# Towards Self-Optimization in HPC I/O

Michaela Zimmer and Julian Martin Kunkel and Thomas Ludwig

University of Hamburg, Germany *
michaela.zimmer@informatik.uni-hamburg.de

**Abstract.** Performance analysis and optimization of high-performance I/O systems is a daunting task. Mainly, this is due to the overwhelmingly complex interplay of internal processes while executing application programs. Unfortunately, there is a lack of monitoring tools to reduce this complexity to a bearable level. For these reasons, the project Scalable I/O for Extreme Performance (SIOX) aims to provide a versatile environment for recording system activities and learning from this information. While still under development, SIOX will ultimately assist in locating and diagnosing performance problems and automatically suggest and apply performance optimizations.
The SIOX knowledge path is concerned with the analysis and utilization of data describing the cause-and-effect chain recorded via the monitoring path. In this paper, we present our refined modular design of the knowledge path. This includes a description of logical components and their interfaces, details about extracting, storing and retrieving abstract activity patterns, a concept for tying knowledge to these patterns, and the integration of machine learning. Each of these tasks is illustrated through examples. The feasibility of our design is further demonstrated with an internal component for anomaly detection, permitting intelligent monitoring to limit the SIOX system's impact on system resources.

**Keywords:** Parallel I/O, Machine Learning, Self-Optimization

## 1 Introduction

While processor performance has been blessed with continual growth according to Moore's Law for decades now, performance increases of persistent storage media fall short of this by several orders of magnitude. To bridge this gap, I/O systems for high-performance computing (HPC) in particular have had to grow horizontally, requiring ever more layers of management infrastructure to control the ensuing complexity. Diagnosing such a system has become a task to challenge even experts. Parametrizing it for optimum performance requires intimate knowledge of every component, its optimization parameters and strategies and the interplay emerging when dozens to tens of thousands of them are combined.

* We want to express our gratitude to the „Deutsches Zentrum für Luft- und Raumfahrt e.V." as responsible project agency and to the „Bundesministerium für Bildung und Forschung" for the financial support under grant 01 IH 11008 A-C.

The vision of autonomous computing, as laid out by Kephart and Chess [1], promised to curb this complexity by marshalling the system itself to share into the effort. The SIOX Project [2] was initiated to realize that vision with respect to self-optimizing HPC-I/O systems. Continually monitoring performance and overhead, an I/O system instrumented for SIOX will autonomously detect problems and infer advantageous settings such as MPI hints, RAID stripe sizes and possible interactions between them. By adjusting its own level of reflexive activity to the situation, it will secure a net positive impact on overall efficiency.

This paper illustrates the structure of and techniques used in the knowledge processing sub-system enabling SIOX to achieve these goals.

In Section 2, we survey some other approaches to the problem, highlighting the differences to SIOX. We briefly outline the SIOX design with some definitions and an overview of the components involved in Section 3. Our main focus lies on Section 4, where we introduce the SIOX knowledge path, its modules and the concepts realized in its operation. Section 5 describes how the knowledge path can use modules for intelligent monitoring, before we conclude with a summary and some thoughts on future work in Section 6.

## 2 Related Work

Early approaches to system self-management relied on the direct classification of system state or behaviour to automatically diagnose problems or even enact optimization policies.

A typical proponent is the work of Madhyastha and Reed [3], comparing classification of I/O access patterns by feed-forward neural networks and by hidden Markov models. As results, higher level application I/O patterns are inferred and looked up in a table to determine the file system policy to set for the next accesses. The table, however, has to be supplied by an administrator implementing his heuristics.

Later approaches are marked by schemes to persist their results. Holding these in a database, problem analysis benefits from past diagnostic efforts, possibly even applying known repair actions to recognized problems. Here, we can divide systems according to whether they observe system state, recording metrics, or behaviour, tracing program execution.

Of those relying on program traces, Modani et al. [4] employ the call stacks reported by system failures as a search index to classify presumptive root causes.

`Magpie`, a system by Barham et al. [5], traces events under Windows, merging them according to pre-defined schemas specifying event relationships. Their causal chains are reconstructed, attributed to external requests via temporal joins over the event stream and clustered into models for the various types of workload observed. Deviations will point to anomalies deserving human attention.

Yuan et al. [6] combine system state and system behavior to identify the root causes of recurring problems. Tracing the system calls generated under Windows XP, they use support vector machines to classify the event sequences.

A presumptive root cause is identified, leaving the sequence – if flagged by a human as accurately diagnosed – available as eventual new training case for the classifier. The root cause description may include repair instructions, which, in some cases can be applied automatically.

Of the systems focussing on system metrics, `Cluebox` by Sandeed et al. [7] analyses logs for anomalies, pointing out the system counters most likely involved in the problem by principal feature analysis, ranking by decision trees and subsequent clustering.Expected latencies can now be predicted for new loads, detecting not only anomalies but also the counters most significantly deviating from par. No direct tracing or causal inference are needed, but once again, only hints for administrators are produced.

Cohen et al. [8] build on their previous work on *metric attribution*, identifying the low-level system metrics most significant for given classes of high-level system states using Tree-Augmented Bayesian Networks [9]. They collect system state information and combine the attributed metrics into *signatures*, clustering those belonging to the same problem class into *syndromes*.

In `Fa`, Duan et al. [10] define a system's base state by service level objectives; compliance constitutes health, violation failure. A robust data base of failure signatures is constructed from periodically sampled system metrics. New data is first classified against failure data annotated by a human, then, if necessary, against data clusters from healthy system states.

The one thing all of these systems have in common is the need for human intervention to benefit from the results, to apply the solutions to the problems identified or prepare the automated responses that some of the authors hint at.

While the work on invasive programming by Bungartz et al. (e.g. [11]) aims to automatically acquire and release resources according to the system's current requirements and capabilities, and can thus be regarded targeted at a similar problem as SIOX, it is mainly concerned with resource management. In contrast, SIOX will enable the existent management mechanisms to adapt their parameters to the system's current state and workload. As its scope is also limited to the I/O subsystem, the results of SIOX and invasive programming may very well complement each other in any system implementations.

## 3 The SIOX Approach

In comparison to described related work, SIOX covers full HPC I/O systems and aims to be applicable at all granularities and portable to platforms and across middleware and file systems, in a flexible and extensible hierarchy accommodating both bespoke heuristics and generic machine learning modules.

Under SIOX, a system will collect information on I/O activities on all instrumented levels, as well as relevant system information and metrics. An example I/O stack and the integration of SIOX is sketched in Figure 1, a scenario for potential activities is illustrated in Figure 2.

Continuous intelligent monitoring, possibly adapting to problem type, will detect more than mere service level violations; instead, locally diverse anomaly

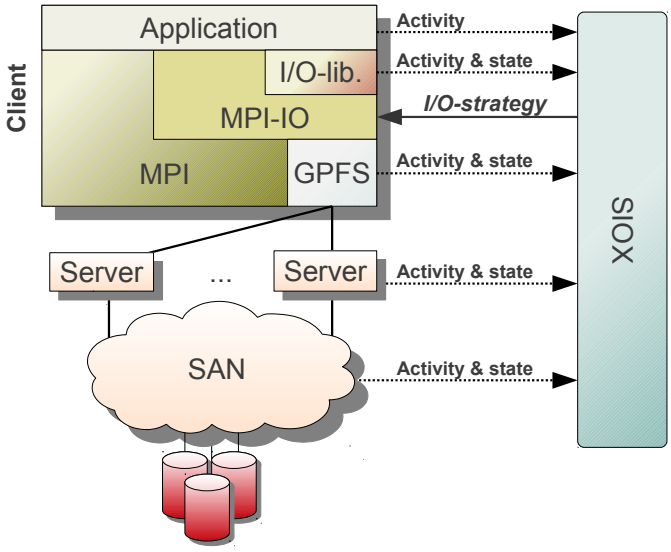


**Fig. 1.** Integration of SIOX into a traditional I/O-stack

conditions will be able to trigger fully automated (and learned) responses rather than provide mere pointers for human intervention. For this, SIOX combines on-line monitoring with off-line learning, joining state- to behaviour-based attributes and comparison to signatures as well as to a base-line. The recorded information will be analysed off-line to create and update a knowledge base holding optimized parameter suggestions for common or critical situations. During on-line operations, these parameters may be queried and used as pre-defined responses whenever such a situation occurs. Furthermore, the choice of responses to every situation is diverse, ranging from a mere log-level adjudication and detailed reports to facilitate human administration right to automated optimizations and problem solution strategies.

To cover the enormous multitude of possible hardware and software components that may be part of today's HPC system, SIOX takes an abstract view, as first suggested by Kunkel and Ludwig [12]. By virtue of this I/O path model (IOPm), a minimal set of components making up the system can be determined and classified according to their functionality, such as cache, block storage, network or an address translation of objects. A model graph of the system can be constructed covering the minimal cause-and-effect chain from application to storage device. Components may need to be represented by more than one of these elements, but this scheme warrants that any and all can be described by a very limited set of generic categories.

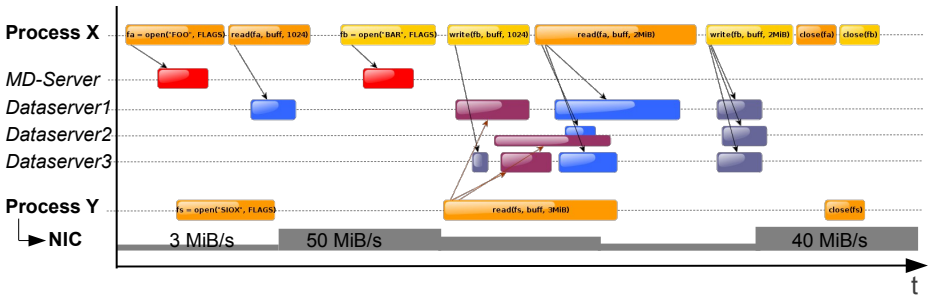


**Fig. 2.** Activity timelines of two processes and four file system servers – one system metric is provided for the entity executing process Y. Details for server activity and intermediate high-level I/O libraries are omitted

### 3.1 Definitions

The literature on computer systems is extensive, with many terms being used ambiguously. We therefore define some terms we will refer to in the following:

**Component** A hardware or software entity, such as a network switch, a hard disk drive, an application or a library.

**Entity** A logical subunit of a component aware of SIOX, using SIOX interfaces or reporting monitoring data to it. Its extent is defined according to its functionality, such as a software layer in a library or a cache in a server.

**Activity** A single, elementary operation on a single entity, possibly bundled with parameters, attributes and metrics pertaining to it. For example, an HDF5 `write()`, ATAPI `read()`, POSIX `fadvise()` or setting an MPI hint.

**System Information** The state of a component and the whole system, depending on the hardware characteristics and executed activities. It consists of **dynamic** information describing the system state, e.g., utilization of the components, and **static** information about hardware and software, such as device types, available resources and performance characteristics.

**Metric** A measurable or derivable quantity describing an aspect of the system, a component, an entity or an activity on any of the former, such as the number of Bytes written per second. An **Activity Metric** is a metric tied at report time to a specific activity, such as the execution time of a call. A **System Metric** is a metric that cannot be accurately assigned to a single activity, though usually influenced by them.

**System Statistics** Data derived or regularly sampled from system metrics. Some system metrics can only be measured periodically, either because the system only provides the difference over that interval, or because the value changes so fast that recording every variation would prove prohibitive. Examples are network and disk throughput and CPU utilization.

**Pattern** A set of activities linked by closeness in executing entity, time or causal relation. Also, a formal representation of such a set like a regular expression.

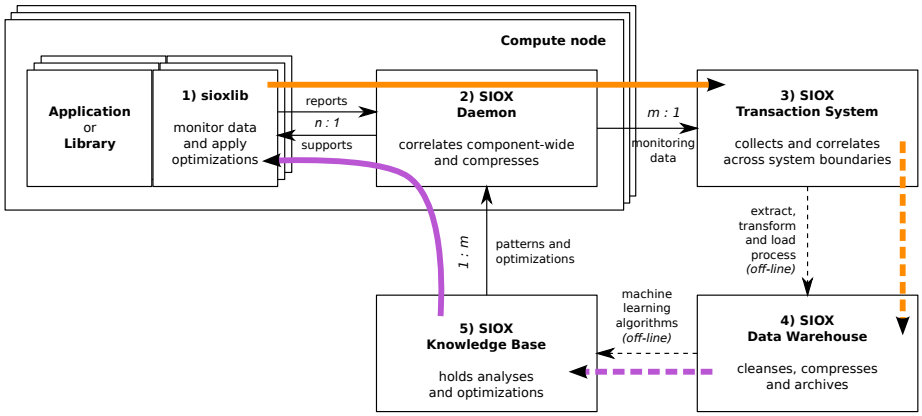


**Fig. 3.** The main components of the SIOX system and their cardinality. Monitoring data moves along the orange path; the knowledge path is shown in violet

**Situation** The current system state as observable by SIOX, including activities being executed and system metrics.

**History** A limited record of recent situations, including a sliding window of previous activities and observed performance statistics.

**Optimization Strategy** A scheme detailing how operations are executed by an entity, while not altering its functionality; e.g., a cache strategy. The strategies supported, if any, depend on the individual entity. The parameters for an optimization strategy, once chosen, influence operational performance according to the system's characteristics. **Optimization** is the choice of and selection of parameters for one or more optimization strategies.

**Log-Level** The level of detail of its history an entity reports to SIOX. A higher log-level will supply more details but also require more system resources.

**Response** Actions that may be taken upon certain situations occurring; they may also depend on the history. Typical examples are re-configuring an optimization or adjusting the log-level, both of which may also cascade through the entity's dependencies.

### 3.2 Components of the SIOX System

As first detailed in Wiedemann et al. [2], the SIOX system will be comprised of five primary components as shown in Figure 3:

1. The *sioxlib* linked to every component instrumented to work with SIOX.
2. One SIOX *daemon* per compute node. It acquires system information and aggregates and pre-processes activity logs, performance data and metrics. In a hierarchical set-up, additional daemons may serve as concentrator nodes to allow for scalability (see Section 4.4).

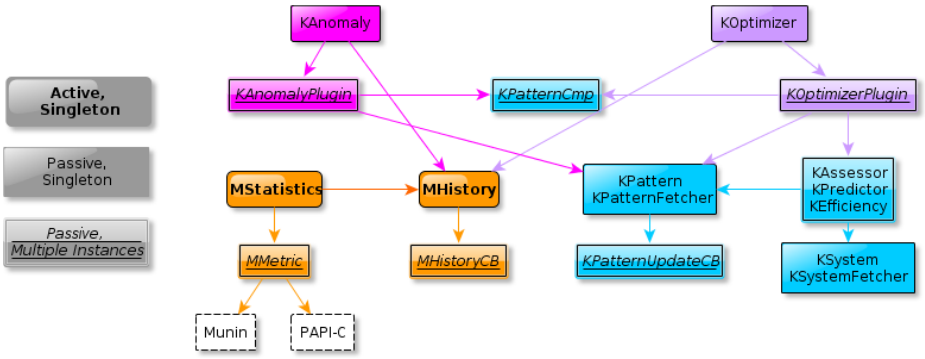


**Fig. 4.** SIOX modules involved in the knowledge path

3. A *transaction system* to collect, revise and concentrate the data submitted.
4. A *data warehouse* for long-time archiving of the monitoring data.
5. A *knowledge base* to hold the system information, as well as knowledge extracted from the data warehouse by off-line machine learning processes.

## 4 The SIOX Knowledge Path

The SIOX knowledge path (see Figure 3) begins at the data warehouse, holding representations of every activity logged, its parameters and the activity metrics that resulted from executing it. Off-line machine learning processes generalize activity sequences with comparable characteristics into patterns and potential responses to them (see Section 4.2). The results are stored in the knowledge base, together with records on system statistics for heuristics to assess observed vs. potential behaviour. A tree-shaped hierarchy of concentrator nodes connects knowledge base and entities to maintain full scalability (see Section 4.4). For every entity, the knowledge base will also hold information about the patterns most likely to occur there, those most likely to allow for effective optimizations, and those most warranting further investigation. Based on this, every entity will keep a local cache, updated regularly, of the patterns most crucial to its operation; this will conserve bandwidth and vastly improve response time. During operations, entities will match the cached patterns against situational and historic information to optimize their behaviour and control logging (see Section 5).

### 4.1 Modules and Interfaces

To allow for maximum flexibility while maintaining manageability, the SIOX knowledge path is built on a modular design. Not only does this permit us to replace a module with another implementation even after starting `sioxlib`, but many modules also allow concurrent loading of multiple plug-ins, all of which

might excel at different jobs. Of the modules involved in the knowledge path (shown in Figure 4), two take lead roles: `KOptimizer` and `KAnomalyDecision`.

`KOptimizer` directs various plug-ins implementing the `KOptimizerPlugin` interface, observing the entity's recent activity history and injecting a response if a relevant pattern occurs. Each of these plug-ins may be tailored to optimize certain aspects of the entity's operation, specializing on certain heuristics for a group of activities or a class of patterns; details are given in Section 4.2.

`KAnomaly` orchestrates various `KAnomalyPlugin` implementations, each of which will monitor the entity's operational metrics or activities to detect exceptional (good or bad) performance (see Section 5). For example, a plug-in might compare an activity's execution time with a model of system characteristics to estimate its performance. Should any of these report anomalous behaviour, the entity's log-level will be adjusted and relevant parts of the history will be dispatched along the monitoring path.

Plug-ins that implement one of the interfaces `KAssessor`, `KPredictor` or `KEfficiency` are basic building blocks performing evaluations on activities and higher granularities. They differ in the scope of information used for evaluation, as described in Section 4.3. By our modular design, an `KOptimizerPlugin` may use any evaluator, easing development and maintainability significantly.

The decision which, if any, pattern matches a given sequence of activities best will be delegated to `KPatternCmp`. Therewith, each plug-in can have its own matching rule, to rely on only certain activities, such as file I/O. `KSystem` manages an entity's knowledge about general system characteristics, such as neighbourhood topology and hardware specifications, accessing the knowledge base via `KSystemFetcher`. Likewise, `KPattern` administers the entity's local cache of patterns and responses regularly, updated via `KPatternFetcher`.

`MHistory` provides access to an entity's sliding history. Certain events, such as a new activity being entered into the history or the history reaching capacity, will trigger callback functions previously registered. The latter follow the form laid down in the `MHistoryCB` interface. Plug-ins for `KAnomaly` and `KOptimizer` will subscribe and automatically be called when the situation changes.

The statistical performance monitor for system metrics, encapsulated in `MStatistics`, performs regular updates on system statistics as supplied by the module `MMetric` and delivers the information to `MHistory`. To provide the metrics themselves, it in turn may interface various tools such as `PAPI-C`, or use existing plug-ins from Munin[1].

### 4.2 From Observation to Pattern and Response

The modules responsible for pattern creation and processing come in pairs of an *off-line machine learning plug-in* (OMLP) and an on-line module implementing `KAnomalyPlugin` and/or `KOptimizerPlugin`.

The OMLP will employ data mining algorithms to extract interesting (read: having performed exceptionally badly or well) sequences of activities and their

[1] http://munin-monitoring.org/

`open`(a, "F") `read`(a, 1024) `open`(b, "B") `write`(b, 1024) `read`(a, 2MiB) `write`(b, 2MiB) `close`(a) `close`(b)

**(a)** Observed activities (timing information omitted)

| pattern | advice |
|---|---|
| Sr()Sr()Sr() | seq & willneed(size) |
| O(ext="nc") | willneed(0, 20 KiB) |
| O(ext="dat") | noReuse & random |
| Rw(size < 4K){5} | noReuse & random |

**(b)** Table for an `fadvise()` plug-in

| pattern | buffer-size |
|---|---|
| O() | 4 MiB |
| W(size < 2 KiB){5} | 1 MiB |
| W(size < 4 MiB) W(size < 4 MiB) | 20 MiB |
| W(size ≥ 100 MiB) | direct-write |

**(c)** Table for a write-behind plug-in

**Fig. 5.** Exemplary patterns including key/value pairs in brackets and responses for two optimization plug-ins. Usage of symbols and key/value pairs are the responsibility of OMLP and the plug-ins

pertinent metric data from the data warehouse, cluster them and create a pattern representation of the set selected. It may simply concatenate the symbols representing the single activities and collect any additional attributes in a list of key-value pairs. It may first transform the sequence of activities, merging subsequences into single symbols or filtering some, deriving new attributes from the original ones in the process. As this process will run off-line, it does not influence a production system.

The result, in any case, will be a table of symbol strings in a form resembling regular expressions, and their attributes as key-value pairs attached to each symbol. Responses will be encoded as key-value pairs and appended to its list. This table will be personal to the OMLP and its on-line plug-ins, stored in the knowledge base and propagated to entities as applicable.

Figure 5 has some illustrative examples: In (a), activities observed at file level are shown; in (b) and (c), patterns and responses are listed for two plug-ins. One controls `fadvise()` while the other manages the size for a write-behind buffer (assuming such controls exist for the deployed parallel file system). Both plug-ins filter observed activities and translate them into symbols and relevant attributes. Timings are not given but are part of the attributes observed. The first plug-in converts sequential accesses to the symbols *Sr* and *Sw* for read and write, respectively. Whenever three sequential reads are observed, Line 1 in its table encodes the response *seq* and *willneed*(*size*) which translates to a `FADV_SEQUENTIAL` of the total file and an `FADV_WILLNEED` which pre-fetches the same amount of data as previously accessed. This plug-in also allows usage of the file extension to restrict matching patterns. The write-behind plug-in could use ranges to match defined file sizes, and prioritize patterns further down in the table. It dynamically adapts to the record size. When a file is first opened, Line 1 sets a default buffer size of 4 MiB; for 5 small write accesses, Line 2 reduces the buffer size. While these are simple examples, they demonstrate the power of the concept.

A plug-in registers a function with the `KOptimizer` or `KAnomaly` modules for these entities, to be invoked in a pre-defined order whenever a defined condition is met, for example, when new activity or metric is reported. It then compares current situation and history to its table, finds the best matches – if any – and decides which of those with attached responses will see them enacted.

Any entity or higher-order sub-system may employ many of these plug-in pairs, each working in turn on its own pattern table and each implementing a different specialization or heuristic. A simple pair concerned only with `MPI_File_open()` and optimizing them by setting appropriate hints is just as possible as a general catch-all pair, forming actual regular expressions over the set of all activity types possible at this entity. This allows for quick implementation of highly specialized heuristics as well as for classical machine learning algorithms which will even determine activities and attributes of interest.

### 4.3 Assessing Activities

To ease assessing system performance, we decided to employ three distinct basic function classes. Though each can operate on activities, patterns, components and other granularities right up to the whole system itself, usually utilizing information generated by its more specific variants, we will concentrate on activity evaluators to demonstrate the concepts.

**Assessors** resort to the local situation and history to evaluate a completed activity's perceived performance impact, returning a performance metric.
*Example*: A very simple model for file access or data transfer, computes an estimate for a device's transfer rate:

$$f_{\text{assess}}(\text{Device, Job}) = \frac{\text{Time(Job)}}{\text{Size(Job)}}$$

**Predictors** forecast an incomplete action's performance given the current situation and history. Its return value has to be comparable to an assessor's, thus following the same rules. Predictors' main usage is to estimate the benefit of possible responses for optimization plug-ins.
*Example*: The most simple and important one, the *historical predictor*, is an extrapolation of all assessments of the pattern's previous occurrences under comparable system states and loads.

**Efficiency evaluators** relate the action's actual performance as given by an assessor to the best performance possible on the given system and may take present conditions into account, e.g. faults in hardware. They compute a real number from the interval $[0; 1]$, but use additional information about the pattern's performance distribution to return one of $\{-1; 0; +1\}$, signifying exceptionally bad, reasonable and exceptionally good performance, respectively.
*Example*:

$$f_{\text{efficiency}} = \begin{cases} +1 & \text{if} \quad 0.9 < \text{eff} \\ 0 & \text{if} \quad 0.2 \leq \text{eff} \leq 0.9 \\ -1 & \text{if} \quad \text{eff} < 0.2 \end{cases}$$

with

$$\text{eff} = \frac{f_{\text{assess}}(\text{Device, Job})}{\text{SequentialTransferRate}(\text{Device})},$$

which will classify any observed transfer rate estimated at more than 90% or less than 20% of the maximum as exceptional, but more complex functions with more than three value ranges are well possible.

Like their base-level cousins, higher-order evaluators may compute simple weighted averages of their component activities' results, or follow any more sophisticated scheme.

### 4.4 Scalable Data Transport

To allow for the degree of scalability required in high-performance computing, SIOX implements a hierarchical subdivision scheme where daemons can act as concentrators, collecting, forwarding and disseminating data as needed. An example topology is illustrated in Figure 6. Daemons cache the information most important to a compute node, observing the local stream of activities as reported via the `sioxlib` and choosing and enacting any responses suggested by the knowledge available to them. They also keep a sliding history window of the activities, performance data and system statistics for each of their assigned entities.

Concentrators are similar but also function as local executive for the subtree of nodes assigned to them, just with more complex patterns combined of their child nodes' patterns. Additionally, each of the concentrators can deploy `KAnomaly` and `KOptimizer` plugins that operate on data generated by multiple entities and evaluate patterns spanning them.

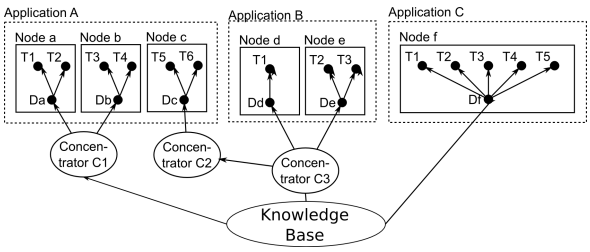


**Fig. 6.** Scalable data transport. SIOX-aware applications run on several compute nodes and fetch information from the knowledge base. Intermediate nodes in the graph cache fetched information to increase scalability

## 5 Intelligent monitoring

Every entity needs to determine which data about the current or past activities to report along the monitoring path, how much of the history to retain and and for how long, which system statistics to collect, at what interval to sample them and what additional derived metrics to compute. As transferring all reported activity over the network and into the transaction database has a huge performance impact, it is imperative to restrict the logging to exceptional behaviour. For SIOX to provide detailed logs on anomalies while remaining unobtrusive otherwise, it must be able to react to developments in system stability – quickly, sensitive to developments spanning whole sub-systems, and with a fine local granularity. Our scheme for intelligent monitoring allows for three ways to influence logging:

- A user can control logging on each component to manually correct situations for which the history window is not sufficient or for bootstrapping the system.
- Neighbouring entities such as the local daemon may request a change in log-level to propagate alerts and allow pattern analysis across components.
- Most importantly, the various `KAnomalyPlugin` implementations may flag anomalies, influencing log-levels in the process. Of course, a plug-in detecting a return to normal behaviour will use the same mechanism to give the all-clear and decrease the log-level again.

As described in Section 4.1, the `KAnomaly` module permits implementations of `KAnomalyPlugin` to register as callback functions, calling them in a pre-defined order whenever a new activity is reported. Each of them may inspect current activity, situation and history to decide whether they constitute an anomaly, i.e., unusually good or bad system performance.

A typical example would be a system metric entering exceptional range, though defining these often is anything but trivial. A low processor load, for instance, may signify either a balanced system coping well with its workload, or a severe unbalance, leaving some components idling and others congested.

Should any plug-in flag the current state, it will cause the history to be dispatched along the monitoring path for later analysis. Additionally, it may adjust local log-levels, which may even be propagated to other entities.

The usual fate of the data thus recorded is to serve as a lesson to SIOX. Distilled into patterns (see Section 4.2) bearing the response "raise log-level" – the other typical example for anomaly detection – it will alert SIOX whenever the conditions leading up to the error are observed. This greatly benefits the analysis of recurring failures, as every instance provides some valuable new insights into the problem's preconditions and, ultimately, cause. In this sense, SIOX will intelligently direct its own analytical efforts where they are needed most.

## 6 Conclusion and Future Work

Our vision for the SIOX project is a system that will collect and analyse activity patterns and performance metrics in order to assess current and possible system

performance, locate and diagnose problems and suggest solutions and improvements. In this paper, we have described the design of and information flow along the SIOX knowledge path. We have shown its logical layout and how its modules interact to perform their various tasks. Examples for our approach to knowledge representation have been given to demonstrate its applicability and a scheme for intelligent logging presented.

Opportunities for future work abound. Different platforms pose different problems even though SIOX is designed to be portable to all major operating and file systems. In the consortium, we are currently working on a first prototype for GPFS and MPI-IO which will rely on simple plug-ins. However, the various modules imaginable for pattern creation and matching, for anomaly detection and optimization as well as the machine learning algorithms offer a rich field for researchers and system administrators alike.

## References


1. Kephart, J., Chess, D.: The Vision of Autonomic Computing. Computer **36**(1) (January 2003) 41–50
2. Wiedemann, M.C., Kunkel, J.M., Zimmer, M., Ludwig, T., Resch, M., Bönisch, T., Wang, X., Chut, A., Aguilera, A., Nagel, W.E., Kluge, M., Mickler, H.: Towards I/O Analysis of HPC Systems and a Generic Architecture to Collect Access Patterns. Computer Science - Research and Development **1** (2012) 1–11
3. Madhyastha, T., Reed, D.: Learning to Classify Parallel Input/Output Access Patterns. Parallel and Distributed Systems, IEEE Transactions on **13**(8) (August 2002) 802–813
4. Modani, N., Gupta, R., Lohman, G., Syeda-Mahmood, T., Mignet, L.: Automatically Identifying Known Software Problems. In: Data Engineering Workshop, 2007 IEEE 23rd International Conference on. (April 2007) 433–441
5. Barham, P., Donnelly, A., Isaacs, R., Mortier, R.: Using Magpie for Request Extraction and Workload Modelling. Proceedings of the 6th Symposium on Opearting Systems Design and Implementation **6** (2004) 259–272
6. Yuan, C., Lao, N., Wen, J.R., Li, J., Zhang, Z., Wang, Y.M., Ma, W.Y.: Automated Known Problem Diagnosis with Event Traces. In: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006. EuroSys '06, New York, NY, USA, ACM (2006) 375–388
7. Sandeep, S.R., Swapna, M., Niranjan, T., Susarla, S., Nandi, S.: CLUEBOX: a Performance Log Analyzer for Automated Troubleshooting. In: Proceedings of the First USENIX conference on Analysis of system logs. WASL'08, Berkeley, CA, USA, USENIX Association (2008)
8. Cohen, I., Zhang, S., Goldszmidt, M., Symons, J., Kelly, T., Fox, A.: Capturing, Indexing, Clustering, and Retrieving System History. SIGOPS Oper. Syst. Rev. **39**(5) (October 2005) 105–118
9. Cohen, I., Goldszmidt, M., Kelly, T., Symons, J., Chase, J.S.: Correlating Instrumentation Data to System States: a Building Block for Automated Diagnosis and Control. In: Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation – Volume 6. OSDI'04, Berkeley, CA, USA, USENIX Association (2004)

10. Duan, S., Babu, S., Munagala, K.: Fa: A System for Automating Failure Diagnosis. In: Data Engineering, 2009. ICDE '09. IEEE 25th International Conference on. (29 2009-April 2 2009) 1012–1023
11. Bader, M., Bungartz, H.J., Gerndt, M., Hollmann, A., Weidendorfer, J.: Invasive programming as a concept for HPC. In: Proc. of the 10h IASTED Int. Conf. on Parallel and Distr. Comp. and Netw, PDCN. (2011)
12. Kunkel, J., Ludwig, T.: IOPm – Modeling the I/O Path with a Functional Representation of Parallel File System and Hardware Architecture. In: PDP 2012, Munich Network Management Team, IEEE (2012)