and a 70% stream similarity, as shown in Table II. The overall time is broken down into three main components:[11] Stream re-ordering, parallel bitmap encoding and compression (Stages 2-3 in Figure 5), parallel tree traversal (Stage 4 in Figure 5), and parallel leaf scanning (Stage 5 in Figure 5). The key finding is that the time for stream re-ordering, parallel bitmap encoding, and compression is negligible, and on average the leaf scanning takes twice as long as the tree traversal. It is noteworth that the ratio between tree traversal and leaf scanning may vary depending on the workload matching probability.

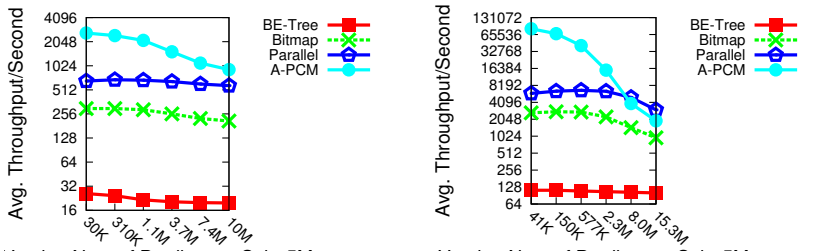
C. Macro Experiments

After incrementally showing the effect of each of the proposed optimizations over the (adaptive) parallel compressed algorithm, we conclude that the A-PCM algorithm is most effective. Next, we compare A-PCM with the best known sequential matching algorithm, BE-Tree [21], and its straight forward parallel counterpart, Parallel; these simple matching algorithms parallelization were also suggested in [7].

Effects of Workload Size. As we increase the number of subscriptions from 0.5M to 5M, the gap between the A-PCM algorithm and BE-Tree substantially increases, 153X and 217X increase in throughput when having only eight parallel threads. The bitmap-based encoding of BE-Tree also results in an improvement over BE-Tree (e.g., up to 12X). Most striking is the improvement of our A-PCM over Parallel, an increase of up to 10X. The gap further widens when scaling the workload size. Finally, we observe that the A-PCM algorithm can sustain a rate of up to 17,947 events/second when there are as many as half a million subscriptions, as shown in Figure 10(a).

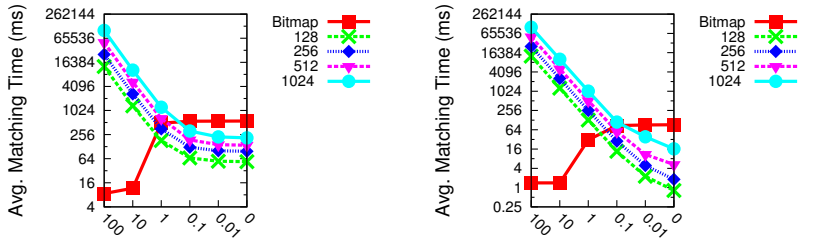
Effects of Workload Dimensionality. Another key workload characteristic is dimensionality (Figure 10(b)). In fact,

[11] We have discarded the execution time for statistics collection and other minor components. Therefore, the sum of these three components are slightly less than 100%.



(a) Effects of num of predicates (m = 1%) (b) Effects of num of predicates ($m \approx 0\%$)

Fig. 11. Comparison of sequential, parallel, and adaptive parallel compressed techniques



(a) Effects of event delays (m = 1%) (b) Effects of event delays ($m \approx 0\%$)

Fig. 12. Latency comparison of sequential (Bitmap) and adaptive parallel compressed techniques (A-PCM) with different batch sizes from 128 to 1024 events (for brevity, we omit the name A-PCM in the legend, and we distinguish among different A-PCM run based on the batch sizes)

the curse of dimensionality is the most challenging aspect of our problem. Similar to BE-Tree, the A-PCM algorithm copes with a space dimensionality in the hundreds, which is partly due to the effective online stream re-ordering that is robust with respect to both stream incompleteness and high-dimensionality. The A-PCM algorithm continues to outperform both BE-Tree and Parallelalgorithms by up to 162X and 7.7X, respectively.

Effects of Matching Probability. Another key distinguishing workload property is the matching probability, which assess the applicability of matching algorithms for a wide-range of applications faring anywhere from only a few matches to hundreds of thousands of matches per event. In all experiments for both synthetic and real workloads, The A-PCM algorithm remain dominant regardless of matching probability and outperformes the Parallel algorithm by up to 27X and 3.9X, when the matching probability is 0.01% and 9%, respectively, as shown in Figures 10(c)-10(d). Furthermore, the A-PCM algorithm substantially improves over sequential BE-Tree by a factor of 503X.

Effects of Stream Similarity. This is one of our most important experiment, which establishes that the A-PCM algorithm can take full advantage of stream similarity even when the stream is unsorted. Thus, the A-PCM algorithm re-orders and identifies similar events, compresses similar events, and process the compressed stream in parallel. In Figure 10(e), as we vary stream similarity (starting with the initial matching probability of 1%), our A-PCM algorithm outperformes BE-Tree and the Parallel algorithm by 330X and 19X, respectively.

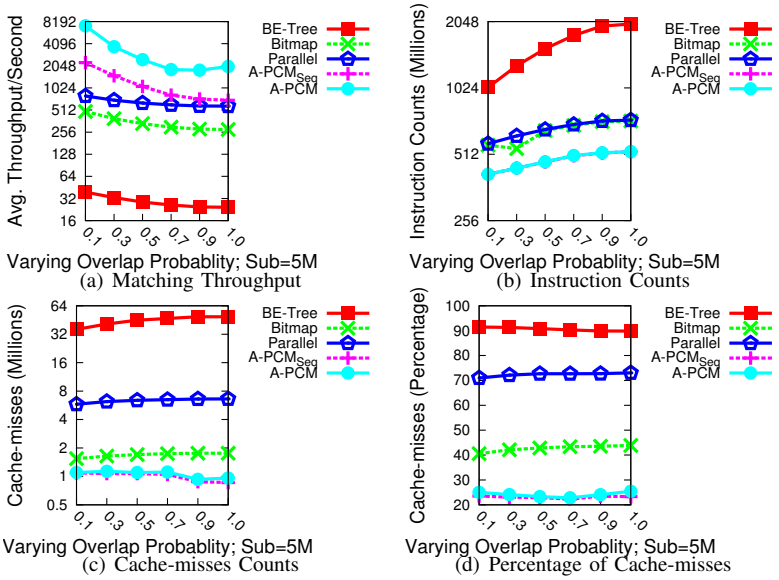
From an experimentation point of view, as discussed previously, a side-effect of varying stream similarity is the loss of control over the event matching probability, which explains why for a lower stream similarity, the throughput of all algorithms are higher. In order to keep the matching probability constant, we repeat the same experiments but without controlling the matching probability, cf. Figure 10(f). As expected, we observe that varying the stream similarity has no effect on the BE-Tree, Bitmap, and Parallel algorithms, but A-PCM is substantially improved by up to 5.9X for an increasing degree of stream similarity. As a result, A-PCM can sustain an event rate of up to 233,863 events/second while the Parallel algorithm is saturated with only 1,429 events/second, over two orders of magnitude slower.

Effects of Number of Distinct Predicates. In all the above experiments, we kept the average number of distinct predicates below a million. Next, we present the result of increasing the number of distinct predicates to tens of millions, such that even a single bitmap-based encoding will not fit in the processor caches entirely. In Figure 11, we capture the effects of scaling the number of distinct predicates from tens of thousands to tens of millions. As expected, our A-PCM algorithm is most effective when the bitmap-based event encoding of all compressed events fits into the processor cache, to be more precise, as long as the frequently accessed parts of bitmap-based encoding are cache-residents.

Effects of Event Batching on Latency. Our second most important experiments is focused on average matching latency. Hitherto, we demonstrated that A-PCM could substantially improve the matching throughput when batching events, but we have ignored the effects of batching events on the latency.

We now consider a scenario in which we decrease the average delay latency between events (i.e., increasing the event rate) from 100ms to 0ms, as shown in Figure 12. For this experiment, we focus our study only to the fastest sequential (Bitmap) and parallel (A-PCM) algorithms observed in our evaluation. Furthermore, we construct the best possible scenario for the latency computation of our Bitmap technique, in which the experiment is repeated for every batch of 128 events implying that after every 128 events, we reset all counters and starts afresh.[12] This method is specially advantageous (substantially underestimating the true latency) for a

[12]Following the same methodology, we reset counters for A-PCM after processing a batch of X events, where X is chosen from 128, 256, 512, or 1024 events.



(a) Matching Throughput (b) Instruction Counts (c) Cache-misses Counts (d) Percentage of Cache-misses

Fig. 13. Low-level CPU performance counter analysis

high-throughput event stream when using Bitmap because we ignore the fact that over time, the backlog of unprocessed events (pending queued events) continues to grow, which in practice will further increases the average matching latency. But even in such a biased experiment setting towards our Bitmap technique, we establish the superiority of the A-PCM algorithm.

We draw two key observations. First, as long as the average matching latency is smaller than event delays, then clearly any sequential algorithm will outperform, with respect to the average latency, any parallel algorithm that relies on batching because in the time that requires to batch a set of events, the entire matching computation could have been completed. This breakaway point in our experiments is when the average delay between events are smaller than 1-10ms (i.e., when the event stream rate is below 1000 events per second). Second, most importantly, when the average delay between event is about a factor of 10-100 smaller than matching time, then we observe that A-PCM not only substantially improve the throughput, but in fact, it reduces the overall matching latency compared to our fastest sequential sequential algorithm. As shown in Figure 12, when the stream rate is higher than 10,000 events/second, then A-PCM exhibits a lower matching latency compared to Bitmap sequential algorithm. Thereby, making A-PCM a robust matching algorithm with respect to both throughput and latency dimensions for high-throughput event stream.

In this work, we focused on uniform average delays between events and rely on a count-based batched size. However, if the event delay follows a skewed distribution, then a simple count-based semantics may not suffice, but our proposed adaptive matching scheme can easily be enriched by detecting event delay anomalies that are longer than the expected delay, and the batching can either follow a count-based or a time-based (or even a priority-based) semantics based on the observed/expected event delay.

Effects of CPU Performance Counter. To substantiate our analytical claims that our proposed A-PCM is both cache-friendly (reducing cache-misses) and algorithmically efficient (reducing the number of instructions), we extract performance counter by capturing CPU low-level hardware events using the Linux profiler tool called perf.[13] In Figure 13, for the same experiment, in which we vary the stream similarity, we present the average matching throughput (in Figure 13(a)), the raw number of instructions for an average run of our experiment (in Figure 13(b)), the raw number of cache-misses for an average run of our experiment (in Figure 13(c)), and the percentage of cache-misses for an average run of our experiment (in Figure 13(d)). To distinguish the benefits of parallelization and algorithmic efficiency of our A-PCM, we devise two versions of A-PCM: the parallel version (A-PCM) and the sequential version (A-PCM$_{Seq}$).

We observe that our compressed matching algorithm indeed algorithmically more efficient, i.e., partly owing to reduction of raw number of executed instructions by a factor of 0.5 in Figure 13(b), because A-PCM$_{Seq}$ throughput outperforms both sequential Bitmap and Parallel algorithms, as shown in Figure 13(a). In addition to algorithmic superiority of A-PCM and A-PCM$_{Seq}$, our compression matching algorithm exhibits a better cache-locality, which is reflected in the reduced number of caches-misses in Figure 13(c). In both A-PCM and A-PCM$_{Seq}$ (as expected, they exhibit almost an identical number of cache-misses), the number of cache-misses compared to Bitmap and Parallel are reduced by a factor of 2 and 8, respectively.[14] Similarly, the percentage of cache-misses and inferred total number of cache-reference requests in A-PCM and A-PCM$_{Seq}$ are smaller, as shown in Figure 13(d). Thus, the benefits of our compressed matching algorithm to reduce the matching computation are two-fold: an effective parallelization algorithm and a cache-friendly design.

VIII. Related Work

The problem related to indexing Boolean expressions has been studied in the database (e.g., [10]) and the publish/subscribe (e.g., [2], [6], [4], [25], [21], [7], [22], [20])

[13]For gathering the performance counter we used a newer machine with one Quad-core Intel Xeon W3565 processor running at 3.20GHz with shared 8MB L3 cache size.

[14]For Parallel algorithm, many cache-misses and the subsequent memory-requests may be issued by CPU in parallel; thus, the overall delay of cache-misses could be smaller.

communities are different in two important ways. First, the database indexing solves the reverse problem: in the database context, querying means finding the relevant tuples (events) for a given query (subscription), but in event processing context, matching (through indexing) means finding the relevant subscriptions (queries) for a given event (tuple). Second, publish/subscribe matching algorithms overlook both parallel event matching and event stream re-ordering, which is central in exploiting the exponential growth trend in the number of cores of modern hardware [14], [18].

A closer look manifests that counting-based methods primarily aim to minimize the number of predicate evaluations by constructing an inverted index over all unique predicates resulting in a clustering. The two most efficient counting-based algorithms are Propagation [6], a key-based method, and the k-index [25], a non-key-based method. Likewise, tree-based methods are primarily designed to reduce predicate evaluations and to recursively divide search space by eliminating expressions on encountering unsatisfiable predicates. The first major tree-based approach, Gryphon, is a static, a non-key based method [2], which is shown to be effective only for equality predicates [2]. The latest tree-based structure is BE-Tree, a key-based approach, that introduces a two-phase space-cutting abstraction for supporting workload changes and overcoming the curse of dimensionality [21], [20]. BE-Tree leverages an effective clustering that is inherently designed for finite and discrete domains. Furthermore, BE-Tree is a dynamic structure and is proven to outperform all existing techniques [21], [20].

Despite BE-Tree's effectiveness, it provides no guarantee for exactly-once predicate evaluation. More importantly, there are no attempts to parallelize BE-Tree, which is a non-regular, tree-based structure that suffers from the well-known pointer-chasing and tree traversal issues that inevitably incur cache-misses. This problem is further amplified as the number of parallel tree traversal are increased, resulting in a higher number of cache-misses. In addition, BE-Tree fails to algorithmically exploit the large number of cores and large shared caches prevailing in multi-core architectures [18]. In this work, we present novel adaptive parallel compressed event matching (A-PCM) and online stream re-ordering (OSR) algorithms that exploit all these new hardware properties.

Finally, online sorting and stream re-ordering in database indexing and in the storage context has relied heavily on locality-sensitive hashing [13] in order to cope with the curse of dimensionality (e.g., [11], [17], [9]). In general, the main shortcoming of locality-sensitive hashing, which limits its applicability in practical settings, is the complexity and uncertainty of tuning locality-sensitive hashing parameters [5]. These parameters are highly data dependent, which complicates the performance tuning approaches [5]. In contrast, in this work, we present our OSR algorithm for high-dimensional data by leveraging BE-Tree [21] that not only eliminates the parameter tuning challenge, but also copes with *incomplete event data* in the stream and aids to identify *stream heterogeneity*.

IX. Conclusions

In this paper, we studied the problem of parallel event processing. Particularly, we enhanced and parallelized an existing state-of-the-art matching algorithm by introducing a novel event stream compression algorithm enabled through a bitmap-based event encoding. Furthermore, we developed an efficient online event stream re-ordering (OSR) approach to exploit the full potential of our adaptive parallel compressed matching algorithm (A-PCM). Our extensive evaluation demonstrates the effectiveness of the proposed A-PCM algorithm that outperformed both sequential and naive parallel algorithms by a factor of up to 503X.

References

[1] R. Agrawal, A. Ailamaki, P. A. Bernstein, et al. The claremont report on database research. *SIGMOD Rec.'08*.

[2] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *PODC'99*.

[3] J. Daily. There's millions in those microseconds. *The Globe & Mail, 29/1/10*.

[4] Y. Diao, P. Fischer, M. Franklin, and R. To. Yfilter: Efficient and scalable filtering of XML documents. In *ICDE'02*.

[5] W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li. Modeling lsh for performance tuning. CIKM'08.

[6] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for fast pub/sub systems. *SIGMOD'01*.

[7] A. Farroukh, E. Ferzli, N. Tajuddin, and H.-A. Jacobsen. Parallel event processing for content-based publish/subscribe systems. DEBS'09.

[8] A. Farroukh, M. Sadoghi, and H.-A. Jacobsen. Towards vulnerability-based intrusion detection with event processing. In *DEBS'11*.

[9] F. Fusco, M. P. Stoecklin, and M. Vlachos. Net-fli: On-the-fly compression, archiving and indexing of streaming network traffic. *PVLDB'10*.

[10] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.'98*.

[11] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. VLDB'99.

[12] K. Heires. Budgeting for latency: If I shave a microsecond, will I see a 10x profit? *Securities Industry'10*.

[13] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. STOC'98.

[14] C. Johnson and J. Welser. Future processors: flexible and modular. CODES+ISSS'05.

[15] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. EDBT'09.

[16] T. Liu and E. D. Berger. Sheriff: precise detection and automatic mitigation of false sharing. OOPSLA'11.

[17] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. VLDB'07.

[18] J. Parkhurst, J. Darringer, and B. Grundmann. From single core to multi-core: preparing for a new exponential. ICCAD'06.

[19] M. Sadoghi, I. Burcea, and H.-A. Jacobsen. GPX-Matcher: a generic Boolean predicate-based XPath expression matcher. In *EDBT'11*.

[20] M. Sadoghi and H.-A. Jacobsen. Analysis of Boolean expressions indexing techniques. In *TODS'13*.

[21] M. Sadoghi and H.-A. Jacobsen. BE-Tree: An index structure to efficiently match Boolean expressions over high-dimensional discrete space. In *SIGMOD'11*.

[22] M. Sadoghi and H.-A. Jacobsen. Relevance matters: Capitalize on less (top-k matching in publish/subscribe). In *ICDE'12*.

[23] M. Sadoghi, M. Labrecque, H. Singh, W. Shum, and H.-A. Jacobsen. Efficient event processing through reconfigurable hardware for algorithmic trading. In *VLDB'10*.

[24] D. Srivastava, L. Golab, R. Greer, T. Johnson, J. Seidel, V. Shkapenyuk, O. Spatscheck, and J. Yates. Enabling real time data analysis. *PVLDB'10*.

[25] S. Whang, C. Brower, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, R. Yerneni, and H. Garcia-Molina. Indexing Boolean expressions. In *VLDB'09*.