Evolutionary Algorithms that use Runtime Migration of Detector Processes to Reduce Latency in Event-Based Systems

Christoffer Löffler[1,2] and Christopher Mutschler[1,2] and Michael Philippsen[1]
christoffer.loeffler@informatik.stud.uni-erlangen.de
{ christopher.mutschler | michael.philippsen }@fau.de

[1]Programming Systems Group, CS Dept., University of Erlangen-Nuremberg, Germany
[2]Sensor Fusion and Event Processing Group, Locating and Comm. Systems Dept.,
Fraunhofer Institute for Integrated Circuits IIS, Erlangen, Germany

***Abstract*—Event-based systems (EBS) are widely used to efficiently process massively parallel data streams. In distributed event processing the allocation of event detectors to machines is crucial for both the latency and efficiency, and a naive allocation may even cause a system failure. But since data streams, network traffic, and event loads cannot be predicted sufficiently well the optimal detector allocation cannot be found a-priori and must instead be determined at runtime.**

This paper describes how evolutionary algorithms (EA) can be used to minimize both network and processing latency by means of runtime migration of event detectors. The paper qualitatively evaluates the algorithms on synthetical data streams in a distributed event-based system. We show that some EAs work efficiently even with large numbers of event detectors and machines and that a hybrid of Cuckoo Search and Particle Swarm Optimization outperforms others.

***Index Terms*—Heuristics, Evolutionary Algorithms, Particle Swarm Optimization, Publish/Subscribe, Distributed Computing.**

I. Introduction

Event-based systems (EBS) are used to analyze high data rate sensor streams in many applications such as surveillance, sports, finances, RFID-systems, etc. [1]. The aim of an EBS is to filter and aggregate events, i.e., special occurrences of interest, and to iteratively transform them into higher level events until they reach a level of granularity that is appropriate for an end user application or for triggering some actions.

Think of autonomous robots that play soccer and locate their positions [2]. The robots form a wireless ad-hoc network to run both distributed decision making algorithms and rule violation detection. The algorithms are implemented as a distributed EBS, and events and sensor readings can be transmitted and processed on any robot. For instance, a high-level tactical rule violation such such as *offside* can be defined by composing events from lower levels, see Fig. 1. Event detectors are spread over the available computing nodes and the robots iteratively aggregate and process events until they detect the top-level event. Especially for tactical decisions but also for the detection of rule violations a low latency processing is crucial and event detectors must be allocated carefully across the network. Similar use-cases appear in military maneuvers or in autonomous coordination of (Mars) robots.

Rarely there is an optimal allocation of event detectors to the distributed processing units because the robots are mobile, they constantly interact with each other, experience continuously changing tasks and latencies, and the event loads are unknown a-priori.

Hence, an allocation must continuously adapt to the system environment and data load in order to be optimal. Unfortunately, not only is finding an optimal allocation known to be NP-hard, but also as good allocations quickly turn deprecated, slow exact algorithms are useless. If the optimizer takes too long to allocate a large number of event detectors to the available nodes, event loads may already have changed again. The optimal allocation is a moving target.

Greedy approaches are insufficient as they only improve an allocation locally, are likely to get stuck in local optima, and only obtain small performance benefits.

This paper exploits the fact that sensor data, events and performance measurements can be wirelessly broadcast to a central unit. We show how Evolutionary Algorithms (EA) can be used to derive good solutions for the latency optimization problem and to produce migration commands that are sent to the respective nodes to migrate their event detectors. EAs are well-suited for the moving target problem. They terminate

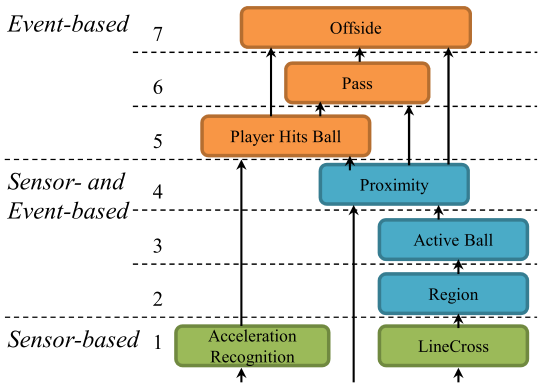


Fig. 1. Event hierarchy for an *offside* rule violation.

their search after a while, pick the currently best solution, and migrate the event detectors accordingly.

We evaluate different heuristics for the optimization of event-based systems and show why some of them are or are not suited for event-based systems. The requirements for such heuristics are twofold. First, since networking delays are high for wireless transmissions and since we face a hierarchical communication pattern, the primary goal is to minimize network transmissions. Second, global knowledge on event loads can be harvested from all robots/hosts and should be used for better results. Event stream data is continuously collected at a centralized point and a search heuristics is triggered to calculate an optimal allocation for the distributed system. Event detectors are then migrated between nodes to achieve a better performance in terms of latency.

The rest of the paper is organized as follows. Section II reviews the related work. Section III provides basic definitions and describes our runtime system. Section IV then formalizes the optimization problem before Section V presents the optimization heuristics. We evaluate the heuristics in Section VI, discuss the results when processing synthetic event data streams, and show that a Cuckoo Search / Particle Swarm Optimization (CS/PSO) hybrid outperforms the other heuristics.

II. Related Work

Related work for runtime latency optimization is found in various areas such as distributed operating systems, sensor networks, and event-based systems. But most of them optimize other criteria or do not meet the above requirements.

Xing et al. [3] and Balazinska et al. [4] periodically collect CPU load statistics and use a greedy algorithm to migrate pairs of filter operators in the Borealis EBS [5]. Because their filter operators run for many seconds, they only consider CPU load but do not address network latency at all.

Plan-based approaches [6, 7, 8] statically calculate an optimal allocation a-priori but do not adapt event detector placements at runtime to changing event loads.

Liu et al. [9] address main memory shortage and split operator states for long running, non-blocking queries such as multi-joins. However, they focus on splitting large database operations and do not consider network latency as critical. Threshold-based techniques [10] also only optimize CPU and memory loads but do not optimize networking latency.

Lakshmanana et al. [11] split event processing agents into several *strata* that run in parallel before they profile for load balancing. But since they just can migrate few agents they only approach local optima.

With projecting event queries onto distributed hash tables [12, 13], queries can be divided into sub-queries and loads can be moved between hosts that are near to save communication cost. These approaches optimize locally and may also result in worse allocations if event detectors form a detection hierarchy.

Zhou et al. [14, 15] focus on local load balancing in publish/subscribe systems where communication is not tightly coupled. In contrast, in our application domain the communication paradigm is tightly coupled and the event loads are dynamic.

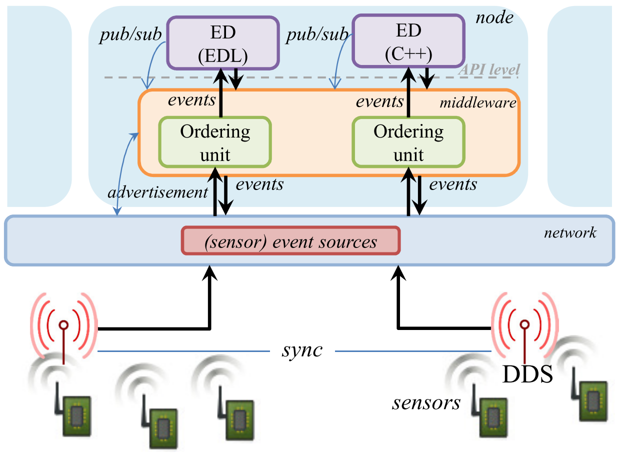


Fig. 2. Distributed publish/subscribe EBS.

The distributed Lagrangian Rates, Greedy Populations (LRGP) [16] algorithm optimizes resource allocation by adapting flow rates and consumer admissions. However, reducing consumer admission to control the resource allocation is no viable solution because they drop events which may result in event detection failures.

Babcock et al. [17] adaptively optimize operator scheduling to minimize memory consumption and throughput latency. However, they replace the order in which stream operators are applied to a number of data packets from multiple input streams on a single machine. This is an orthogonal type of optimization as we optimize the placement of operators in a network.

MOSIX [18, 19] is a distributed cluster operation system that proactively migrates processes at runtime to optimize the performance. Decisions are based on a theoretical cost model that incorporates CPU time and memory consumption of jobs, collects node statistics from the network, and (re-) assigns jobs. However, the communication costs *between* jobs are not optimized.

Khanna et al. [20] use a genetic algorithm (GA) to generate an optimal number of cluster heads in sensor networks to minimize communication cost, and hence to maximize the lifetime of the sensor net. However, in sensor nets the requirements usually differ because sensor nodes transmit data towards a sink and only rarely communicate with each other to perform distributed tasks.

Ant Colony Optimization (ACO) [21] is a popular method for network-based optimization. However, those technique are mainly used to optimize network routes or to provide alternative routes for end-to-end or multiple-end communications. Since we essentially want to optimize a complex multi-hop scenario, known ACO methods cannot be applied.

III. Architectural Overview

Fig. 2 depicts our distributed publish/subscribe-based event processing system. It is a network of several machines that run the same middleware to process sensor readings that are collected from a number of data distribution services (DDS), e.g., antennas that collect RFID readings. Event detectors are spread across the machines. A detector communicates subscriptions, publications and control information with the middleware that does not know the detector's event pattern. The detector is unaware of both the distribution of other event detectors and the runtime configuration. The middleware implements a push-system with unknown subscribers. At system startup the middleware has no clue about event delays on other hosts but just notifies other middleware instances about event publications. The middleware is thus generic and encapsulated, and does not incorporate the application-specific event definitions of the event detectors. The middleware can be externally triggered to migrate event detectors to other middleware instances.

IV. Runtime Latency Optimization

Our runtime latency optimization moves event detectors to improve latencies. As event streams cannot be predicted sufficiently well, plan-based approaches do not work for online optimization. As greedy approaches often end up in local optima we apply heuristics to optimize the detector distribution at runtime. This requires continuous monitoring of all event stream statistics in a centralized optimization master (OM) component. The OM collects the number of transmitted events for each `ID`, the measured network latencies to other nodes, and the number of generated events. The OM periodically triggers one of the heuristics of Section V.

A. Problem Formalization

Let n be the number of event detectors, m the node count. An allocation is a three-tuple consisting of an $m \times n$ matrix X with $x_{i,j}$=1 iff detector i runs on node j, an $n \times n$ matrix L where $l_{i,j}$ is the latency between hosts i and j, and an $m \times m$ matrix T where $t_{i,j}$ is the number of events transmitted from node i to j. The goal is to minimize

$$\varphi(X, L, T) = \sum_{i=1}^{m} \sum_{j=1}^{m} \sum_{k=1}^{n} x_{i,k} \cdot x_{j,k} \cdot (c_t \cdot t_{i,j} + c_l \cdot l_{i,j})$$

which describes the cost of the current detector distribution under the runtime measurements L and T, weighted by c_t (emphasis on network transmissions) and c_l (emphasis on latency).

B. Solution

An optimization based on heuristics needs (1) a formal encoding of possible solutions (individuums), and (2) a fitness function that grades the solution's performance.

Individuum. A detector i runs on host x_i in an event detector allocation $\vec{X} = (x_1, x_2, \cdots, x_n)^T$, with $x_i \in [1; m]$.

Fitness function. The fitness function is used to grade a possible solution and projects it onto a number. The higher the quality $f(\vec{X})$ of a solution $\vec{X}$ is, the fitter is the individuum in the evolutionary algorithm. Our fitness function incorporates network latencies and traffic, and is again weighted by the parameters c_t and c_l:

$$f(\vec{X}) = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{i,j} \cdot c_t \cdot t_{i,j} - \sum_{i=1}^{n} \sum_{j=1}^{n} b_{i,j} \cdot (c_t \cdot t_{i,j} + c_l \cdot l_{i,j}).$$

$a_{i,j}$ is 1 if event detectors i and j run on the same host, i.e., $x_i = x_j$, and 0 otherwise. Contrary, $b_{i,j}$ is 1 if detector i and j run on different hosts, and 0 otherwise. The left part of the fitness function sums up the *cost savings* by events that are transmitted locally between event detectors and that must not travel through the network. The right part sums up the cost for detector dependencies that cause network traffic. Hence, $f(\vec{X})$ grows with locally transmitted events and decreases with events that are subscribed by hosts with higher latency or with more generated events. Due to limited CPU power we skip and discard solution vectors that would overload hosts.

V. Heuristics

This paper evaluates four heuristics to dynamically optimize event detector allocations: a Genetic Algorithm (GA) [22], a Cuckoo Search (CS) [23], a Particle Swarm Optimization (PSO) [24], and a hybrid form of CS and PSO [25]. Below we shortly describe their principles and the guts of their application to the runtime optimization problem.

A. Genetic Algorithm (GA)

The classic GA is based on fundamental mechanisms of the theory of the natural evolution and the continuous change of DNA sequences to improve a species' fitness.

A population consisting of d distributions $\vec{X}$ reproduces for k generations. In each generation we apply three operations: (1) selection, (2) crossover, and (3) mutation.

(1) Selection. We evaluate the fitness of each solution of the current generation and sort them by their calculated fitness values. We use a fraction p, with $0 < p < 1$, of the highest ranked solutions in the following step.

(2) Crossover. We choose two solutions, $\vec{X}_a$ and $\vec{X}_b$, randomly from the top fraction p of the current generation. We randomly choose a subsequence $(x_i, x_{i+1}, .., x_j), 1 \leq i < j \leq m$ from both $\vec{X}_a$ and $\vec{X}_b$, and exchange this sequence between the solution vectors. We evaluate the fitness of the resulting distributions $f(\vec{X}'_a)$ and $f(\vec{X}'_b)$, and if they are better than $f(\vec{X}_a)$ and $f(\vec{X}_b)$, we replace them.

(3) Mutation. To overcome local optima and to generate new solutions, we randomly choose a distribution $\vec{X}_c$ among the population for mutation and replace a randomly chosen subsequence of its solution vector with valid random values. We determine the fitness of the resulting distribution $\vec{X}'_c$, and if it is better, we replace $\vec{X}_c$ with $\vec{X}'_c$.

B. Cuckoo Search (CS)

Cuckoo Search [23] is an iterative and evolutionary algorithm that manipulates an initial set of d solutions over a given number n of generations. CS uses (1) the parasitic breeding behavior of some species of cuckoo birds and (2) the characteristics of the random walk of the common fruit fly's method to determine its route for exploring feeding grounds.

(1) Breeding behavior. The population of solutions is represented by d bird's nests (containers), each of which holds a solution $\vec{X}$. To stay in the metaphor of CS, these nests are owned and used as breeding locations by other species of birds. A cuckoo bird reproduces by laying its own egg into one of these nests and lets the victimized bird breed it. The victims either hatch the egg, evict it, or abandon their own nest. How a cuckoo generates an egg is discussed in (2).

We implement these options as follows. In each generation, the cuckoo generates a new egg $\vec{X'}$, and if it represents a better solution with a higher fitness value $f(\vec{X'})$ than the egg $\vec{X_j}$ of a randomly chosen nest j, it replaces $\vec{X_j}$. Otherwise $\vec{X'}$ is discarded. Furthermore, in each iteration a fraction p of eggs with low fitness values is abandoned and replaced by randomly generated solutions/eggs in new containers/nests.

(2) Egg generation. After randomly selecting a nest i the cuckoo takes the egg $\vec{X_i}$ and performs a Lévy Flight in the solution vector. The Lévy Flight is the common fruit fly's method for determining its route, i.e., a special random walk, and we use it to explore the possible allocations. The random walk shifts some of the dimensions of vector $\vec{X_i}$:

$$\vec{X'} = \vec{X_i} + c \cdot Levy\vec{Flight}(\alpha).$$

The step length is determined by a heavy-tailed random distribution that has an infinite variance with an infinite mean. The factor c is used to weight the random walk, and α is the Lévy distribution's parameter. This walk varies from local exploration steps to steps far out of the current proximity. This avoids local optima and explores new locations more quickly.

Due to the discrete and interdependent characteristics of the representation of the event detector allocation in $\vec{X_i}$, we do not shift all dimensions of $\vec{X_i}$ at once. A change caused by a random walk in one dimension can have a ripple effect on the quality of the allocation of other event detectors to their host machines. Therefore it is necessary to limit the impact of the random walk in our scenario. Hence, we randomly select a small subset of dimensions of $\vec{X_i}$ and perform the random walk on each of them. The step length of the random walk is also important. It influences not just its own dimension but possibly several more because event detectors subscribe events from other event detectors.

Algorithm 1: The Cuckoo Search algorithm.

```
Data: fitness function $f(\vec{X})$, Lévy Parameter $\alpha$,
unfit fraction p;
nests ← generate initial set of $d$ allocations;
while (t < MaxGeneration) or (!stop-criterion) do
    $i$ ← nests.getRnd();
    Allocation $\vec{X'}$ ← $\vec{X_i}$.LévyFlight($\alpha$);
    $j$ ← nests.getRnd();
    if $f(\vec{X'}) > f(\vec{X_j})$ then
        $\vec{X_j}$ ← $\vec{X'}$;
    sort all allocations $\vec{X}$ in nests by fitness value;
    replace the fraction $p$ of the nests with a low fitness
    value with random new and valid ones;
return best allocation
```

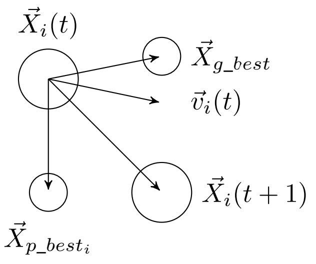


Fig. 3. The way PSO combines weighted forces.

The result $\vec{X'}$ of the random walk is then used as a new cuckoo egg. It is evaluated according to the rules described in step (1) and, depending on the result, placed into the nest j chosen in step (1).

The fraction p of eggs to abandon at the end of each generation is also important for the algorithm's performance. It is essentially a global search with a high randomness, and not a random walk. However, since the probability is small to create good eggs by chance we get better results with a small fraction p.

Alg. 1 shows the pseudo-code of the CS algorithm. The stop criterion can be any condition, for instance a well chosen maximal number of iterations in which the best solution did not change significantly. We leave that out for simplification.

C. Particle Swarm Optimization (PSO)

PSO applies techniques from the swarm behavior of some animals/particles, especially birds and fish. The pseudo-code is given in Alg. 2. A population of d animals/particles, each a representation of a solution $\vec{X}$, flocks towards an optimal state. To implement the swarm behavior, it is necessary to know the velocity $v_i(t)$ at time t of each particle i and the globally best particle $\vec{X}_{g\_best}$ found so far, evaluated by $f(\vec{X})$. Furthermore, we need the personal best position $\vec{X}_{p\_best_i}$ of each particle i. A particle i then moves through the solution space into a direction computed from these values as shown in Fig. 3. Operating on the set of particles, a time-discrete loop performs the following three steps.

(1) Fitness evaluation for all particles' current positions. If necessary, $\vec{X}_{g\_best}$ and $\vec{X}_{p\_best_i}$ are updated.

(2) Update velocities. Based on these new values we calculate the velocity $\vec{v}_i(t+1)$ as a weighted sum of the current speed and the direction to the personal and the global best particles, see Fig. 3:

$$\begin{aligned}\vec{v}_i(t+1) = K \cdot [\vec{v}_i(t) + c_1 \cdot r_1 \cdot (\vec{X}_{p\_best_i} - \vec{X}_i(t)) \\ + c_2 \cdot r_2 \cdot (\vec{X}_{g\_best} - \vec{X}_i(t))].\end{aligned}$$

We calculate the velocity separately for each dimension of a solution vector $\vec{X_i}$ and use a constriction factor model proposed by Clerc et al. [26]. This model is fast to calculate, performs similarly, and is a good alternative to the inertia weight method. The factor c_1 is used to weight and prioritize the local optimum, c_2 is used to weight and prioritize the

Algorithm 2: The PSO algorithm.

```
Data: fitness function $f(\vec{X})$
Allocation $\vec{X}_{g\_best}$;
swarm ← generate initial set of $d$ allocations;
while (t < MaxGeneration) or (!stop-criterion) do
    foreach (particle $\vec{X}_i$) do
        evaluate fitness $f(\vec{X}_i)$;
        update $\vec{X}_{p\_best_i}$ and/or $\vec{X}_{g\_best}$;
        $\vec{v}_i(t+1)$ ← update velocity $\vec{v}_i(t)$;
        $\vec{X}_i(t+1)$ ← update position $\vec{X}_i(t)$;
return $\vec{X}_{g\_best}$
```

global optimum, and r_1 and r_2 are random values between 0 and 1 that are used to let the particles escape from local optima to explore new areas. K is the constriction coefficient and satisfies

$$K = \frac{2}{\left|2-(c_1+c_2)-\sqrt{(c_1+c_2)^2-4(c_1+c_2)}\right|}.$$

(3) Update positions. We derive the new valid position for each particle from a sum of its old position and its new velocity:

$$\vec{X}_i(t+1) = \vec{X}_i(t) + \vec{v}_i(t+1).$$

D. Cuckoo Search / Particle Swarm Optimization (CS/PSO)

The hybrid of CS and PSO [25] combines swarm intelligence and Lévy Flights to benefit from both concepts. The random walk of CS helps the algorithm to escape from local optima while the swarm intelligence of PSO accelerates the overall convergence by letting the cuckoos collaborate. The pseudo-code is given in Alg. 3.

Algorithm 3: The CS/PSO algorithm.

```
Data: fitness function $f(\vec{X})$, Lévy Parameter $\alpha$,
unfit fraction $p$;
Allocation $\vec{X}_{g\_best}$;
nests ← generate initial set of $d$ allocations;
swarm ← generate initial set of $e$ allocations;
while (t < MaxGeneration) or (!stop-criterion) do
    $cuckoo$ ← swarm.getRnd();
    Allocation $\vec{X}'$ ← $\vec{X}_{cuckoo}$.LévyFlight($\alpha$);
    $n$ ← nests.getRnd();
    if $f(\vec{X}') > f(\vec{X}_{cuckoo})$ then
        $\vec{X}_{cuckoo}$ ← $\vec{X}'$;
    if $f(\vec{X}') > f(\vec{X}_n)$ then
        $\vec{X}_n$ ← $\vec{X}'$;
    sort all allocations in mesh by fitness value;
    replace the fraction $p$ of the mesh with a low fitness
    value with random new and valid ones;
    foreach (cuckoo's allocation $\vec{X}_i$) do
        evaluate fitness $f(\vec{X}_i)$;
        update $\vec{X}_{p\_best_i}$ and/or $\vec{X}_{g\_best}$;
        $\vec{v}_i(t+1)$ ← update velocity $\vec{v}_i(t)$;
        $\vec{X}_i(t+1)$ ← update position $\vec{X}_i(t)$;
return $\vec{X}_{g\_best}$
```

The key idea of the algorithm is that not a single cuckoo bird performs Lévy Flights on a randomly chosen egg but a swarm of cuckoo birds is used to find new solutions. Furthermore, we also have a population of bird's nests, each of which holds a solution $\vec{X}$. In contrast to CS each cuckoo bird in the swarm represents a solution $\vec{X}$ as well. The swarm of cuckoo birds follows the rules defined by the PSO algorithm. In each generation the swarm's birds move as defined by the PSO algorithm, and a single randomly selected cuckoo additionally performs a Lévy Flight on its allocation to create a new egg $\vec{X}'$. This is done in order to escape from local optima and to speed up convergence. This way, the cuckoo's chance to put a better egg into its victim's nest improves.
A time-discrete loop performs the following three steps.

(1) Cuckoo selection. In every generation we randomly choose a cuckoo and its allocation $\vec{X}_{cuckoo}$ from the swarm and perform a Lévy Flight on its solution vector to create a new solution $\vec{X}'$. We evaluate the fitness of the result $\vec{X}'$.

(2) Nest selection. Next, we randomly choose a nest n. The fitness $f(\vec{X}_n)$ is evaluated and if it is worse than $f(\vec{X}')$, we replace $\vec{X}_n$ with $\vec{X}'$, otherwise we discard $\vec{X}'$.

(3) Selection of fittest. At the end of each iteration a fraction p of nests with low-fitness distributions is replaced by random new ones and the cuckoo swarm's positions and velocities are updated, as are the personal and global best solution vectors.

To adapt this algorithm to our specific scenario, we again perform the Lévy Flight only on a subset of dimensions of a solution $\vec{X}$. This again improves the local search due to the interdependent nature of the dimensions of $\vec{X}$. By limiting the random walk we prohibit overly random new solutions $\vec{X}'$, as a change in one dimension can have a ripple effect on the quality of the allocation of other event detectors to their host machines. Furthermore, we also update the cuckoos' personal bests and the swarm's global best if a Lévy Flight results in an allocation with better fitness. Additionally, to simplify the parameterization of the swarm's behavior, we again use the constriction factor model to update the swarm's velocity. Analog to CS, CS/PSO works best with a small value for the fraction p.

VI. Evaluation

To evaluate the four heuristics we created a benchmark scenario consisting of 30 event detectors that process synthetic event data streams on 6 nodes. These 6 VMs (LinuxContainer instances, each with a 2 GHz CPU core, 1GB of main memory and 1GBit virtual network) run on a server with 2 Intel Xeon X5675 Six Core CPUs and 64GB of main memory, and are linked by ns-3, a network simulator [27], to implement the virtual topology shown in Fig. 4. Every 60s the synthetic event detectors, symbolized as colored shapes in Fig. 4, change their subscription patterns. They switch between the two situations shown in Fig. 5. The numbers

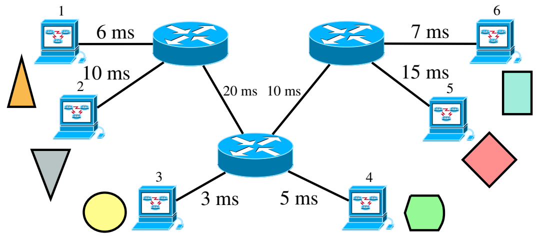
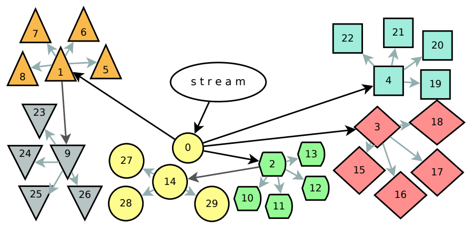


Fig. 4. Topology of virtual network environment.

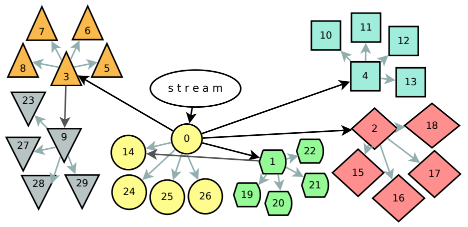
denote the event detector IDs. The different colors and shapes denote the *optimal* allocations of event detectors to the 6 nodes, if not more than 5 event detectors may run per node. The arrows show the direction of events detected/published. The optimal allocations are different for the two phases. They have been determined a-priori by means of an exhaustive search.

To evaluate the heuristics, we feed a synthetic event stream to the root event detector 0, and analyze the flood of events along the event hierarchy. The heuristics then minimize the number of events that are sent over network links with higher latency by migrating event detectors to nodes with lower latency, or by grouping highly dependent event detectors to the same node. Therefore, the heuristics minimize the *average latency* between any two collaborating event detectors, as more events are routed locally on the processing node or via links with lower latencies. The average latency is the arithmetic mean of all latencies of the event detectors' subscriptions over the network, measured at runtime. Local subscriptions have a latency of 0.

We evaluate the heuristics' performances and convergence behaviors by comparing the average latencies they achieve to the optimal allocations. We discuss each algorithm's performance separately and present only their essential evaluations for clarity. The number of iterations/generations for all



(a) Optimal allocation for phase 1.



(b) Optimal allocation for phase 2.

Fig. 5. Subscription patterns of event detectors.

heuristics is set in a way to limit the runtime to 3 seconds. This is needed for our low-latency event based system with changing dependencies and workloads. Additionally, before each optimization the heuristics measure the event count and latencies for 2 seconds. In Fig. 6 the straight black line shows the average latency of the optimal allocations found a-priori (Fig. 5). The second straight line in each of the diagrams in Fig. 6 shows what the heuristics can achieve when configured with the parameter set that works best. In addition we show the performance that the heuristics achieve with other parameter sets. Every 60 seconds the subscription pattern of the event detectors switches between the phases. While this does not affect the optimal/manual allocation, the heuristics face a peak of latency before they dynamically adapt. We start the measurements after 50s because initially detectors are distributed randomly.

A. Genetic Algorithm (GA)

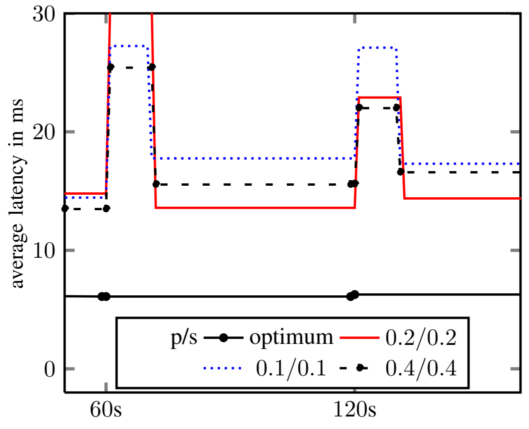
The parameters of the GA are (1) the percentage p of the solutions that perform crossover and (2) the percentage s of a subsequence of the solution vector $\vec{X}$ that is exchanged between two randomly chosen solutions $\vec{X_a}$ and $\vec{X_b}$ in the crossover step. We use a fixed population with a size of 80 solution vectors as recommended in [22]. Fig. 6(a) shows the average latency of all event detectors that the GA achieves with three parameter sets p/s. With p=0.2 and s=0.2 the GA outperforms other parameter sets. The number of generations has been set to $35,000$.

After the first phase change, the GA with the best parameters improves the event detector allocation by 56%, i.e., the average latency drops from 30.8ms to 13.6ms. This is still considerably slower than the optimal average latency (6.1ms). After the next phase change the best GA again needs to run twice before it adapts. It achieves an average latency of 14.4ms which is more than twice the optimum (6.27ms). Although the GA keeps monitoring and searching, it does not find better allocations. Therefore there are no reallocations of event detectors and no extra dots on the line in the diagram. Other parameter sets, e.g., p=0.4, s=0.4, exhibit a slower convergence and end up with slower latencies. With higher values of the parameters p or s, the GA often misses good solutions that are close to the population's solution vectors. With lower values the GA stagnates at local optima and does not explore new solutions.

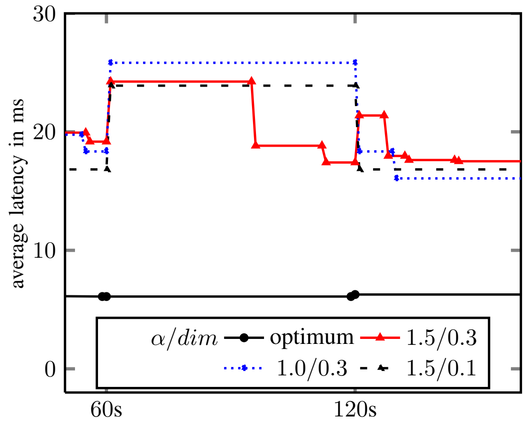
B. Cuckoo Search (CS)

CS has two relevant parameters (1) the Lévy Flight's distribution α that influences the step length of the random walk. With higher values for α the CS is more likely to generate small variations only instead of far leaps. Parameter (2) is the fraction of dimensions dim on which the Lévy Flight is performed. Preliminary experiments suggest that the fraction p of abandoned nests is not significant. The reason is that randomly generated results with a high fitness are unlikely. We set the population size to 15 nests [23].

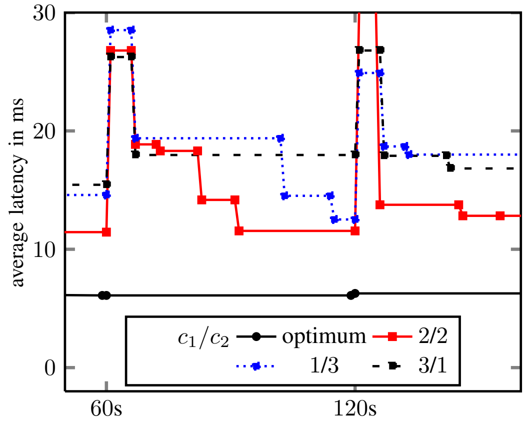
Fig. 6(b) shows the performance of the CS algorithm. Of all



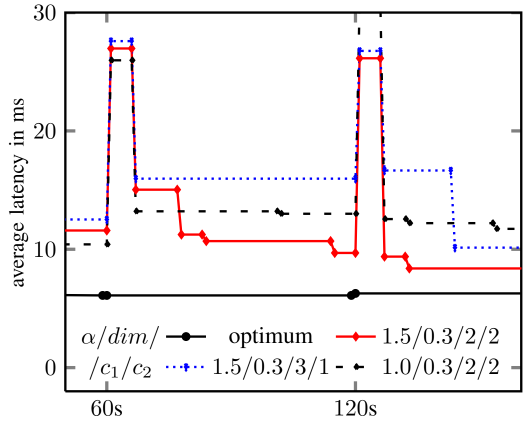
(a) GA: % crossover p, % substitution s.



(b) CS: Lévy Parameter α, affected dim.



(c) PSO: personal and swarm weights, c_1 and c_2.



(d) CS/PSO: Parameters α, dim, c_1, and c_2.

Fig. 6. Dynamic latency adaption achieved by GA, CS, PSO and CS/PSO.

evaluated parameter sets, the parameter set α=1.5 and dim=0.3 converges best. Unfortunately it takes 95 seconds to find an event detector allocation that slightly improves the average latency. Just before the phases change again, the best found allocation has an average latency of 17.4ms, which is far from the optimum (6.1ms).

One other parameter set in the diagram shows the effect of a smaller Lévy parameter α (1.0 instead of 1.5). Due to the resulting far leaps the solution space exploration misses nearby good allocation vectors. Potentially better allocations around local optima are not explored efficiently. Even if the effect is not as drastic after the second phase change, the resulting latencies are still worse than what the GA found. The other variation of parameters, i.e., α=1.5 and dim=0.1, exhibits similarly bad results. They suggest that small values for dim should be avoided.

The CS with the parameter set that adapts best to the system's dynamics still performs considerably worse than the GA.

C. Particle Swarm Optimization (PSO)

Recall that the the velocity formula made use of each particle's local/personal best position and the swarm's global best solution, weighted by c_1 and c_2 respectively. We evaluated PSO for different sets of these weights, using a population of 80 particles and 27,000 iterations, which preliminary experiments suggested as suitable numbers.

Fig. 6(c) shows that the best PSO with equal weights almost continuously optimizes the allocations after the first phase change until it settles at an average latency of 11.5ms, which is close to the optimum (6.1ms). After the second phase change depending on the parameter set, PSO improves the event detectors' allocations after just one run by between 24% and 54%. With equal weights c_1 and c_2 PSO reaches an average latency of about 13ms while the parameter sets with unmatching weights end up around 18ms.
With unmatching weights, the swarm's particles either stagnate at their respective local best allocation ($c_1 > c_2$) or the swarm quickly degenerates to the global best ($c_1 < c_2$). PSO only works well if the personal weight c_1 and the swarm weight c_2 match, e.g., c_1=c_2=2.

PSO achieves much better latencies than both the GA and the CS (11.5ms in the first phase and 12.8ms phase 2). It also exhibits better convergence.

D. Cuckoo Search / Particle Swarm Algorithm (CS/PSO)

The hybrid CS/PSO combines the CS and the PSO parameters. As before we use 15 nests and we set the swarm's population size to 80. Since the hybrid is computationally slightly more complex than PSO, only 25,000 iterations fit into the 3s optimization window.

Fig. 6(d) shows the performance of the best parameter set we found and two additional sets that we use to explain the results. After the first phase changes, the best CS/PSO already reduces the average latency to 13.2ms in its first run. Eventually it finds an allocation with 9.7ms average latency, just 3.6ms above the optimum of 6.1ms. After the second phase change, it performs even better.

As with the PSO, the weights c_1 and c_2 ought to be equal to maintain a balance between stagnation and degeneration. Small steps in the Lévy Flight applied to more dimensions work better than large leaps on fewer dimensions. The PSO's swarm, as a part of the hybrid algorithm, is well suited to conduct global searches. The cooperation of the swarm's particles speeds up global searches towards (local) optima. The local search around allocations near the local optimum is improved by the Lévy Flights. By slightly varying solution vectors, the CS/PSO hybrid explores immediate neighbors, even if the allocations' fitness is close to an optimum.

In comparison, the CS/PSO hybrid algorithm clearly outperforms the other heuristics. Whereas the GA and CS stay far away from the optimum, PSO and CS/PSO both approach the optimal detector allocation. As Fig. 6 shows, the CS/PSO hybrid performs better than the PSO heuristics, and it converges sooner and gets closer to the optimum.

VII. Conclusion

Evolutionary algorithms can be exploited to optimize the allocation of event detectors in a distributed event-based system at runtime. In our system a centralized unit collects runtime information and latencies, triggers the heuristics, and sends migration commands to the respective nodes.

We evaluated four heuristics and showed that they all reduce network latencies and hence enhance the system's average detecting latency. The Particle Swarm Optimizer outperforms both a Genetic Algorithm and a Cuckoo Search. A combination of Cuckoo Search and Particle Swarm Optimization performs best. Multi-swarm optimization may further improve the heuristic.

Acknowledgements

This work is supported by the Fraunhofer Institute for Integrated Circuits IIS whose RTLS called RedFIR we have used. We are grateful to the researchers at the Fraunhofer IIS for sharing their work and system with us.

References

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proc. 21th ACM Symp. Principles Database Systems*, (Madison, WI), pp. 1–16, 2002.

[2] P. P. Jonker, J. Caarls, W. J. Bokhove, W. Altewischer, and I. T. Young, "RoboSoccer: autonomous robots in a complex environment," in *Proc. 3nd Intl. Conf. Image and Graphics*, (Hefei, China), pp. 47–54, 2002.

[3] Y. Xing, S. Zdonik, and J.-H. Hwang, "Dynamic load distribution in the borealis stream processor," in *Proc. 21st Intl. Conf. Data Eng.*, (Tokyo, Japan), pp. 791–802, 2005.

[4] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker, "Fault-tolerance in the Borealis distributed stream processing system," in *Proc. Intl. Conf. Management of Data*, (Baltimore, MD), pp. 13–24, 2005.

[5] D. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The design of the Borealis stream processing engine," in *Proc. 2nd Conf. Innovative Data Systems Research*, (Asilomar, CA), pp. 277–289, 2005.

[6] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. Zdonik, "Providing resiliency to load variations in distributed stream processing," in *Proc. Intl. Conf. Very Large Data Bases*, (Seoul, Korea), pp. 775–786, 2006.

[7] P. R. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. I. Seltzer, "Network-aware operator placement for stream-processing systems," in *Proc. 22nd Intl. Conf. Data Engineering*, (Atlanta, GA), pp. 49–60, 2006.

[8] M. Akdere, U. Çetintemel, and N. Tatbul, "Plan-based complex event detection across distributed sources," *Proc. VLDB Endow.*, vol. 1, pp. 66–77, 2008.

[9] B. Liu, M. Jbantova, and E. A. Rundensteiner, "Optimizing state-intensive non-blocking queries using run-time adaptation," in *Proc. 23rd Intl. Conf. Data Eng. Workshop*, (Istanbul, Turkey), pp. 614–623, 2007.

[10] A. K. Yeung Cheung and H.-A. Jacobsen, "Dynamic load balancing in distributed content-based publish/subscribe," in *Proc. 7th Intl. Conf. Middleware*, (Melbourne, Australia), pp. 141–161, 2006.

[11] G. T. Lakshmanan, Y. G. Rabinovich, and O. Etzion, "A stratified approach for supporting high throughput event processing applications," in *Proc. 3rd Intl. Conf. Distributed Event-Based Systems*, (Nashville, TX), pp. 5:1–5:12, 2009.

[12] L. Ouyang and Q.-p. Guo, "A dynamic load balancing technique of distributed stream processing system," in *Proc. 2nd Intl. Conf. Future Generation Communication and Networking Symposia*, (Hainan Island, China), pp. 52–57, 2008.

[13] A. Gounaris, C. Yfoulis, and N. Paton, "An efficient load balancing LQR controller in parallel database queries under random perturbations," in *Intl. Conf. Control Appli.*, (St. Petersburg, Russia), pp. 794 –799, 2009.

[14] Y. Zhou, B. C. Ooi, K.-L. Tan, and J. Wu, "Efficient dynamic operator placement in a locally distributed continuous query system," in *Proc. Intl. Conf. On the Move to Meaningful Internet Systems*, (Montpellier, France), pp. 54–71, 2006.

[15] Y. Zhou, K. Aberer, and K.-L. Tan, "Toward massive query optimization in large-scale distributed stream systems," in *Proc. 9th Intl. Conf. Middleware*, (Leuven, Belgium), pp. 326–345, 2008.

[16] C. Lumezanu, S. Bhola, and M. Astley, "Utility optimization for event-driven distributed infrastructures," in *Proc. 26th Intl. Conf. Dist. Comp. Systems*, (Lisboa, Portugal), pp. 24–33, 2006.

[17] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas, "Operator scheduling in data stream systems," *The VLDB Journal*, vol. 13, no. 4, pp. 333–353, 2004.

[18] A. Keren and A. Barak, "Opportunity Cost Algorithms for Reduction of I/O and Interprocess Communication Overhead in a Computing Cluster," *IEEE Trans. Parallel Distrib. Syst.*, vol. no. 14, pp. 39–50, Jan. 2003.

[19] L. Amar, A. Barak, E. Levy, and M. Okun, "An On-line Algorithm for Fair-Share Node Allocations in a Cluster," in *Proc. 7th IEEE Intl. Symp. Cluster Computing and the Grid*, (Washington, DC), pp. 83–91, 2007.

[20] R. Khanna, H. Liu, and H.-H. Chen, "Self-Organization of Sensor Networks Using Genetic Algorithms," in *IEEE Intl. Conf. Communications*, (Istanbul, Turkey), pp. 3377–3382, 2006.

[21] K. M. Sim and W. H. Sun, "Ant colony optimization for routing and load-balancing: survey and new directions," *Trans. Sys. Man Cyber. Part A*, vol. 33, pp. 560–572, Sept. 2003.

[22] Y.-C. Hou and Y.-H. Chang, "A new efficient encoding mode of genetic algorithms for the generalized plant allocation problem," *J. Inf. Sci. Eng.*, vol. 20, no. 5, pp. 1019–1034, 2004.

[23] X.-S. Yang and S. Deb, "Cuckoo search via lévy flights," in *Proc. World Cong. Nature & Biologically Inspired Comp.*, (Coimbatore, India), pp. 210–214, 2009.

[24] J. Kennedy and R. C. Eberhart, "Particle swarm optimization," in *Proc. Intl. Conf. Neural Networks*, (Perth, Australia), pp. 1942–1948, 1995.

[25] A. Ghodrati and S. Lotfi, "A hybrid CS/PSO algorithm for global optimization," in *Proc. 4th Asian Conf. Intelligent Inf. and Database Systems*, (Kaohsiung, Taiwan), pp. 89–98, 2012.

[26] M. Clerc and J. Kennedy, "The particle swarm - explosion, stability, and convergence in a multidimensional complex space," *IEEE Trans. Evolutionary Computation*, vol. 6, no. 1, pp. 58–73, 2002.

[27] Nsnam, "Network Simulator 3, http://www.nsnam.org/."