Optimization of Massive Pattern Queries by Dynamic Configuration Morphing

Nikolay Laptev #1, Carlo Zaniolo #2

# *University of California, Los Angeles*
California, USA

1 nlaptev@cs.ucla.edu

2 zaniolo@cs.ucla.edu

***Abstract*—Complex pattern queries play a critical role in many applications that must efficiently search databases and data streams. Current techniques support the search for multiple patterns using deterministic or non-deterministic automata. In practice however, the static pattern representation does not fully utilize available system resources, subsequently suffering from poor performance. Therefore a low overhead auto-reconfigurable automaton is needed that optimizes pattern matching performance. In this paper, we propose a dynamic system that entails the efficient and reliable evaluation of a very large number of pattern queries on a resource constrained system under changing stress-load. Our system prototype, Morpheus, pre-computes several query pattern representations, named templates, which are then morphed into a required form during run-time. Morpheus uses templates to speed up dynamic automaton reconfiguration. Results from empirical studies confirm the benefits of our approach, with three orders of magnitude improvement achieved in the overall pattern matching performance with the help of dynamic reconfiguration. This is accomplished only with a modest increase in amortized memory usage.**

I. Introduction

In this paper, we improve the performance of pattern matching by changing the pattern representation dynamically according to available resources. Previously, pattern matching was done using a static automaton, however because 'one size does not fit all', we propose a dynamic model that adopts to its resource environment thus dramatically improving the performance of pattern queries. Indeed, the optimization of pattern queries represents a problem of great research interest and practical importance due to the many applications of such queries.

In the financial sector, pattern queries (PQs), that run continuously over the data stream, are used to detect fleeting opportunities by monitoring trends. Informally we define a pattern query as a regular expression (RE) with optional predicates assigned to each symbol. Note that if a pattern query q only contains equality predicates, then q is exactly equivalent to a regular expression [22]. In network management continuous pattern queries can be utilized to monitor online traffic and detect anomalies (e.g., link congestion) and their cause (e.g., hardware failure, denial-of-service attack). Because a set of patterns is usually large (a set of HTTP patterns alone takes a few GB [24]) and because we look at applications where processing of data needs to be done in realtime, dynamically adjusting the pattern matching system to take advantage of all hardware resources without sacrificing the overall performance is of great importance. The interest in pattern matching is further underscored by the large series of increasingly powerful and sophisticated query languages proposed for defining and evaluating patterns. A very incomplete list includes SQL-TS supporting regular expressions and backtrack optimization [25], SASE+ [9] supporting powerful Kleene-closure queries, and K*SQL [20] supporting the nested word model and queries on XML software logs, and genomics.

In these pattern languages, regular Kleene-star expressions are used to characterize patterns which are then implemented using a finite state automaton (FSA). The type of FSA constructed and thus the resulting query performance much depend on the available system resources. Thus in this paper, we address the problem of dynamically changing the underlying automata in response to changing system resources, in order to optimize pattern matching performance.

High level of performance is, for instance, required by long running pattern queries that perform decision support and data mining against massive databases or bursty data streams that put a strain on the already limited resources such as processing power, network bandwidth and battery capacitance. The problem is exacerbated when pattern query evaluation is performed using a multi-core or a cloud environment where frequent changes in the availability of workstation resources and the overall workstation load are common. Thus in this work we make the first steps towards designing a dynamic pattern matching system, called *Morpheus* that adopts to the available system resources. In fact, Morpheus automatically reconfigures its pattern representation, based on the finite state automata, according to the available workstation resources.

Further evidence for the need of dynamic systems is provided by the introduction of Adaptive Query Processing (AQP)[14]. With AQP run-time statistics are recorded and optimizations on the query, including selection and projection optimizations, are performed [10] based on the current runtime statistics. AQP is especially popular in data streams processing where the data and system resources change frequently. Our work leaves the query unchanged and instead modifies the *system*, which in our case is the pattern automaton, for efficient pattern matching execution. It is common for a multicore system to experience a mixture of predictable and sporadic changes in the workload thereby creating a

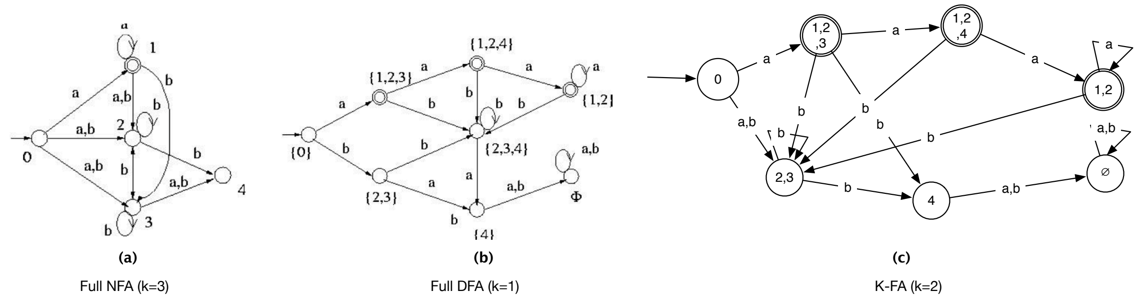

Fig. 1. Bounded FSA example for a regular expression

necessity for a dynamic system optimization. To address these conditions, "resource-aware" systems are beginning to emerge [6], [21] that provide a varying quality of service depending on the available resources. The closest work to ours relies on convoluted tactics to cache frequently accessed program instructions to help with the dynamic workload [17]. This approach, however, is not effective when workloads are unpredictable. Other previous works mainly focused on tuning the query representation given a static environment. For example authors in [8] employ a non-deterministic finite state automaton (NFA) to represent a set of XPath patterns to filter XML documents. Note that XML queries will benefit from our approach, but so will many other query languages (e.g., K*SQL), for which FSA provides a powerful execution model as it has been widely recognized in previous literature [8], [29], [22]. Other methods [15] explore the trade-off between a deterministic finite automaton (DFA) pattern representation and an NFA to find the middle ground that achieves desired performance while minimizing memory usage. The previous work discussed, exclusively consider a static environment. This is a serious limitation given the bursty nature of data streams, and the real-time response requirement of many applications.

Dynamic reconfiguration methods present several problems in terms of reconfiguration overhead (time and space). Because reconfiguration is done during run-time, the overhead is part of the run-time cost and needs to be minimized. The dynamic reconfiguration problem is further complicated by the search process that must select the best system configuration. This is often difficult because resource and performance estimations of a potential configuration are required. One way to address the issues with dynamic reconfiguration is by using a naive method. The obvious, but deficient solution, is to let the user reconfigure manually the representation of pattern queries. Unfortunately, the user may know *what* she wants but not know *how* to achieve it. The assumption that the user understands the low level system details including the available resources and is able to adapt a query representation to environment changes is clearly unreasonable. Thus in our approach to deal with the reconfiguration overhead we use fast approximation methods that are very effective in practice and significantly improve the state of the art in dynamic pattern matching.

For completeness we briefly define a pattern matching problem. Informally a pattern matching problem seeks to detect all sub-sequences of input that match a given set of patterns. More formally, given a potentially infinite stream I consisting of symbols drawn from the alphabet Σ and given $P = \{\rho_1, ..., \rho_m\}$ which is a set of patterns over Σ then the pattern matching problem involves detecting all subsequences of I matching $\rho_i \in P$.

Traditional pattern matching approaches generally construct one finite state machine (a deterministic or a nondeterministic finite automaton (DFA or NFA respectively)) for all patterns to be matched. A DFA is a quintuple $M(S, s_0, \sigma, \Sigma, F)$ where S is a finite set of states, s_0 is an initial state, $\sigma : S \times \Sigma \to S$ is a transition function, Σ is the alphabet and $F \subseteq S$ is a set of accepting states. NFA is defined similarly except that the transition function σ may return multiple states, thus multiple states may be active for input i. As compared to a DFA, an NFA can represent a pattern using storage that is on the same order as the number of characters present in the pattern, however the processing cost of the NFA is expensive due to a potentially large number of concurrently active states. While a DFA is largely immune from the processing cost issues plaguing an NFA, representing a patterns with a DFA may lead to a state explosion and thus to a prohibitively large automaton. Therefore, the number of concurrently active states is the primary criterion that determines both the processing bandwidth and the space requirements of the automaton. Thus, by using a middle-ground between an NFA and a DFA to represent a set of patterns in a dynamic environment we could potentially achieve the speed of a DFA and the space efficiency of NFA during run-time.

In order to dynamically adjust the automaton configuration our system prototype needs to be able to efficiently switch between a non-deterministic finite automaton (NFA), a determinism finite automaton (DFA) and an automaton that has k active states. There are standard techniques to convert a pattern (RE) into an NFA or a DFA [12]. However no published research has so far considered the important problem of dynamically adjusting an automaton with a bounded number of concurrently active states. Thus we define a *K-Finite Automaton* (K-FA) to be a finite automaton where there are at most k concurrently active states at any time.

Figure 1 shows a motivating example of using an automaton with a bounded number of concurrently active states. Figure 1 (a) shows an NFA that accepts pattern p and has up to three concurrently active states. Figure 1 (b) shows a DFA that accepts p and has only one active state at any time. Figure 1 (c) presents a *K-FA* that accepts p where there is only a bounded number of $k = 2$ concurrently active states. Now, if $D(i, j)$ is the difference in the number of states between automaton i and j, we have that $D((b),(c)) < D((a),(c))$. Therefore if resources change and the automaton in Figure 1 (c) was the desired configuration, template in Figure (b) would be morphed into (a) as this morphing would require least amount of operations. Similar analysis will be used when deciding how dynamic reconfiguration should be done.

Morpheus performs dynamic system reconfiguration to reconfigure the underlying pattern matching automaton based on the current system resource availability. Morpheus precomputes *critical* query configurations that will be used as *templates* which will be morphed into the desired configurations during run time. A template is a K-FA pattern representation. By using templates the overhead of dynamic reconfiguration is significantly reduced as shown in the experimental section. The *critical* templates to precompute are selected such that the average expected reconfiguration time is minimized. Empirical evidence also shows that the templates virtually do not increase the amortized memory cost while greatly increasing the throughput when compared to static solutions. Throughout this paper we refer to a *configuration* as a representation of an automaton m at time t given resources R_{it} where $i \in$ *Resources (R)*.

Morpheus performs efficient dynamic reconfiguration in three stages: (i) template computation, (ii) run-time optimization and (iii) template adjustment. In (i) the templates are computed such that the expected dynamic reconfiguration (morphing) time is minimized. The optimal number of templates is determined based on the estimated amortized memory cost and the template update cost. In (ii) resource estimation is performed when searching for a potential configuration. Furthermore the run-time stage identifies the time when it is feasible and desirable to perform the automaton reconfiguration by analysing the cost/benefit of the reconfiguration. Steps necessary to perform the reconfiguration are also carried out in (ii). Stage (iii) updates the precomputed templates based on the template usage statistics. All three of the dynamic reconfiguration stages are seamlessly integrated into Morpheus.

This paper makes the following contributions:

1) A system prototype, termed Morpheus, is presented that uses a novel way of performing dynamic reconfiguration of pattern automata via morphing of precomputed templates to speed up pattern matching in a dynamic environment.
2) Linear optimization techniques are discussed that are used for dynamic reconfiguration of pattern automaton that increase/decrease automaton parallelism thus adjusting to the current system resources.
3) An Empirical study that evaluates Morpheus over various workloads is presented.

The rest of the paper is organized as follows: in section II we present an overview of the architecture of Morpheus. Section III is devoted to discussion of the details of each of the components of our system. Section IV presents a performance evaluation of our system. Finally, section VI summarizes the paper and presents our future work.

II. System Overview

The main components of Morpheus are briefly described next. The focus of Morpheus is on improving the performance of pattern matching and the main way this is accomplished is by reconfiguring the pattern representation based on the available system resources. Dynamic reconfiguration can negatively impact matching performance, however templates help mitigate the problem by precomputing several configuration in advance that can be used as starting points for achieving the final configuration. Precomputed templates are k-FAs with $k > 1$ and have a small memory foot-print as shown by empirical evidence in Section IV-D.

Morpheus as an input takes an *NFA* representing a collection of patterns. The inputted NFA is then used by the reconfiguration engine to precompute templates. The key components presented in Figure 2 can be summarized as follows:

1) *Reconfiguration Engine:* The reconfiguration engine is responsible for pre-computing and adjusting the automata on the basis of the current workload. Specifically, Morpheus uses linear optimization techniques to find (i) the number of precomputed configurations, (ii) the set of operations needed to transition between configurations and (iii) the time when to update the precomputed configurations. The linear optimization techniques used provide near optimal solutions. The automaton access probabilities are also taken into account during reconfiguration. Furthermore the reconfiguration engine evolves over time because both the resources and the access probabilities change over time.
2) *Resource Monitor:* Morpheus includes an automated statistics collection tool that, with a negligible overhead, captures data from the OS (CPU and RAM usage) and from the currently active automaton (state access probabilities). The CPU usage reported by the Linux kernel is expressed as a percentage of one CPU core. Initially we assume a uniform random distribution of the input symbols (i.e., an equal access probability of all states). During run-time, however, we consider a trace driven probability distribution of various input symbols. With these traces we can more accurately determine the nondeterminism of a particular configuration. The collected statistics are utilized to decide *when* to reconfigure and into *what* reconfiguration.
3) *Resource Usage Estimator:* When choosing an automaton M from the search space we must estimate its resource usage to insure that the available resources are properly utilized. Based on our experiments the average

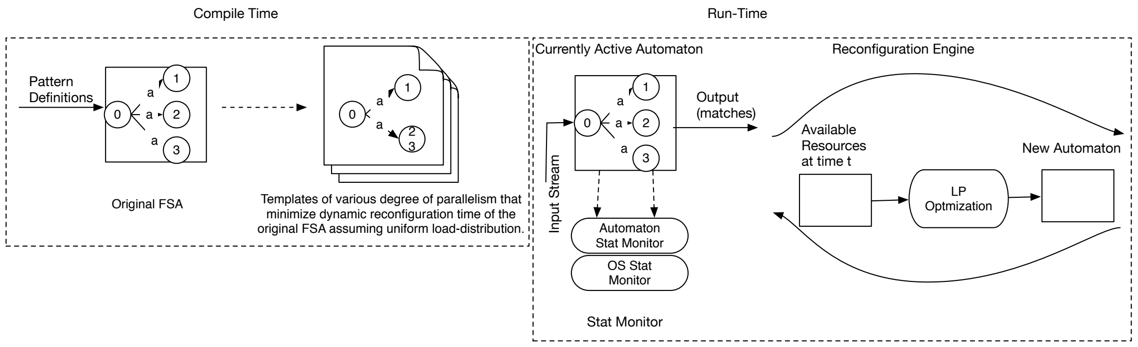
Compile Time
Run-Time
Pattern Definitions
Original FSA
Templates of various degree of parallelism that minimize dynamic reconfiguration time of the original FSA assuming uniform load-distribution.
Currently Active Automaton
Reconfiguration Engine
Output (matches)
Input Stream
Available Resources at time t
New Automaton
LP Optmization
Automaton Stat Monitor
OS Stat Monitor
Stat Monitor

Fig. 2. Architecture of Morpheus

CPU load is linearly proportional to the average non-determinism of approximately the first 5% of the states of the automaton. Therefore doubling non-determinism of the first 5% of the automaton states doubles the CPU usage requirement. The intuitive explanation for the 5% figure is that on average only the first 5% of the pattern is matched by the input. This crude estimate produces good results as discussed in the performance evaluation section. The memory usage of M is again estimated by a linear approximation and as seen in Figure IV this produces good results.

Next we describe the details of the reconfiguration engine.

III. Reconfiguration Engine Details

The reconfiguration engine can be decomposed into three stages: *pre-configuration*, *execution* and *update*. In the first phase several automaton configurations are pre-computed to minimize the dynamic re-configuration time when system resources or workload varies. Then in the execution phase the pre-computed configurations are used as *templates* to *morph* the pre-configuration into a desired state. Finally during the last phase the pre-computed configurations are updated based on the historical statistics gathered. The above phases are discussed in detail next.

A. Pre-Configuration

When pre-computing templates, the assumption is made that all operations are done on a single FSA, (i.e. no clustering is performed). Breaking up the FSA into disjoint clusters may improve memory usage however for the sake of simplicity we only deal with a single automaton. Nevertheless clustering is important especially when dynamically adding patterns into the automaton. While Morphesus also supports simple pattern clustering method using distance metrics such as MDL [5], these are orthogonal to the focus of this paper. Thus we will assume that a single NFA N representing a set of patterns is given as input. Given N and a set of resources R representing system constraints (e.g. memory, cpu) the pre-configuration problem consists of computing a set of automata *derived* from N such that the average *conversion* time between any two automata is minimized and all resource constraints are satisfied. Therefore, we will derive from N a new automaton M that accepts the same language but has a different level of non-determinism, k, where k denotes the average number of states that are concurrently active per input. A DFA has $k = 1$ and an NFA has no upper bound on k. Thus we define *K-FA* (K-Finite Automaton) as automaton that has a bounded number of concurrently active states equal to k.

$$\begin{aligned} minimize: & \\ & max \sum d_i \\ & s.t. \sum m_{ij} x_i \leq R_j \\ & x_i \in \{0, 1\} \end{aligned} \tag{1}$$

More formally, the pre-configuration problem can be formulated as shown in Equation 1. Variable d_i is defined as the minimum *distance* to automaton i from any other automaton and overall we want to minimize the maximum *distance* to any automaton. The *distance* refers to the difference in the number of states between the original and the derived automaton. Variable d_i is directly proportional to the number of operations (*merge* or *split*) needed to be performed to derive automaton i. The formalization of these operations is discussed in Section III-B. The estimation of the size of various automata is presented later this section. m_{ij} refers to the amount of resource j that automaton i uses, where i varies from k to 1 and where k is initially set to be the average number of the concurrently active states in N.

The constraints in Equation 1 guarantee the feasibility of the solution. Because each x_i can only vary between 0 and 1, and because the sum of all $m_{i,j}$ for a given i must be less than or equal to R_j, we guarantee that the combined load imposed on the system will not exceed what is available; this avoids saturation and overcommitment of the system resources. The *resource constraints* include the CPU and the RAM because these were the most constrained resources in the real-world scenarios we explored. Extending a set of constraints, however, to include other resources (e.g., network, disk space) is straight forward. Furthermore note that the goal function in Equation 1 can also depends on the time t as discussed in Section III-B. The feasible solution imposed by the constraints may be fractional and must be rounded as discussed in Section III-D.

Pattern Type	Example	# of states
Strings of length k	^abcd	$k+1$
Wildcards	ab.*cd	$k+1$
Patterns with ^, a wildcard and a length restriction j	^ab.{j+}cd	$O(kj)$
Patterns with ^, overlapping with prefix and a length restriction j	^a+[a-z]—{j}d	$O(k+j^2)$
Patterns with length restriction j, where prefix overlaps with a wildcard or a set of characters	.*ab.{j}cd	$O(k+2^j)$

TABLE I
SIZE ESTIMATES OF DIFFERENT TYPES OF PATTERNS

	Processing complexity	Storage cost
NFA	$O(n^2m)$	$O(nm)$
DFA	O(1)	$O(\sum nm)$
K-FA	O(k)	$O(\frac{1}{k}\sum nm)$

TABLE II
CLASSIFICATION OF AUTOMATA WHERE n REPRESENTS THE AVERAGE LENGTH OF A PATTERN AND m REPRESENTS THE NUMBER OF PATTERNS

The number of pre-computed templates also affects the overall dynamic reconfiguration performance. Recall that templates are used as a way to speed up reconfiguration by picking a starting point to reconfigure from that will result in the desired configuration in the least amount of time. Section IV-D demonstrates that the number of precomputed template configurations, n, should be such that the benefit (average time to perform dynamic reconfiguration) is equal to the cost (the sum of the average search and the update times of the templates). Automatically determining n is important in decreasing the overhead of Morpheus.

Estimating the number of operations: In Equation 1, the number of operations (d_i) is measured in terms of the difference in the number of states between two automata. To estimate the number states (size) of the *K-FA* for some k we use linear fitting between the size of an NFA N and the size of the corresponding DFA D. Thus given a desired k we can quickly estimate the size of the *K-FA* without explicitly deriving it. The number of states of N can be accurately computed, because N is given as the input. The number of states in D can be estimated based on the average nondeterminism per state of the automaton and on the underlying pattern characteristics. The summary of the memory and processing complexities of different types of automata, including *K-FA* is presented in Tables II and I [29] where n is the length of the regular expression and m is the number of regular expressions. Given the initial pattern set we can determine the class of N from Table I, based on which the size of D is computed. To estimate the sizes of subsequent automata a linear model is constructed given two points $|D|$ and $|N|$. When the exact sizes of different $K-FA$ automaton are computed subsequently, Lagrange polynomial is used to adjust our size estimation model into a more accurate, polynomial, version.

Estimating non-determinism: It was shown how to estimate the memory resource requirements of K-FA given k. Next we demonstrate how non-determinism of the K-FA is estimated. Estimating the non-determinism of the K-FA is done by computing the average nondeterminism per state of the automaton as is shown in Equation 2.

$$\frac{\sum_{states} P(i) \times a}{|s_n|} \tag{2}$$

$P(i)$ is the probability of executing a transition with input i, and a is the number of states that is activated when input i is processed. $|s_n|$ is the total size of the input. In our experiments $|s_n|$ is equal to the length of a random string that is accepted by the current automaton. Note that initially all symbols in alphabet σ occur with equal probability, however throughout execution $P(i)$s can be updated to $P(i)'$. This presents a problem because when $P(i)$ is updated, the nondeterminism of K-FA can increase/decrease thus violating the guarantee of our automaton that at most k states are simultaneously active. When this occurs a set of merge/split operations needs to be performed to decrease/increase the non-determinism. In our prototype the threshold α determines the delta such that if $\alpha < |P(i) - P(i)'|$ then the update is performed.

B. Run-Time

During run-time three main steps are performed: (i) decide whether a reconfiguration is needed (ii) when a reconfiguration is needed, find the most applicable configuration from the search space and (iii) perform the necessary merge and split operations to achieve the desired configuration. In this section we give a detailed overview of these three steps.

A *reconfiguration* is the process of applying a set of *operations* to an automaton to make it more or less deterministic depending on the available system resources. More determinism is beneficial for systems with high available memory but low processing availability. Less determinism is attractive for systems with low memory resources and high processing availability. Valid *operations* that can be applied to an automaton include a *merge* operation and a *split* operation. The *merge(S_i, S_j)* operation is performed by merging two simultaneously active states S_i and S_j (for input a) thus decreasing k. The *split(S_i)* operation is performed by splitting state S_i into two states S_{im} and S_{in}. Both the *split* and the *merge* operations are performed with a goal of minimizing the overall size of the resulting automaton.

To change the automaton m (into a more deterministic or into a more non-deterministic system m') we search for a template automaton that requires the least number of operations to derive m'. Once the automaton m_i, which will be used to derive m', is determined, the reconfiguration process can start by applying the *merge* or the *split* operations required to achieve the level of non-determinism of m'. The operations are applied continuously until the desired level of non-determinism (k) is obtained. The merge and split operations are *safe*, meaning that after reconfiguration the automaton is left in the same state as prior to reconfiguring and the matching process can continue.

When to reconfigure: The reconfiguration process can be required or optional:

1) *Reconfiguration is required:* Occurs when the available system RAM does not meet the memory requirements of the

current automaton configuration.
2) *Reconfiguration is optional:* Occurs for two reasons: (i) system RAM is increased or (ii) the percentage of free cores is increased. In (i) we can decrease the non-determinism (via the *merge* operation) of the current configuration thus increasing system performance if the current nondeterminism of the system is greater than the number of available cores. In (ii) nondeterminism can be increased (via the *split* operation) to match the number of free cores. If both (i) and (ii) occur simultaneously, Morpheus considers (ii) first when searching for a feasible configuration because applying the *split* operation will not increase memory usage and therefore is considered a *resource safe* operation.

Search Space: When searching for a solution we consider the available resources R to find the target configuration T that maximizes the throughput. T is determined by first considering the available cores k and estimating if $|T_k| \leq R_m$ where R_m is the available memory resource and T_k is the target automaton with k level of non-determinism. Once T_k is determined we search for a template M which will be morphed into T_k. M is picked such that the potential number of operations $(|(|T_k| - |M|)|)$ required to derive T_k is minimal. To find both T_k and M in our implementation we perform a binary search. This is an applicable solution because in our case $max(k)$ (maximum level of nondeterminism) is about 16 (number of concurrent threads in a system) which is acceptable for today's multi-core systems.

Next we discuss the formulation of the split and merge operations as an optimization problem.

Split Operation: Suppose we have a DFA automaton M and the available cpu or free memory resources change. Suppose we are given the available parallelism of the system k. Assuming that $k > 1$, we want to increase the parallelism (non-determinism) of M while bounding the number of concurrently active states at k resulting in automaton M'. The nondeterminism of automaton M is increased via a *split* operation. Intuitively the split operation takes state S_i and splits it into two states S_{i1} and S_{i2}. All transitions belonging to S_i now are part of S_{i1} and S_{i2}. The split operation is repeated until nondeterminism of M is equal to k. Furthermore S_i is selected in such a way that $|M'|$ is minimized. The above problem is referred to as the *State Split Problem (SSP)* [28], and if we let S be the set of all NFA states, N be the set of states in M (i.e. all NFA state combinations), S_i $(i = 1, ..., N)$ be i-th combination of NFA active states, $S_{i,j}$ be the j-th subset split from S_i and Q be the union of $S_{i,j}$ (i=1,...,N; j = 1,...,k) the SSP can be formulated as the following equation:

$$\begin{gathered} min|Q| \\ s.t. \\ \cup_j S_{i,j} = S_i; (i = 1, ..., N; j = 1, ..., k) \\ Q = \{S_{i,j} | i = 1, ..., N; j = 1, ..., k\} - \{\emptyset\} \end{gathered} \tag{3}$$

Equation 3 minimizes the number of distinct $S_{i,j}$ which are non-empty and the union of which will translate into a smaller automaton. Authors in [28] show that the *SSP* problem is NP-complete for any $k > 1$ and therefore we rely on a heuristic to solve it. The heuristic used by Morpheus depends on u which specifies the minimum size of each $S_{i,j}$, which restricts the number of resulting sets. With this heuristic, when splitting S, at least one of the splits must be of size u. Once the $S_{i,j}$ is obtained, the problem reduces to a *minimum subset cover* problem where the minimum number of sets has to be picked to minimize $|Q|$. We implement other heuristics, such as limiting the amount of non-parallelism to consider k, requiring that there is no overlap between the subset split from the same NFA active state combination and requiring that at least one of the subset splits $S_{i,j}$ for some NFA active state combination y must satisfy $S_{i,j} = y$ where $i \neq y$. For more details on heuristics and optimizations used see Section III-E. The SSP problem can be formulated as a Linear Program (LP) which is presented in Equation 4.

$$\begin{gathered} min \sum_T X_T \\ \textstyle\sum_{D \in S} X_D \geq 1 \; \forall \; States \; S \\ X_T \geq X_D \forall \; New \; States \; T \\ 0 \leq X_T, X_D \leq 1 \forall \; decompositions \; D \end{gathered} \tag{4}$$

The LP formulation presented in equation 4 can be interpreted as minimizing the number of new states, with the following conditions: (i) for each decomposition of states X_D, we must pick at least one such decomposition and (ii) the number of new states must be at least as great as the number of decompositions, which guarantees that all original states can be derived. X_D is generated using the heuristics discussed. Note that during run time we collect access statistics p_i for each state i, thus all subsets are weighted by p_i. Initially the weight of all subsets is the same.

Merge Operation: Similarly to the split operation above we can define a linear program that solves the merge optimization problem. In the merge optimization problem a set S of states has to be derived that is a combination of a subset of the current K-FA states the union of which minimizes the overall size of the resulting automaton. The relaxed LP is depicted in Equation 5. The LP in equation 5 minimizes the number of unique sets as a result of merging two states from the possible K-FA 2-state combinations. The variable X_T represents the number of new states as a result of picking X_M.

$$\begin{gathered} min \sum_T X_T \\ \textstyle\sum_{M \in S} X_M \geq 1 \forall \; States \; S \\ X_T \geq X_M \forall \; New \; States \; T \\ 0 \leq X_T, X_M \leq 1 \forall \; decompositions \; D \end{gathered} \tag{5}$$

C. Update-Time

Based on the history of reconfigurations performed, an *update* of pre-computed configurations (*templates*) can be carried out. In other words, we may *change* a pre-computed configuration if a *better* pre-configuration can be generated which will result in a lesser dynamic configuration overhead.

A *better* pre-configuration p' may arise if an existing pre-configuration p is infrequently used thus utilizing resources without having a significant impact on the reconfiguration time. *Changing* a precomputed reconfiguration p involves either removing p or *deriving* p' via *split* or *merge* operation on p. Additionally, when resources change, we may need to remove or add a reconfiguration. Therefore the formulation in 1 must be rephrased to encompass the dynamic nature of the problem in terms of time t.

At time t two scenarios might occur:

- Resources may change.
- By keeping run-time statistics, we may notice that some pre-configurations p are less frequently used compared to others and the available resources can be utilized better by pre-computing other configurations instead of p'.

Thus, at time t, R_{tj}, specifies the capacity of resource j at time t, d_i, specifies the minimum number of operations needed to *derive* automaton (template) i, w_{it} specifies the weight of automaton i at time t. w_{it} increases with the increase of the frequency of usage of i. By changing the maximum distance d_i via the weight w_{it} as a function of time t our pre-configuration algorithm becomes dynamic with respect to both the time and the available resources. The above formulation can be expressed using the LP in Equation 6.

$$\begin{aligned} &minimize: \\ &\qquad max(\textstyle\sum w_{it}d_i) \\ &\qquad s.t. \textstyle\sum m_{ij}x_{ti} \leq R_{tj} \\ &\qquad x_{it} \in \{0,1\} \forall i \in m, t \in T \end{aligned} \tag{6}$$

Thus, based on the template usage statistics we can remove a template p, modify it to p' or add a new template p''. The weight w_{it} is adjusted by a constant c each time template i is used. In our prototype version of Morpheus c was specified manually, and a fully automated approach is part of the future work.

When solving the optimization problem in Equation 6, we have to estimate if updating template i to j is 'worthwhile'. When template i needs updating, we term this event as template i *expiring*. We can estimate the average throughput of i and j by calculating the average nondeterminism and the available processing power present. Specifically, the overall throughput can be estimated as

$$T = \alpha \sum_j \frac{\sum_i (d(s_j) \times (p_j + (1 - p_j))}{|s|} \tag{7}$$

where $|s|$ is the number of states of the current FSA, α is a system maximum throughput, d_sj is the determinism measure of state j (see Equation 2). Notice that if s_j has low throughput (e.g ≤ 0.1) and its activation probability p_j is also low (close to 0), the overall throughput will be close to 1 because the slow s_j will not be active frequently hence the term $(d(s_j \times (p_j) + (1 - p_j))$. Thus, T (the overall throughput of the system) is equivalent to the average throughput of all states j of the current automaton.

We can also track the average time i_t before template i *expires*. Thus to check if updating i is 'worthwhile' we compare the cost with the benefit of updating. The cost of updating is the reconfiguration (update) time and the benefit is the time it takes for i to process the same data as j ($\frac{T_i \times i_t}{T_j}$) where T_i is the throughput of automaton i. Thus reconfiguration is performed if the benefit is greater than the cost. Because of the benefit analysis before reconfiguration, Morpheus does not suffer from too much oscillation between different reconfigurations. Furthermore the response frequency to the changing system environment can be easily tuned in Morpheus by adjusting the minimum gap required for reconfiguration between the benefit and the cost.

D. Implementation

In this section we briefly discuss the implementation details. The components shown in Figure 2 are implemented in Java. We use *lp_solve* [2] to solve the Linear Program optimizations defined. To solve the fractional optimization LP problems presented, we use the technique of *rounding* [23], which in *expectation* will give an optimal integer solution. Thus suppose that a solution to an instance of the LP in Equation 4 is as follows: X_{D_1} = 0.2 X_{D_2} = 0.5 and X_{D_3} = 0.3. Via *rounding* we can get a good approximation to the above solution by rounding the largest X_D to 1 and others to 0, therefore X_{D_2} would be rounded to 1 and others to 0. Our experiments clearly show that using a rounding technique in expectation produces a good solution.

During run-time the access probabilities of states have to be updated and automaton reconfiguration has to be performed without losing the currently active states. The original NFA states are stored in a hash-table, which allows for a fast update of access probabilities of states. During reconfiguration, the necessary state transitions are copied to the currently active automaton. Furthermore, upon reconfiguration, the state of the previous automaton M_{old} should be restored in the new automaton M_{new} for the pattern matching process to continue from the same point. This is achieved by recording the active state labels in M_{old} and setting the same states to be active in M_{new}. By keeping the original NFA states separate from the states in the currently active automaton, the update of access probabilities and the reconfiguration of the current automaton can be performed efficiently.

E. Optimizations

In this section more details about the optimization techniques used for solving the split problem are presented. In [28] it is shown that the upper bound on $|Q|$ (the size of unique states after a split) using the heuristic presented in Section III-B is given by Equation 8.

$$|Q| \leq 2u' + N - \frac{\sqrt{2\alpha}}{m^2} t'^2 \tag{8}$$

The run time of the SSP algorithm [28], that is used to determine the split, is too expensive for the online environment thus we use an approach based on Fractional Linear Programming that provides a fast approximation (see Section IV-E).

Furthermore the heuristic in [28] generates a search space that can still be very large. We make this simple observation: a subset split must be such that: $\exists i S_i \in S_{i,j}$. In other words to reduce the size of the overall automaton, and thus the memory usage, a subset split must contain at least one of the nodes. If no such subset exists then the split is not performed. This observation in practice further reduces the search space. Figure 3 illustrates this heuristic, where the arrow from S_i to S_j indicates that S_i is a subset of S_j. Thus intuitively our LP solver would first pick the largest-degree subset whose sizes are larger than the threshold, and the degrees are larger than 1. Thus, first {A,D,O} is picked, and {A,D,O,G} is split into {A,D,O} and {G}. The sets {A,D,O,G} and {A,D,O} are then removed from the graph as well as the edges between these two vertices. Then we pick {G,O} and {O}, and split {G,O} into {G} and {O}. Removing all edges and nodes associated with {G,O} and {O} leaves an empty graph, therefore the final graph set of nodes resulting from the split are {A,D,O}, {O}, {G}.

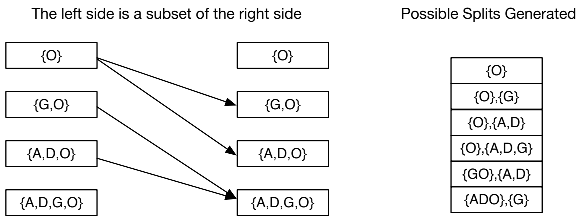

Fig. 3. Example of Subset Split Generation

Other optimizations used in Morpheus include the restriction on the amount of non-determinism to consider (k) and the *isolation constraint*. The bound on the amount of non-determinism is usually determined by the number of cores or by the number of concurrently supported threads. The bound on k reduces the search space during *run-time*. Furthermore similar to the heuristic in [28], we add the following isolation constraint:

$$S_{i,j}\hat{S}_{i,k} = \emptyset (\forall j \neq k, i = 1, ..., N) \quad (9)$$

The isolation constraint requires that there is no overlap between the subset split from the same NFA active state combination. Even with the isolation constraint, as evident from the experimental results (see Section IV-E), the approximation obtained by Morpheus for the split and merge operations is reasonable.

IV. Performance Evaluation

In this section we begin by demonstrating the accuracy of our resource and throughput estimation models of the K-FA automaton. The main motivation of our research is to improve the performance of pattern matching in environments where system resources constantly change. Because we are the first to propose a dynamic pattern matching system, we compare our approach against the fastest known static pattern matching system, YFilter [8], to validate our claim that dynamic pattern reconfiguration can dramatically improve pattern matching performance. We then study how the number of precomputed configurations affect the dynamic reconfiguration overhead and the amortized memory usage. The goodness of the split/merge approximation algorithms presented in Section III-B is also measured with respect to the optimal. The effect of our dynamic reconfiguration strategy is then measured on resource utilization and on the overall performance given workloads of different severity levels. As is explained in more detail in the following section, a *severity level* refers to the current input speed measured in characters per second (*cps*) and to the current memory and CPU load due to non-pattern matching related processes.

A. Experimental Setup

Workloads: There are two types of workloads that we use to test our system: (i) Server workloads and (ii) Input (transaction) workloads. For (i) we use our *synthetic micro-benchmark* (SMB) and the real work-load datasets. Using SMB we derive five independent workloads. In order to specify these workloads we use a *BurnInTest* [4] utility with which we specify the amount of *CPU* and *RAM* to utilize. By precisely controlling these parameters we obtain five different workloads of varying time-patterns (sinusoidal, sawtooth, flat with different amplitude and period etc). The goal of these workloads is to validate that Morpheus can dynamically adjust the underlying automaton under various system loads. Furthermore these workloads test the ability of our system to automatically recognize an opportunity for dynamic reconfiguration. Furthermore we obtained real-world load-statistics from *Wikipedia* and *Nasdaq* servers. Pattern matching is heavily used in these services to respectively detect any malicious activity and investment opportunities in the current stream of stock prices.

To simulate the transaction load we use the MIT Lincoln Lab intrusion detection traces [19]. These traces provide a way to test a real system against the network intrusion attacks. To process these traces we used a set of patterns found in Snort network intrusion detection system [24]. The advantage of using the real MIT Lincoln Lab traces is that it tests the nature of real attack traffic that includes periods of spurious activity.

A high severity level refers to the system load of 90% for both CPU and RAM (10% of resources available) and the input speed of 100K *cps*. A low severity level refers to 10% system load and 1K *cps*.

Data Sets: We use a mixture of real pattern queries and synthetically designed queries. Real pattern queries are retrieved from Snort [24] Network Intrusion Detection system. To generate synthetic queries, we design a query generation tool to synthesize pattern queries with predictable properties.

Our query generator tool responds to a number of parameters presented in Table III.

System Information and Implementation: All experiments were run on an eight core 2.0GhZ with hyper-threading server with 16GB of RAM running Java 1.6. We implement the monitoring tools in Java and utilize SSH to collect OS-level statistics. Morpheus utilizes an open source LP solver *lp_solve* [2] to solve our global linear approximation program. We have used the standard automaton library for java to generate NFAs and DFAs [1].

B. Resource Estimation

Given an environment with limited resources, the resource consumption by a particular configuration c must be estimated without explicitly computing c. Based on the observations made in Table I we used a simple linear approximation for the size of the automaton. This experiment measures the estimated size of our automaton versus its actual size, for a given level of non-determinism K. The results indicate a fairly small overestimate of the memory usage of *K-FA* (Figure 4). This is expected because frequently the memory is reduced due merging opportunities of duplicate states. The CPU usage is slightly over-estimated (Figure 5), which can be explained by an irregular work-load. The memory overestimation problem is mitigated by a reasonable over-estimation of the *RAM* resource is acceptable because it is considered to be a limiting constraint.

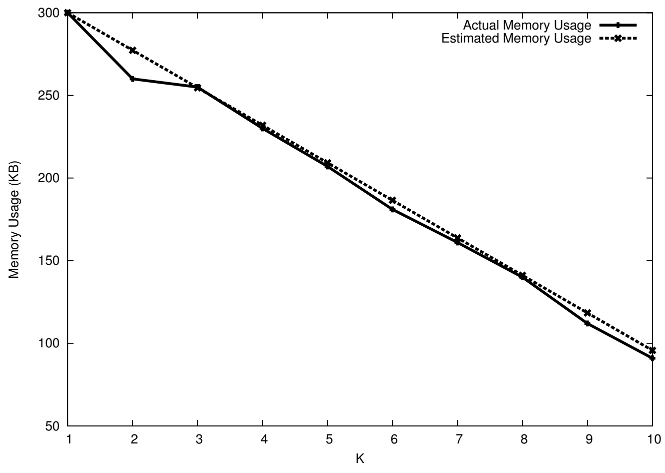

Fig. 4. Actual vs Estimated resource usage (Memory)

C. Comparison against YFilter

We compared Morpheus against the state of the art XML index YFilter. The results can be seen in Figure 6. We have varied the input speed *cps* given an automata consisting of 500 HTTP pattern queries found in Snort. As the input speed increases, the system load also increases thereby reducing system resources, measured in available memory from 90% to 10% at the peak of *cps*. The results confirm that due its inherently static NFA implementation, YFilter is unable to adapt to a changing system environment. We have varied the input speed, measured in characters per second (*cps*), between 500 *cps* and 128K*cps* and recorded the average time for both YFilter and Morpheus to perform a match. The match time can be interpreted as latency for the overall system. Thus, even though YFilter has a fast and memory-light approach to represent patterns, its inability to adjust to the changing system environment causes it to suffer from poor performance by not fully utilizing the available system resources. Our approach is able to quickly adjust to the current system load and thus shows a five-fold performance gain over YFilter. Morpheus experiences slight increase in system resources during reconfiguration (roughly 3%), however the overall matching time of Morpheus is still significantly less than that of YFilter. Given the input speed of 1K *cps* Morpheus outperforms YFilter in matching speed 1100230ns to 50000000ns, and for 128K *cps*, due to a lesser amount of available resources the performance gap decreases with Morpheus and YFilter having match times of 2400230ns to 68000000ns. It should be also noted that in the case when the amount of system resources is equal to the NFA storage requirements, then our approach defaults to that of the YFilter, namely utilizing a full NFA for pattern matching.

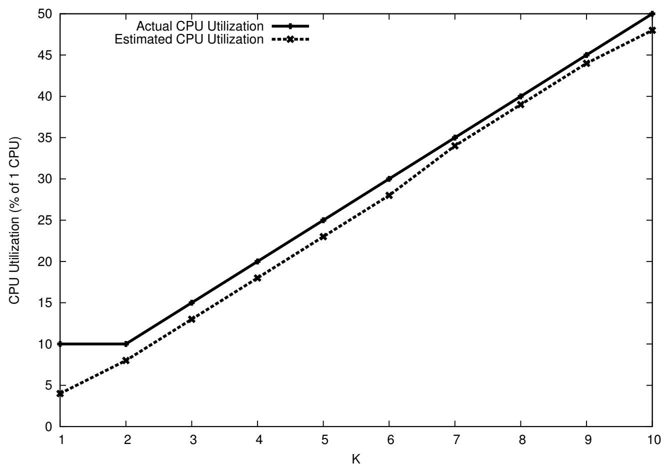

Fig. 5. Actual vs Estimated resource usage (CPU)

Synthetic Query Parameter List		
Parameter	Range	Description
Q	1000 to 500000	Number of queries
W	0 to 1	Probability of a wild-card ”*” occurring in a qeury
Distinct	0-100%	Percentage of unique predicates
P	0 to 20	Number of predicates per query

TABLE III
PARAMETERS FOR SYNTHETIC QUERY GENERATION

D. Optimal number of precomputed configurations

The experiment in Figure 7 shows the effect of the number of precomputed configurations on the dynamic reconfiguration time and on the memory usage. Given a current system C, a target system T and a set of precomputed configurations $S = \{S_1, S_2, ..., S_n\}$, the dynamic reconfiguration time includes the time to search for a solution in S given C and T, morph the found solution S_k into T and update S depending on the reconfigurations performed so far (see Section III-C). The *amortized memory* usage refers to the memory used for the buffer to hold the tuples from the input stream while

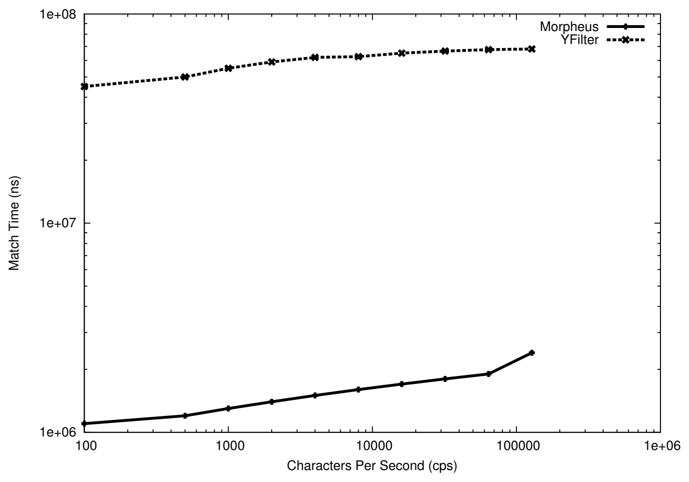

Fig. 6. Matching speed

reconfiguring and the size of S. Note that the longer it takes to perform a reconfiguration, the larger the input buffer has to be. Figure 7 shows that in our case, after precomputing more than five configurations, the search and update time of S starts to dominate and the performance deteriorates. Furthermore the amortized memory usage starts to increase after five precomputed configurations because an additional precomputed configuration consumes more memory than the decrease in the size of the temporary input memory buffer needed. From this experiment we can conclude that given our system set-up and the workload from SMB the optimal number of precomputed configurations to achieve the highest throughput should be five.

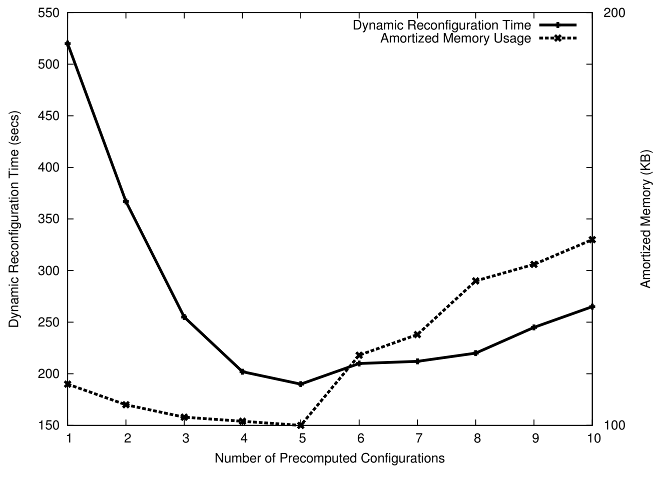

Fig. 7. The effect of increasing the number of preconfigurations on the dynamic reconfiguration time and on the amortized memory usage

E. Optimal Split vs Approximate Split

In Section III-B we have presented an approximation *LP rounding* formulation for split and merge problems. In this experiment we measure how the approximation factor ϵ (error) affects the resulting size of the automaton. The error in Figure 8 measures the fraction of the set split generated. Thus error of 'n' implies that $\frac{1}{n}$ subset splits out of possible $S_{i,j}$ splits are generated. Therefore when ϵ=1, we get the best split because our search space includes a collection of all possible splits. By decreasing ϵ we spend more time on deciding how to perform a split but in return lower the size of the resulting automaton. We observe that the benefit of a lower ϵ decreases rapidly (in terms of the memory usage). Furthermore the less time we spend on choosing the right split, the less of an input buffer we need, thus the point p of where the buffer size and the automaton size intersect is the optimal ϵ necessary to minimize the overall memory usage for our SMB. The optimal value of p can be determined by a simple online process that is being incorporated into Morpheus.

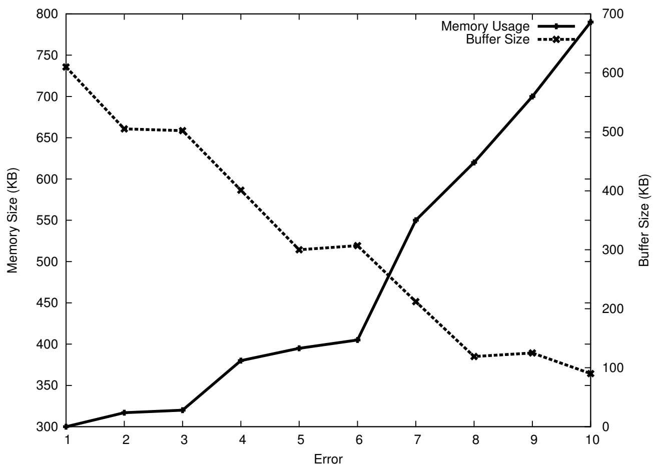

Fig. 8. Optimal vs Approximated split operation

F. Effects on Throughput

In this experiment we evaluate the effect of dynamic reconfiguration on the overall throughput. The evaluation is done by comparing the memory usage increase and the overall throughput increase with respect to a situation where there is no dynamic reconfiguration. Figure 9 shows that for a fraction of the cost in memory increase we get a great increase in throughput. The x-axis in Figure 8 represents the severity of the underlying system changes (i.e. severity of 1 implies that only memory slightly increased/decreased and severity of 10 indicates that the CPU usage, memory and access patterns have changed dramatically). To simulate severity we used the *Burn In Test* utility to specify precisely how much *CPU* and *RAM* resources to consume.

V. Related Work

Our work is dependent on research areas of query languages, hardware description languages, pattern matching and adaptive query processing. A user uses a query language and a hardware description language to provide patterns and a description of system resources and their tradeoff, respectively, to Morpheus. Subsequently, Morpheus uses dynamic reconfiguration of a pattern automaton to improve the performance of pattern

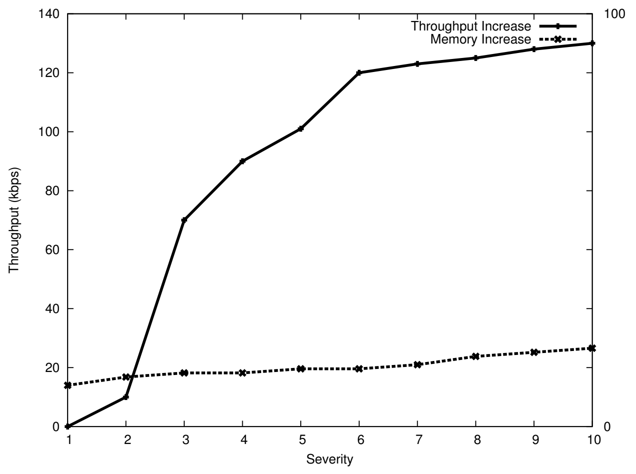

Fig. 9. Affects of the dynamic update on throughput

matching, in a similar way that adaptive query processing adjusts the query plan during run time to improve query performance. We discuss main works from the fields above next.

Pattern Matching: There is a wide variety of work on pattern matching all of which assume a static environment and thus static optimizations. To keep the static size of the automaton small, several limitations had to be imposed. Examples of such limitations include (i) disallowing the use of the Kleene Closure in underlying REs [27], (ii) performing pattern matching without outputting complete matches [7] and (iii) employing a spatial indexing structure for indexing which is not suitable for data streams [5].

Authors in [29] proposed a technique for rewriting regular expressions to avoid exponential memory blow-up of a DFA due to Kleene Closure, however according to [18] their approach only works for a subset of Snort REs [24].

Another way to adhere to the system resource constraints is to construct a DFA lazily (on the fly) [11] assuming that a completely expanded lazy DFA will be small enough to fit in main memory; however this is a strong assumption which does not always hold.

There are many other efficient techniques to do pattern matching [5], [16], [8], however none of the techniques address a resource constrained environment where system-stress level changes and dynamic adjustment of the underlying pattern representation is necessary.

Language: Static pattern matching is supplemented with several notable languages including *SQL-TS* [25], *MATCH-RECOGNIZE* [3], *SASE+* [9] and *K*SQL* [20] .

MATCH-RECOGNIZE [3] comes from a 2007 ANSI standard proposal to add the new SQL functionality for finding patterns defined as regular expressions over sequence of rows via a *MATCH_RECOGNIZE* clause.

SQL-TS [25] (Simple Query Language for Time Series) is the first SQL language extension proposal to support pattern queries. *SQL-TS*, besides proposing a set of new language constructs also discusses the optimizations of pattern matching.

SASE+ [9] proposes a compact language that can be used to define a wide variety of patterns containing Kleene Closures. The authors rigorously studied language semantics and analyzed the expressive power of SASE+, SQL-TS and Cayuga [7], which is a stateful publish/subscribe system based on the NFA.

*K*SQL* [20] supports a more expressive language that allows for generalized Kleene-Closure queries and also achieves the expressive power of the nested word model, which greatly expands the application domain to include XML queries, software trace analysis, and genomics.

The above languages provide an efficient way to express patterns. Morpheus would be used to accept pattern definitions expressed in these languages to construct an automaton for dynamic pattern matching.

Adaptive Query Processing: Our work uses similar motivation to Adaptive Query Processing (AQP) [10], [14], namely the fact that the static query does not provide optimal performance in an environment where the system resources and data characteristics constantly change. Instead of dynamically modifying the query plan as is done with AQP our approach modifies the underlying system, which in our case is a pattern automaton, for optimal pattern matching performance. The key idea of our work is dynamic automaton reconfiguration, which is different from AQP, and has not yet been addressed in literature.

Hardware Description Languages: In dynamic reconfiguration environments, a description language plays an important role in defining system dynamics when resources change. AADL (Architecture Analysis and Design Language) [26] is an architecture description language which allows one to describe distributed real time embedded systems by assembling components. AADL defines an interface for each component and separates between the implementation of component and the description of its interface. It can describe both the software and the hardware parts of the system. AADL supports reconfigurable systems by describing their modes and the mode transitions. Similar to the *pre-configurations* defined in our work AADL defines *models* which represent a particular reconfiguration of a system and a transition represents an event which allows for the reconfiguration of the system. A language like AADL can aid Morpheus in computing a utility of a particular reconfiguration. A user may describe a tradeoff between CPU and memory using AADL which Morpheus would then take into account when selecting a reconfiguration.

There are existing hardware systems that use AADL to provide support for dynamic reconfiguration. This work is done largely in the realm of FPGA reconfiguration and is related to Morpheus because of the emphasis on minimizing the time of dynamic system reconfiguration in order to improve the overall run-time performance.

Authors in [17] study the dynamic reconfiguration of hardware (FPGA). A caching approach is proposed where the assumption is made that the hardware system needs to be frequently reconfigured to derive the full potential of the underlying system. The caching approach reduces the reconfiguration overhead thus increasing the overall performance of the system. The proposed FPGA caching techniques are similar to traditional caching where authors exploit both spatial locality and temporal locality. Similarly, in [13] authors use the idea of distributed caching to minimize the overhead of dynamic reconfiguration. The key observation is that a small percentage of loops account for the majority of execution time, therefore only these loops are mapped onto the reconfiguration coprocessor. The loops are represented as graphs, where nodes and edges are functions and dependencies between functions respectively. Final graph G is constructed that maximizes parallelism. For that purpose bipartite matching is used. The frequently used configurations are remembered using distributed caching. Our work does not rely on caching, but can supplement caching approaches.

Currently, in our work the user specifies patterns using a *regular expression* syntax which are then compiled and automatically tuned given the available resources. We use a static utility which optimizes throughput given a constrained memory resource. In our future work, similar to *AADL*, we will strive to give the user the flexibility of specifying the custom details of system resources and their dynamics.

VI. Conclusion and Further Work

This paper describes the dynamic pattern matching prototype called *Morpheus* that improves the performance of pattern matching with the help of dynamic reconfiguration of the underlying pattern automaton given a changing system stress load. Morpheus precomputes several key pattern configurations (templates) that are then morphed into a required form during run-time depending on the available system resources. Our extensive experiments show that the approach proposed is attractive due to its low amortized memory overhead and an order of magnitude decrease in dynamic reconfiguration time when compared to standard approaches that do not use templates.

The few assumptions made by Morpheus are actually satisfied in real world applications. Morpheus prototype does not support load-shedding because we assume that every tuple must be processed. This is an acceptable assumption given a sequential pattern semantics. Nevertheless, by supporting load-shedding, the overhead of dynamic reconfiguration may further be reduced by decreasing the quality of service guarantee, and we plan to investigate this in our future work.

Morpheus also assumes that no new patterns are added throughout execution. This is reasonable for network intrusion systems where a set of patterns is predefined.

The implementation presented is primarily geared towards pattern matching using a standard finite automaton, however the ideas discussed can be extended to abstract objects. In order to support dynamic reconfiguration of abstract objects the user would have to supply her own *merge* and *split* operations using the language provided.

Overall, the Morpheus prototype is the first step towards solving a largely overlooked yet important problem of dynamic pattern matching.

References

[1] Automaton implementation. `http://www.brics.dk/automaton/`.

[2] lp_solve, a Mixed Integer Linear Programming (MILP) solver. Website.

[3] Pattern matching in sequences of rows. sql change proposal. http://asktom.oracle.com/tkyte/row-pattern-recogniton-11-public.pdf. Technical report, 2007.

[4] Burn-in test, 2011. `http://www.passmark.com/products/bit.htm`.

[5] C. Y. Chan, M. N. Garofalakis, and R. Rastogi. Re-tree: an efficient index structure for regular expressions. *VLDB J.*, 12(2):102–119, 2003.

[6] E. de Lara, D. S. Wallach, and W. Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. In *USITS*, pages 159–170, 2001.

[7] A. J. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. M. White. Towards expressive publish/subscribe systems. In *EDBT*, pages 627–644, 2006.

[8] Y. Diao and M. J. Franklin. High-performance xml filtering: An overview of yfilter. *IEEE Data Engineering Bulletin*, 26:41–48, 2003.

[9] Y. Diao, N. Immerman, and D. Gyllstrom. Sase+: An agile language for kleene closure over event streams. Technical report, 2007.

[10] F. Farag, M. A. Hammad, and R. Alhajj. Adaptive query processing in data stream management systems under limited memory resources. In *PIKM*, pages 9–16, 2010.

[11] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.

[12] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[13] Z. Huang and S. Malik. Managing dynamic reconfiguration overhead in systems-on-a-chip design using reconfigurable datapaths and optimized interconnection networks. In *DATE*, page 735, 2001.

[14] Z. G. Ives, A. Deshpande, and V. Raman. Adaptive query processing: Why, how, when, and what next? In *VLDB*, pages 1426–1427, 2007.

[15] S. Kumar, B. Chandrasekaran, J. S. Turner, and G. Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *ANCS*, pages 155–164, 2007.

[16] D. G. Lee, Y. J. Jung, Y. W. Lee, and K. H. Ryu. Hashed multiple lists: A stream filter for processing continuous query with multiple attributes in geosensor networks. pages 104–109, July 2008.

[17] Z. Li, K. Compton, and S. Hauck. onfiguration caching techniques for fpga. IEEE Symposium on FPGAs for Custom Computing Machines, 2000.

[18] A. Majumder, R. Rastogi, and S. Vanama. Scalable regular expression matching on data streams. In *SIGMOD Conference*, pages 161–172, 2008.

[19] MIT. Darpa intrusion detection evaluation.

[20] B. Mozafari, K. Zeng, and C. Zaniolo. K*sql: a unifying engine for sequence patterns and xml. In *SIGMOD Conference*, pages 1143–1146, 2010.

[21] B. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *SOSP*, pages 276–287, 1997.

[22] F. Peng and S. S. Chawathe. Xpath queries on streaming data. In *SIGMOD Conference*, pages 431–442, 2003.

[23] P. Raghavan and C. D. Thompson. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. Technical Report UCB/CSD-85-242, EECS Department, University of California, Berkeley, May 1985.

[24] M. Roesch. Snort: Lightweight intrusion detection for networks. In *LISA*, pages 229–238, 1999.

[25] R. Sadri, C. Zaniolo, A. M. Zarkesh, and J. Adibi. Optimization of sequence queries in database systems. In *PODS*, 2001.

[26] A.-. E. C. S. C. SAE. Architecture Analysis & Design Language (AADL). SAE Standards n^{o} AS5506, November 2004.

[27] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD Conference*, pages 407–418, 2006.

[28] Y. Xu, J. Jiang, Y. Song, T. Jiang, and H. J. Chao. i-dfa: A novel deterministic finite automaton without state explosion. Technical report, Polytechnic Institute of New York University, Brooklyn, NY, 2010.

[29] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *ANCS*, pages 93–102, 2006.