ScrubJay: Deriving Knowledge from the Disarray of HPC Performance Data

Alfredo Giménez*
Lawrence Livermore National Laboratory
gimenez1@llnl.gov

Todd Gamblin
Lawrence Livermore National Laboratory
tgamblin@llnl.gov

Abhinav Bhatele
Lawrence Livermore National Laboratory
bhatele@llnl.gov

Chad Wood†
Lawrence Livermore National Laboratory
wood67@llnl.gov

Kathleen Shoga
Lawrence Livermore National Laboratory
shoga1@llnl.gov

Aniruddha Marathe
Lawrence Livermore National Laboratory
marathe1@llnl.gov

Peer-Timo Bremer
Lawrence Livermore National Laboratory
ptbremer@llnl.gov

Bernd Hamann
University of California, Davis
hamann@cs.ucdavis.edu

Martin Schulz‡
Lawrence Livermore National Laboratory
schulzm@in.tum.de

ABSTRACT

Modern HPC centers comprise clusters, storage, networks, power and cooling infrastructure, and more. Analyzing the efficiency of these complex facilities is a daunting task. Increasingly, facilities deploy sensors and monitoring tools, but with millions of instrumented components, analyzing collected data manually is intractable. Data from an HPC center comprises different formats, granularities, and semantics, and handwritten scripts no longer suffice to transform the data into a digestible form.

We present ScrubJay, an intuitive, scalable framework for *automatic* analysis of disparate HPC data. ScrubJay decouples the task of *specifying* data relationships from the task of *analyzing* data. Domain experts can store reusable transformations that describe relations between domains. ScrubJay also automates performance analysis. Analysts provide a query over logical domains of interest, and ScrubJay automatically derives needed steps to transform raw measurements. ScrubJay makes large-scale analysis tractable, reproducible, and provides insights into HPC facilities.

CCS CONCEPTS

• **General and reference → Performance**; • **Information systems →** *Database query processing*;

KEYWORDS

HPC Performance Analysis,Facility Monitoring,Performance Tools

*Also with University of California.
†Also with University of Oregon.
‡Also with Technische Universität München.

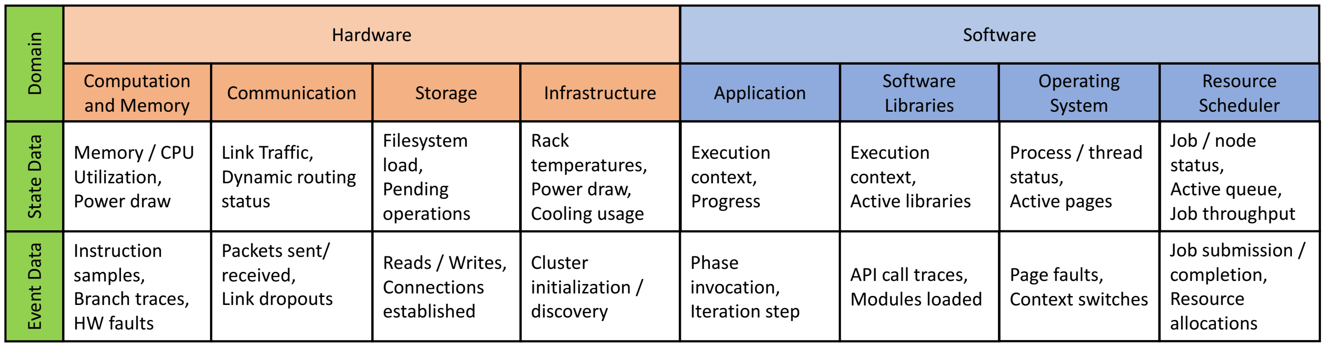
ACM Reference Format:
Alfredo Giménez, Todd Gamblin, Abhinav Bhatele, Chad Wood, Kathleen Shoga, Aniruddha Marathe, Peer-Timo Bremer, Bernd Hamann, and Martin Schulz. 2017. ScrubJay: Deriving Knowledge from the Disarray of HPC Performance Data. In *Proceedings of SC17*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3126908.3126935

1 INTRODUCTION

The performance of HPC applications is the product of more factors than ever before. A parallel application's runtime may vary greatly due to contention for the network, contention for the parallel filesystem, dynamic processor frequency scaling, data-dependent inputs, and many other factors. Often, application-level problems, such as the slow execution of a particular file or communication operation, are caused by contention at a lower level of the stack. Moreover, predicting how entire workloads will perform depends strongly on understanding the interaction of *all* constituent applications.

As supercomputers grow larger, and as compute performance increases relative to both network and filesystem performance, facilities become increasingly interested in how to optimize their systems for their workloads. Large machine procurement decisions have long been based on the performance of targeted benchmark workloads, but now facilities have begun to collect monitoring data on running systems *in production*. At our HPC center, we have begun to collect streaming data from network and filesystem counters, job queue logs, node-level performance counters, and application-level profiles. This results in tens of gigabytes per day, and we expect that as we instrument more systems at our facility, we may ingest multiple terabytes per day. To store and process this data, we use a dedicated *data cluster*, the scalable, "NoSQL" Cassandra database, and the Apache Spark data-parallel framework.

While our database can easily store the ingested data, and our parallel processing tools are known to scale to large cluster sizes, we have found that the main obstacle to large-scale facility analysis data is the complexity of connecting, aggregating, and comparing data from many heterogeneous sources. Nearly every HPC monitoring tool collects data in a different format, with its own



Domain	Hardware				Software			
	Computation and Memory	Communication	Storage	Infrastructure	Application	Software Libraries	Operating System	Resource Scheduler
State Data	Memory / CPU Utilization, Power draw	Link Traffic, Dynamic routing status	Filesystem load, Pending operations	Rack temperatures, Power draw, Cooling usage	Execution context, Progress	Execution context, Active libraries	Process / thread status, Active pages	Job / node status, Active queue, Job throughput
Event Data	Instruction samples, Branch traces, HW faults	Packets sent/ received, Link dropouts	Reads / Writes, Connections established	Cluster initialization / discovery	Phase invocation, Iteration step	API call traces, Modules loaded	Page faults, Context switches	Job submission / completion, Resource allocations

Figure 1: A non-exhaustive set of available HPC performance data sources (orange, blue), the kinds of data they produce (green), and the data collected (white).

sampling frequency and units. Establishing a common basis between values from different tools is difficult. Consider a set of CPU instruction samples, each annotated with latency and CPU id. We may also collect periodic counts of read and write events to the parallel filesystem. In order to determine whether IPC was affected by the utilization of the parallel filesystem, we need to associate specific instructions with filesystem events. In general, the raw data tables ingested by the system do not naturally expose these relationships. Existing solutions require us to encode them by hand in domain-specific scripts, but this requires significant knowledge and expertise from the analyst. Analysts may need to ask HPC administrators for information to accurately calculate this data correspondence. This is unscalable and error prone.

In this paper, we present ScrubJay, a general and scalable framework for analyzing big heterogeneous HPC performance data. ScrubJay is motivated by the increasing variety and volume of HPC performance data, and the need for scalable, reproducible ways to analyze it. ScrubJay separates the concerns of performance tool experts, system administrators, and performance analysts. Administrators and experts define the structure and semantics of collected data. They may also provide reusable transformation functions that can be applied to associate and attribute data to one another. Performance analysts can make queries on a *logical* taxonomy of fields, and ScrubJay will automatically derive and apply the necessary data processing operations, for an accurate and consistent result. Our contributions are as follows:

- An extensible annotation framework to describe data sources and the heterogeneous data they produce;
- a logical query engine that uses annotations to automatically infer relationships between data from different sources;
- scalable, parallel algorithms for mapping, transforming, and aggregating diversely typed distributed datasets;
- a language to describe data-parallel pipelines; and
- an implementation of the above concepts in ScrubJay.

In the remainder of this paper, we motivate our design for ScrubJay and describe show it greatly simplifies the task of performance analysis. ScrubJay automates the tedious aspects of performance analysis and enables "big data" analytics for HPC performance data.

2 HPC PERFORMANCE DATA DISARRAY

HPC performance data takes various forms depending on the source from which it is collected and the mechanism used to collect it. Sources include hardware components (e.g. CPU cores, network switches, racks) or software components (e.g. operating systems, software libraries, application code). The mechanisms to collect performance data include event counters, function tracing libraries, and sensors for capturing state information.

Typically, performance tools provide collection and analysis capabilities for a restricted domain of sources within the HPC ecosystem. Developer performance tools, such as HPCToolkit [1], VTune, and MemAxes [11], typically describe the domain of the application and on-node hardware, through instruction call-paths, aggregated call trees and hardware metrics. Other tools focus on the domain of the network and communication patterns, e.g. MPIP [27] and Ravel [14], and provide analysis capabilities for message-passing traces with complex dependencies. Orthogonal to both, facility-monitoring infrastructures provide valuable performance information at a high level such as power and temperatures across entire clusters or racks. While these data sources are connected behaviorally and often physically, discrepancies between their collected data representations and the domains over which they are defined make it exceedingly difficult to draw relations between their values.

Ideally, we should allow users to benefit from the capabilities of these different heterogeneous tools and the information they provide at various different levels, rather than attempt to replace them. Enabling their coexistence and cooperation requires providing common ground for storage, retrieval, and cross-integration. We have paved this common ground in our analysis framework. We enable scalable and distributed ingestion and storage of performance data on our dedicated data cluster and consolidate heterogeneous data sources using a common, generalized format. This common ground enables the advanced query and data integration capabilities we present in ScrubJay and thus expose new levels of analysis insight for HPC performance across the entire facility.

2.1 Data Sources and Collection Mechanisms

To define the space of available HPC performance data sources, we refine the broad categories of hardware and software components

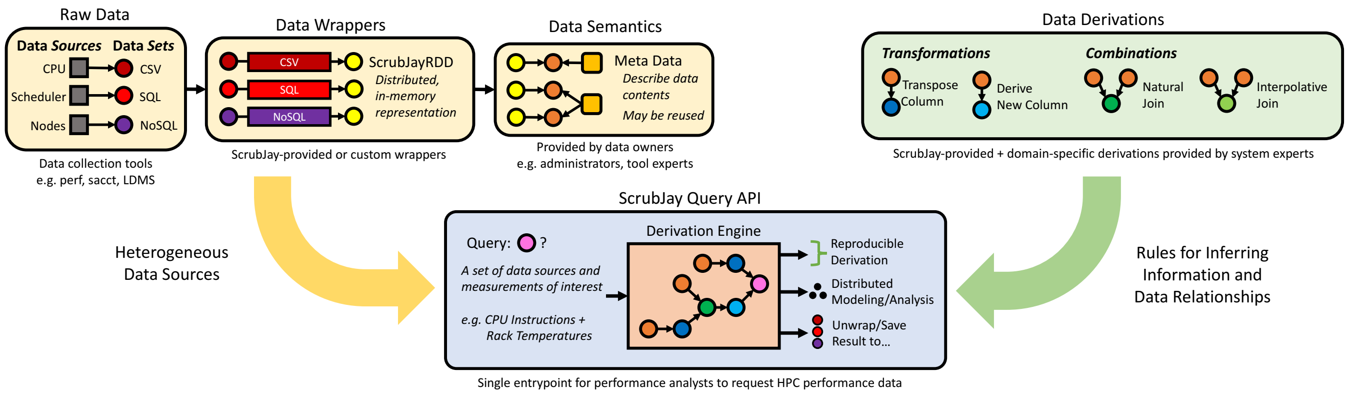

Raw Data
Data Sources
Data Sets
CPU
Scheduler
Nodes
CSV
SQL
NoSQL
Data collection tools e.g. perf, sacct, LDMS
Data Wrappers
ScrubJayRDD
Distributed, in-memory representation
ScrubJay-provided or custom wrappers
Data Semantics
Meta Data
Describe data contents
May be reused
Provided by data owners e.g. administrators, tool experts
Data Derivations
Transformations
Transpose Column
Derive New Column
Combinations
Natural Join
Interpolative Join
ScrubJay-provided + domain-specific derivations provided by system experts
Heterogeneous Data Sources
ScrubJay Query API
Query: ?
A set of data sources and measurements of interest
e.g. CPU Instructions + Rack Temperatures
Derivation Engine
Reproducible Derivation
Distributed Modeling/Analysis
Unwrap/Save Result to...
Single entrypoint for performance analysts to request HPC performance data
Rules for Inferring Information and Data Relationships

Figure 2: ScrubJay system overview. ScrubJay exposes performance data from multiple heterogeneous sources (yellow) by providing a framework to define how to parse different storage formats (data *wrappers*) and to describe the contents of data (data *semantics*). System experts describe rules and methods (green) for inferring information from a dataset (*transformations*) and inferring relationships between datasets (*combinations*). ScrubJay uses this information in a *derivation engine* to satisfy high-level queries from performance analysts (blue). The results may be shared as reproducible data processing pipelines, used in distributed modeling and analysis frameworks, or saved to different storage formats for analysis with other tools.

into subdomains. We also categorize data collection mechanisms as either event information or state information. Event information describes details about the occurrence of a single performance-related event, and state information describes the status of some resource at an instant in time. We represent a non-exhaustive set of commonly available HPC performance data sources and collection mechanisms within this taxonomy in Figure 1.

The top two headers represent the data source domains, and the entries represent the data collection mechanisms. The behaviors of any combination of these sources may significantly impact performance. For example, high network counter values may indicate a congested network due to a sudden increase in nodes contacting a parallel filesystem server. This increase may be due to multiple applications entering their checkpoint phases simultaneously. However, diagnosing such interconnected behaviors would require correlating data collected from network counters, filesystem events, and application phases. We can draw relations between these datasets from relations between their respective sources; in this case, by determining which nodes an application is running on, the filesystem servers it is connected to, and the network through which it is connected to them. This information is available through additional mechanisms: the job queue, the network routing tables and the hardware setup of the I/O infrastructure. There, the collected data serves not only as information to directly analyze, but as information with which to draw indirect relations between data sources. The latter also includes a host of static information, such as, which nodes reside on which racks, or which power meters measure which hardware components.

3 SYSTEM OVERVIEW

ScrubJay mitigates the disarray of HPC performance data by decoupling the collection, representation, and semantics of data. This modular design enables a user to analyze data collected from any number of heterogeneous tools and different data sources together.

ScrubJay converts each heterogeneous dataset into a common in-memory representation using a set of specified functions called data *wrappers*. These are either provided by ScrubJay or defined by users familiar with a particular collection format. This common representation is distributable, and thus we are able to scale both memory utilization and processing tasks for large datasets by distributing them over several nodes.

Data is useless without meaning, however. Thus, ScrubJay also provides a framework for defining the *semantics* of different data recordings. Semantics define what different elements of the dataset are; for example, whether some column in a table represents the time in which a recording was collected or a temperature reading. This information enables ScrubJay to determine explicit correspondences between datasets—e.g. whether a "node" column in a dataset describes the same value as a "NODEID" column in a different dataset—and defines the available operations ScrubJay can perform on some data—e.g. splitting a time span into several time stamps.

Data semantics provide a basis for *derivation* functions. Derivations define rules for inferring information from or computing relationships between datasets. We categorize types of derivations into *transformations*, which derive a new dataset from an existing one, and *combinations*, which combine two datasets into a merged result. Transformations include unit conversions and changes in the data representation, and combinations map elements from one dataset to elements of another.

With this information, we can create datasets that describe disparate data sources in relation to each other by applying a sequence of derivations on them. However, the set of possible derivation sequences is combinatorially huge, even infinite in some cases, and many of these sequences may not prove useful for analysis. Therefore, instead of enumerating the space of possible derivations, we developed a *derivation engine* to efficiently navigate this space and find a derivation sequence that provides relevant information for some specific analysis. The performance analyst specifies a set of data sources and measurements of interest, and ScrubJay finds an appropriate derivation sequence that yields a dataset exposing correlations between those sources and measurements. For example, if an analyst wants to determine whether an application's CPU instruction rates correlate with temperatures in the facility, s/he requests CPU instructions and rack temperatures, and ScrubJay returns a derivation sequence that yields a dataset containing a relation between CPU instructions and rack temperatures.

The resulting derivation sequences may then be either (1) stored without computing their results, such that different analysts may utilize the same derivation sequences, (2) computed in distributed memory and used for distributed modeling and analysis, or (3) unwrapped into a storage format, such as a CSV file or SQL table, for sharing or analyzing using other analysis tools. Figure 2 displays a schematic of ScrubJay's design.

ScrubJay's design effectively separates concerns such that tool experts define their custom storage formats, data owners and tool experts describe the semantics of collected datasets, system experts define how to derive information from existing data, and analysts need only to request a set of data sources and measurements for analysis. This design eliminates redundancy by storing the necessary knowledge for parsing and processing datasets in a reusable format and provides a baseline for reproducing the complex processing operations involved in finding correlations between different HPC performance data sources.

4 META ANNOTATION

ScrubJay's annotation framework makes it possible for the appropriate users to specify the formats, meanings, and rules for inferring information and relationships between heterogeneous datasets. These annotations comprise the knowledge base of ScrubJay, and once specified, they may be shared and reused. The three types of information ScrubJay requires are:

(1) A data *wrapper* defining how to convert some format into ScrubJay's in-memory representation
(2) A set of data *semantics* describing the data contents
(3) A set of data *derivations* that describe how to derive information from a dataset or a relation between two datasets

ScrubJay provides wrappers and derivations for most common cases as well as an modular library for defining custom extensions.

4.1 Data Wrappers

Internally, ScrubJay datasets are represented in a distributed in-memory data structure. This data structure is a type of resilient distributed dataset (RDD) as provided by Apache Spark and henceforth called a ScrubJayRDD. As such, it may be distributed over any number of nodes operating as Spark slaves, and any operations on it are lazily executed, meaning they are enqueued but not run until their results are explicitly requested.

ScrubJayRDDs consist of variable-length tuples with named elements of varied types. For example, a ScrubJayRDD row containing timestamped temperature readings from nodes may look like:

```
("timestamp" -> "Mon Mar 27 16:43:27 PDT 2017",
 "node_id" -> 5,
 "node_temp" -> 67.4)
```

As such, ScrubJay readily handles sparse data with heterogeneous values. This schema is general enough to handle all forms of data we have come across, and most with few modifications. Although compressed and hierarchically formatted data must be expanded and denormalized to fit this schema, we readily trade off memory for generality because we can easily distribute the data over a large available memory space.

The data wrapper is responsible for parsing data stored in some format and reading its contents into a ScrubJayRDD. ScrubJay provides a set of default wrappers for commonly used formats, such as Comma-Separated Value (CSV) files and NoSQL database tables. For custom formats, tool experts specify a function that takes any number of arguments and produces a ScrubJayRDD. Once wrapped, users define a set of semantics for the resulting data.

4.2 Data Semantics

While there are infinitely many ways to describe data, ScrubJay's semantics serve purely to provide a common language with which to define the derivation operations. Derivations may require comparing, aggregating, interpolating/extrapolating, or converting data. The methods for doing so vary based on the units and dimensions of the data. For example, 10 degrees Celsius is less than 20 degrees Celsius, but a node ID of 10 is not "less than" a node ID of 20. We also cannot compare recordings across different dimensions; a node ID of 10 does not compare to a temperature reading of 20. Furthermore, we can relate recordings from two datasets that describe the same *domains*; e.g. dataset A contains hardware faults for a CPU, and dataset B contains the instruction rate for the same CPU, so we can draw a relation between the faults in A and rates in B. However, if two data recordings describe the same *value*, such as the same temperature, we cannot infer that the recordings are related. We therefore define a semantic representation for different elements in a ScrubJayRDD based on their high-level types, their dimension, and whether they describe a domain or a value.

We store semantics as simple keyword tuples and map the names of elements in a ScrubJayRDD to these tuples. The semantics for the previous ScrubJayRDD example may look like:

```
("timestamp" ->
    ("relation type" -> "domain",
     "dimension" -> "time",
     "units" -> "datetime"),
 "node_id" ->
    ("relation type" -> "domain",
     "dimension" -> "compute nodes",
     "units" -> "identifier")
 "node_temp" ->
    ("relation_type" -> "value",
     "dimension" -> "temperature",
     "units" -> "degrees Celsius"))
```

The names of tuple values are keys, and their values consist of a *relation type*, *dimension*, and *units*. ScrubJay uses a semantic dictionary to look up the keywords stored here and enables appropriate operations on different data elements based on their definitions. Users specifying semantics may either use existing keywords or define new entries in this semantic dictionary.

Relation Type The relation type defines whether a data element describes a *domain* or a *value*. A domain is a descriptor for the resource being measured, e.g. a CPU core, a rack, or an application, in addition to the time or location it was measured. A value is the actual measurement, e.g. instruction rate or temperature. Note that the elapsed time of an application execution also constitutes a measurement, and therefore a value, thus elements may share the same dimension but not the same relation type.

Dimension We use the general definition of dimension: an aspect or attribute of something. This may include physical dimensions, such as space/time, and conceptual, such as the identity of a CPU.

The dimension also defines the space in which an element may be defined. Dimensions may be either *continuous* or *discrete* depending on whether values along them may be halved indefinitely, and they may be either *ordered* or *unordered* depending on whether values may be compared along them. Temperature is both continuous and ordered, event counts are discrete and ordered, and identifiers are discrete and unordered. ScrubJay looks up these properties of specified dimensions in its semantic dictionary and determines the appropriate operations for manipulating data elements. For example, we can interpolate time by averaging neighboring values, but we cannot take an average of two node identifiers.

Units Finally, we specify the units in which a measurement was recorded. This includes physical units such as degrees Celsius, degrees Fahrenheit, seconds, or minutes, and definitions of subspaces, such as time spans and time stamps. In order to support derived units, ScrubJay also recognizes compositions of different units, such as instructions per second, or lists of identifiers.

ScrubJay internally uses an advanced type system to operate on different units. Using the provided units for a data element, ScrubJay constructs a high-level object with the appropriate functionality. For example, seconds may be readily converted to minutes, and time spans may be expanded into a sequence of time stamps. As such, data manipulation operations are constrained by a set of valid operations defined for the data.

Semantic Dictionary Problems arise for both units and dimensions in cases where multiple different semantic keywords mean the same thing (synonyms) and where the same semantic keywords mean different things (homonyms). To mitigate the issue, we introduce a *dictionary* of available semantic keywords in which no synonyms or homonyms may exist. This dictionary defines the available units recognized by ScrubJay and the dimensions on which they may lie. For example, we may define "t_seconds" for units of time and "d_seconds" for angular measurement. Users may use ScrubJay's default semantic dictionary and/or define new entries; ScrubJay validates any loaded datasets against the active dictionary.

4.3 Data Derivations

These semantics provide the necessary information to determine what kinds of derivations may be performed on a dataset and how to perform them. Derivations are functions which take one or two semantically annotated datasets as input and produce a new dataset, with new semantics, as output. These functions either produce a modified dataset from an existing one, or draw relations between datasets. We define the former as *transformations* and the latter as *combinations*.

Transformations Transformations serve the purpose of either inferring some new information from provided information or changing the representation of information in a dataset. Deriving new information involves calculating a new data element composed of one or more existing elements; e.g. dividing instruction counts by elapsed times to obtain instruction rates. Transformation functions are only valid for a provided dataset if the dataset contains the required set of semantics. In the previous example, in order to derive instruction rates, we require a dataset containing instruction counts and elapsed times. Transformations that change data representation modify the semantics of some dataset, e.g. by converting units or denormalizing an element into its constituent parts. In all cases, however, we cannot validly modify the dimensions of the *domain* elements of a dataset—a measurement defined over time may never *not* be defined over time. We may instead discover additional domain dimensions over which a dataset is defined, by inferring relations to other datasets using combinations.

Combinations Two data elements form a *relation* if both elements describe the same domain entity, such as a CPU, an application, a point in time, or some combination of these. Combination functions take two datasets and infer a relation between their elements, yielding a dataset with rows containing elements from both input datasets. They are essentially generalized JOIN operations that make use of data semantics, rather than user input, to determine correspondences between two datasets and combine their elements.

We may draw a relation between values of one dataset and values of another if both share a common domain; therefore, a combination is possible if (and only if) the two input datasets share a domain dimension. Furthermore, all common domain dimensions between two datasets must match to yield a relation—two CPU measurements taken at the same time but on different CPUs do not form a valid relation.

Under these rules, combinations must compare all common domain elements between pairs of rows from both datasets to determine a correspondence between them. This comparison function is defined by the semantics of the domain elements: unordered elements like CPU ID must match exactly to form a relation, while ordered elements like time may be compared with a distance metric. In the latter case, we may interpolate or extrapolate the elements from one row to match the domain elements of another. The resulting correspondence between rows of one dataset and rows of another may overlap, resulting in one-to-one, one-to-many, or many-to-one correspondences.In the last two cases, we must aggregate multiple elements with the same semantics. We define the aggregation operation again using these semantics. For example, if we have two temperature readings that occurred within one second of each other, and both map to the same second in time, we average their results. In another case, if two application instructions map to the same second in time, we may pick the instruction closest to that second. By constraining the operations available to the semantics of the data involved, we ensure that the combination yields valid relations between its provided datasets.

Semantic annotation represents the largest hurdle in user adoption of ScrubJay. For this reason, we have designed ScrubJay to require a minimal amount of user-provided data semantics. We provide defaults and make many previously defined semantics available. In the worst-case scenario, a user would need to specify custom semantics for all elements in a dataset. We argue that doing so is still worthwhile, as ScrubJay would then provide all the necessary infrastructure to perform distributed data processing operations consistently and scalably. It is also potentially less tedious than defining custom data processing scripts, especially if the semantics can be later reused.

5 QUERY SATISFACTION

The fact that we can gain new information by performing derivations, along with the fact that we can run derivations on already-derived data, leads to a combinatorial growth of available information. In some cases, we may even infinitely perform derivations. However, the information gained may contain several redundancies, and much of it may never be requested for analysis. We therefore designed ScrubJay to determine sequences of derivations on-demand, based on the kinds of performance data requested by a user. We developed an algorithm that efficiently navigates this immense search space to find solutions to queries at interactive rates.

5.1 Queries

Performance analysts request data by submitting a ScrubJay query. Unlike traditional query languages consisting of table names, columns and additional clauses, ScrubJay queries consist only of a set of data sources and measurements of interest[1]. In our semantic language, they specify the dimensions of the domains and the dimensions (and optionally units) of the values of interest. ScrubJay then determines whether a sequence of derivations exists that produces a dataset containing a relation between the domain dimensions and value dimensions specified. For example, a user may request CPU active frequencies and rack temperatures in order to determine whether frequency scaling contributes to heat generation in the facility. Here, the domain dimensions are CPUs and racks, and the value dimensions are active frequencies and temperatures. ScrubJay finds sequences of derivations that, when executed, produce a ScrubJayRDD where each tuple contains information about CPUs, their frequencies, their associated racks, and relevant temperature readings. These tuples often contain additional domain information associated with the queried dimensions. A row of the solution to this example query will likely also contain the time associated with the frequency and temperature recordings, the node in which the CPU resides, and the rack containing that node.

5.2 Derivation Engine

We formalize the problem of finding these derivation sequences as a variant of the Constraint Satisfaction Problem (CSP) and employ constraint programming techniques to solve it in what we deem the *derivation engine*[2]. A CSP involves finding an assignment of variables that satisfy a set of constraints. Here, we are finding a sequence of derivations on datasets that result in a dataset that satisfies our query. For example, to satisfy a query for power consumption and jobs, we may transform job queue datasets into a representation describing all active jobs during the times that power measurements were collected and combine that result with the power measurement dataset. Thus, our variables consist of both derivations and datasets, and we vary from traditional CSPs because the length of these sequences (and therefore the number of variables) is unspecified.

We can apply traditional CSP solvers to this problem over finite sequence lengths but doing so naïvely is not an option. A single derivation operation may take minutes or longer to compute for large datasets, and many such derivations would have to be repeatedly computed for different sequences. We mitigate this by (1) performing derivations on the data semantics only, rather than on the dataset itself, (2) pruning the search space and prioritizing short derivation sequences, and (3) using memoization to cache previously computed derivation results.

A derivation function defines the semantics required to perform the derivation as well as the semantics of the resulting dataset. Thus, instead of running the actual derivation, we first check whether a dataset is valid, and if so, we compute the derived semantics only. The semantics are internally represented in a hash map, so derivations need only make a constant-time lookup to check validity. Computing the derived semantics may vary with respect to an individual derivation, but we can reasonably expect it to require constant or near-constant time; in all tested cases, the derivation simply provides an additional semantic entry, or combines the semantics of two datasets. Thus, we are able to cheaply calculate the resulting semantics of each derivation step.

Algorithm 1 Derivation Search

```
function CombineSet($D : List(d_1..d_N)$)
    if $N = 2$ then
        return CombinePair($d_1, d_2$)
    end if
    $rest \leftarrow$ CombineSet($List(d_2..d_N)$)
    return CombinePair($d_1, rest$)
end function
function Query($D : List(d_1..d_N), Q : List((s_1, v_1)..(s_M, v_M))$)
    $D^F \leftarrow d \in D \mid s \in d \wedge (s, v) \in Q$
    while $D^F \neq D$ do
        $r \leftarrow$ CombineSet($D^F$)
        if $r$ is a solution then
            return $r$
        end if
        $D^F \leftarrow D^F + d$ for some $d \in D - D^F$
    end while
    return *no solution*
end function
```

We prune and organize the solution search by performing a backward-chaining search with additional logic. We first find the smallest set of existing datasets containing any of the domain dimensions listed in the query. If any single dataset contains all of them, it must satisfy the query, and we thus return it as a solution. If one or more queried domain dimensions are not found, no solution exists, because derivations cannot infer new domain dimensions. If the domain dimensions are split across multiple datasets, we search for sequences of derivations that combine them. In this search, we prioritize shorter derivation sequences, first testing whether we

[1] We recognize the need for filtering and aggregation semantics provided by traditional relational database tools. Rather than reinvent such semantics in our query system, we provide an interoperability layer to enable this functionality.
[2] The name "derivation engine" is a reference to the "inference engine" proposed in early artificial intelligence research. Rather than infer facts using rules, ScrubJay derives datasets using their semantics and derivation functions.

can combine the smallest set of datasets containing the requested domain dimensions, then adding the remaining datasets one at a time. The pseudocode in Algorithm 1 illustrates this process, with a list of existing datasets D and a query Q containing a domain dimension s_i and a value dimension v_i for each entry. The omitted function *CombinePair* performs a straightforward test to determine whether two datasets may be combined through a sequence of transformations and a single combination.

This derivation search algorithm has two main advantages. First, we generally prefer to obtain the highest precision data available, and interpolation and aggregation involved in each derivation step may decrease precision. This search pattern ensures we will return the shortest available derivation sequence solution. Second, we can take advantage of the search pattern to avoid redundant computation. We use a memoization strategy to cache the results of *CombinePair* and *CombineSet* at runtime. At each iteration, *CombineSet* receives a superset of the arguments that it previously received, so most of its recursive calls will have already been computed. These optimizations enable ScrubJay's derivation engine to return solutions to queries at interactive rates.

5.3 Data-Parallel Derivations

We also require scalable methods to perform the actual derivation sequences themselves. We accomplish this by implementing derivations as data-parallel Spark algorithms. The Spark framework provides implicit data-parallelism for functional-programming operations performed on resilient distributed datasets. Most derivations by nature lend themselves readily to functional programming techniques, such as map, aggregate, and reduce operations, so we reasonably enforce ScrubJay derivation functions to operate within them. In doing so, we are able to ensure the ability of derivations to scale to datasets that are potentially large and distributed.

However, there is one particular case where functional operations fall short of our needs. Frequently, we require determining correspondences between elements that do not match exactly. For example, two different performance measurements may occur within a small window of time, in which case we wish to draw a near-relation between them. Naïvely, this scenario would require computing all pairwise distances between two datasets, which is unscalable. In a non-distributed scenario, we could sort all elements and iterate over each dataset sequentially; however, because our data is potentially massive and distributed randomly over any number of nodes, sorting and sequential processing are not viable options. We therefore introduce a novel data-parallel algorithm for relating two datasets based on an ordered distance metric.

We constrain the problem slightly—we calculate correspondences between all elements within a specified window W of each other. This constraint allows us to frame the problem using functional programming semantics. We divide each dataset into bins of size $2W$, then again into the same size bins, but offset by exactly W. This binning scheme guarantees that every element within W of some element resides in one of the same bins. We then filter out elements further than W and aggregate all bins with respect to each element, yielding the desired result.

Because the resulting correspondences are not exact, we interpolate to determine a matching value for each combined element. We thus deem our algorithm the *interpolation join*. This addition enables ScrubJay to accurately combine data over continuous, ordered domains in a scalable way.

5.4 Using Derivation Results

The derivation engine finds sequences of derivations that will yield a result satisfying a query, but does not actually perform the derivation itself. This feature is by design—by decoupling the specification of a data processing pipeline from the actual data processing, we are able to introduce an advanced framework for reproducing, refining, distributing, and collecting derivation results.

Reproducible Derivation Sequences

In the current state of affairs, if an analyst wishes to reproduce analysis results, their best hope is to acquire the same data processing scripts used in the initial analysis to ensure consistency. Such scripts are nearly never included in publication or even in public repositories, and there is no guarantee that they will provide equivalent (or any) functionality for different datasets. In addition, the recent rise of performance variability and heterogeneous data is increasingly impeding efforts to reproduce analysis results, and the scientific community is responding with a surge in demand for reproducibility efforts.

In ScrubJay, we present a solution to this growing problem. We provide a reproducible and compact representation in which users can store specific derivation sequences for distribution and reuse. This representation includes all information necessary to execute an identical processing pipeline and additionally is human-readable and may be edited directly. This latter feature enables advanced users to tweak derivation sequences for custom purposes, the result of which may be readily shared for less advanced users to reuse.

ScrubJay's derivation sequence representation is simple: we serialize the sequence of derivation operations, including data loading and wrapping operations, into JSON . We also do not require additional input from developers of derivation functions, we gather all required information through code reflection.

To our knowledge, ScrubJay introduces the first general reproducible data processing framework. We believe future efforts to enable reproducible analysis may greatly benefit from our system.

Computing and Collecting Derivation Results

The time required to perform a derivation sequence depends on the requested sizes of the datasets, the kinds of derivations and data operations involved, and the overhead of wrapping the data from heterogeneous sources. In addition, some derivation sequences may yield results too large to fit into the memory of a single system, or users may wish to use those results in different analysis tools. Data scientists may prefer to use our distributed in-memory format directly, using the many machine learning libraries already available for RDD datasets. As a result, computing and storing results from derivation sequences poses additional challenges.

We mitigate the problem of collecting results by introducing an analog to data wrappers, data *unwrappers*. These may convert a ScrubJayRDD into a common format, such as a CSV, a custom format, or dump results into distributed/parallel storage systems. In case a user wishes to use the in-memory format directly, we also provide a handle to access it as a Spark data structure.

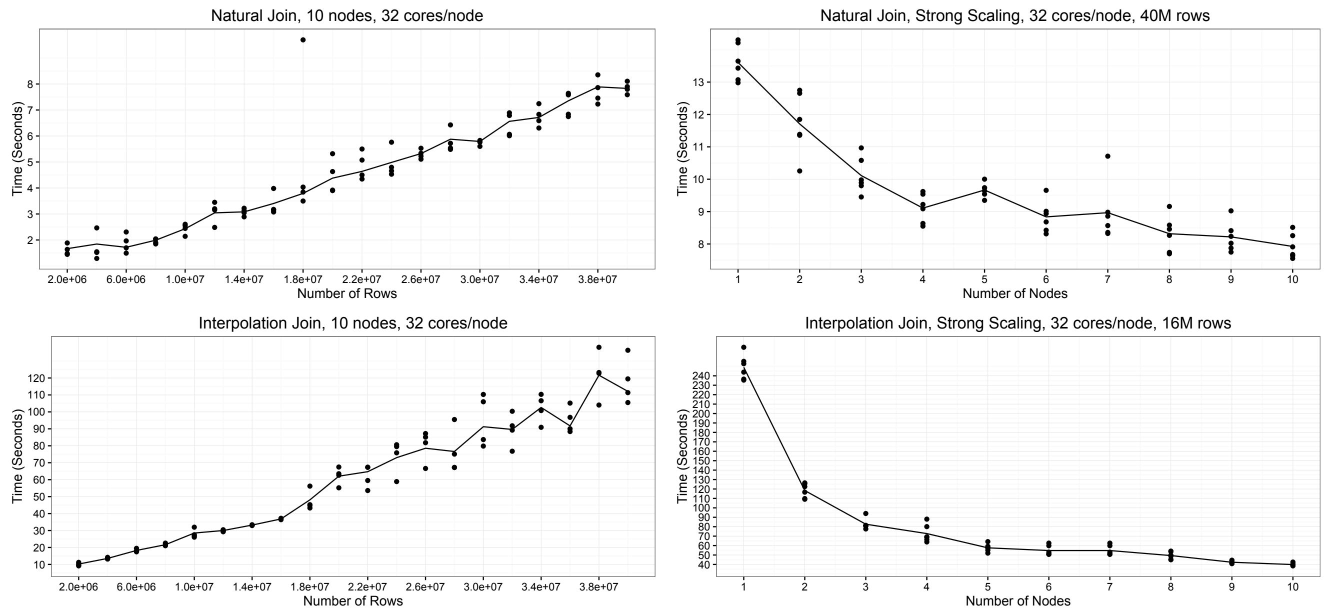


Figure 3: Performance scaling of the two most expensive derivation functions in ScrubJay, Natural Join and Interpolation Join. (Left) 2M to 40M rows on all 10 nodes. (Right) Fixed problem sizes, 1 to 10 nodes (strong scaling).

With this storage capability, we also develop an optimization strategy for computing expensive derivation sequences. Using a similar strategy to our memoization of derivation tests in the derivation engine, we cache intermediate results of derivation sequences and provide unique identifiers to reuse them. As a result, two derivation sequences that perform the same expensive derivation operation need to do so only once. This cache is maintained in non-volatile storage, and thus over time, derivation sequences prune away more and more redundant computations. This strategy may also incur large storage overheads, so we allow it as an opt-in option, and evict stale entries in a least recently used (LRU) strategy.

6 PERFORMANCE

ScrubJay's data-parallel design allows it to scale to large quantities of data distributed over many nodes. Determining derivation sequences is relatively inexpensive since we operate only on the data semantics and use a polynomial time algorithm to do so. Different derivation sequences have widely different performance characteristics, however.

Spark operations are typically bounded by *shuffle* operations, where data must be shuffled between all compute nodes to compare their values. Transformations typically do not incur shuffle operations and are thus relatively inexpensive to perform. Combinations, however, necessarily impose a shuffle when comparing values between datasets. We therefore evaluate the performance of ScrubJay by presenting scaling studies of two different combination methods, Natural Join and Interpolation Join, the most expensive derivations available.

The results are shown in Figure 3. All experiments were performed on a cluster with 10 available compute nodes, 32 cores and 64gb memory per node (Intel Xeon CPU E52667 v3 @ 3.20GHz). We see that for a fixed number of nodes, processing time increases linearly with respect to the number of input rows. The strong scaling study shows that despite the shuffle bottleneck, ScrubJay is able to achieve performance gains by scaling up the data cluster. These results give us confidence that ScrubJay will scale to the growing quantities of performance data when distributed over a wide data-parallel cluster.

7 CASE STUDIES

We used ScrubJay to analyze correlations between a variety of disparate HPC data sources utilized during two seperate dedicated-access time (DAT) sessions on the production cluster Cab. Both DATs involved multiple heterogeneous workloads, with different applications, problem sizes, and node allocations.

7.1 Data Collection and Semantics

In the first DAT, we collected data from a variety of facility-level monitoring infrastructures, notably:

- Job Queue Logs (SLURM resource allocator)
- Rack Temperature, Humidity, Power (OSIsoft PI)
- Node/Rack Layout (provided by system administrators)

The first two data sources are continuously monitored and recorded in relational databases. As such, we created a common data wrapper to extract column names from their schemas and convert their rows to named tuples. We defined semantics for the job queue columns based on the resource scheduler documentation, and worked directly with facility administrators to define semantics for the power and temperature data collected by a proprietary interface. The node/rack layout data is static information defining which nodes reside on which racks. We obtained this information directly from another facility administrator and encoded it in a tabular format.

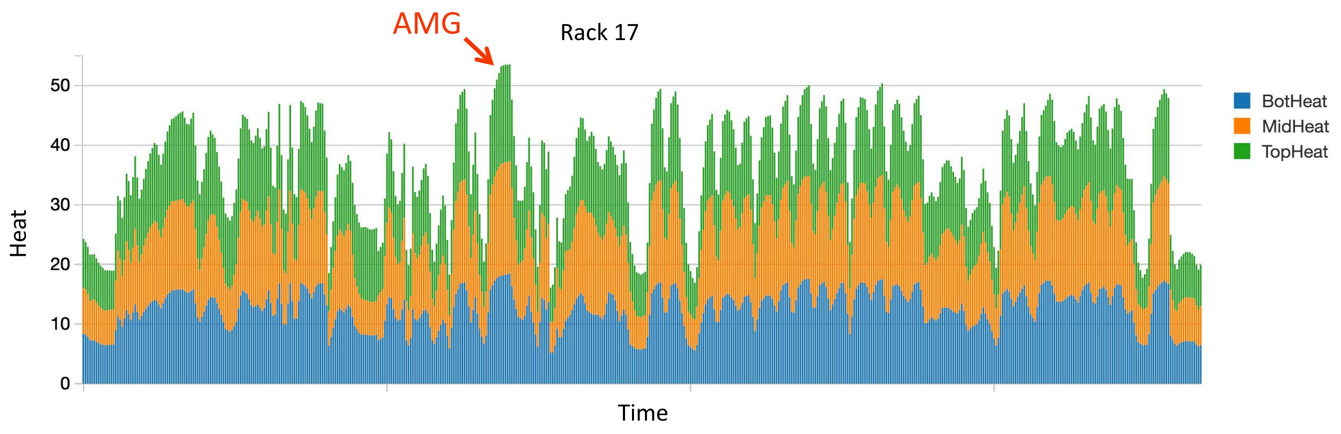


Figure 4: After deriving a dataset incorporating facility heat production on different racks with application names for different jobs, we were able to discern which applications were correlated with the greatest heat generation period. Here we plot the heat profiles during this period, for the bottom (blue), middle (orange), and top (green) of rack 17, over time.

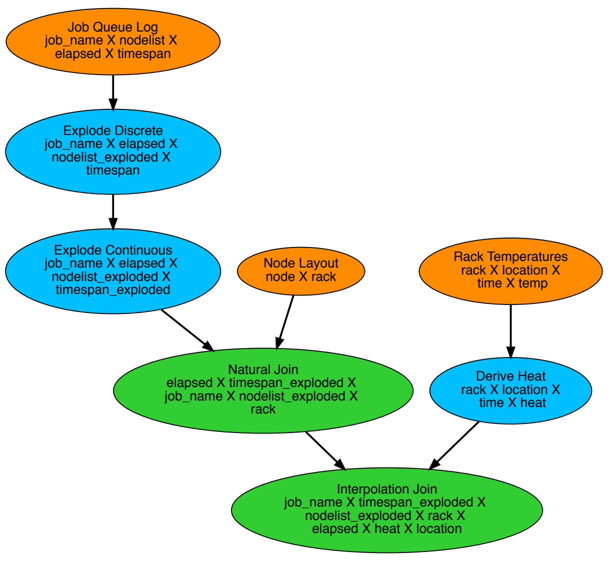


Figure 5: The resulting derivation sequence from querying job application names and rack heat from 3 datasets first DAT, including original (orange), transformed (blue), and combined (green) datasets.

In the second DAT, we additionally collected high-fidelity performance data from the node, motherboard, and CPU cores:

- Intelligent Platform Management Interface (IPMI) motherboard status information
- Lightweight Distributed Metric Service (LDMS) node counters and status information
- PAPI CPU counters
- CPU specifications

IPMI and PAPI recorded performance data directly into tabular files, and we employed a distributed ingestion framework to continuously collect LDMS data into a distributed NoSQL database store. In order to derive cpu-specific performance information, we also collected CPU specifications directly from the available linux device file `/proc/cpuinfo`. We defined data semantics again through close collaboration with system administrators and using information available in documentation.

We incorporated all data wrappers into ScrubJay and stored data semantics in our distributed database with shared access. As a result, we were able to reuse the semantics of the data from the first DAT seamlessly in the second, and this information continues to be readily available in ScrubJay's knowledge base.

In addition to these semantics, ScubJay provides a general set of transformation functions to modify data representations. These transformations essentially perform a transpose on a single element of a dataset, by denormalizing a row containing a list of elements into multiple rows with a single element. This general transformation serves the purpose of creating a dataset with comparable elements to another dataset and thus enables combinations between them. We provide a continuous analog that transforms a row containing a span of data into several rows containing discrete instances within that span, for example converting a time range into a set of time stamps within that range. We henceforth call these transformations for discrete lists and continuous spans *explode discrete* and *explode continuous*.

7.2 Application Impact on Rack Heat Generation

While rack temperature information is continuously collected on Cab, little is known about specific applications' contributions to the generation of heat in the facility. Temperature information is also available for subsystems within the node, such as the CPU or memory controller, but quantifying the contribution of heat at the level of the facility requires measuring the cumulative effects from all subsystems interacting together in a physical space. Heat

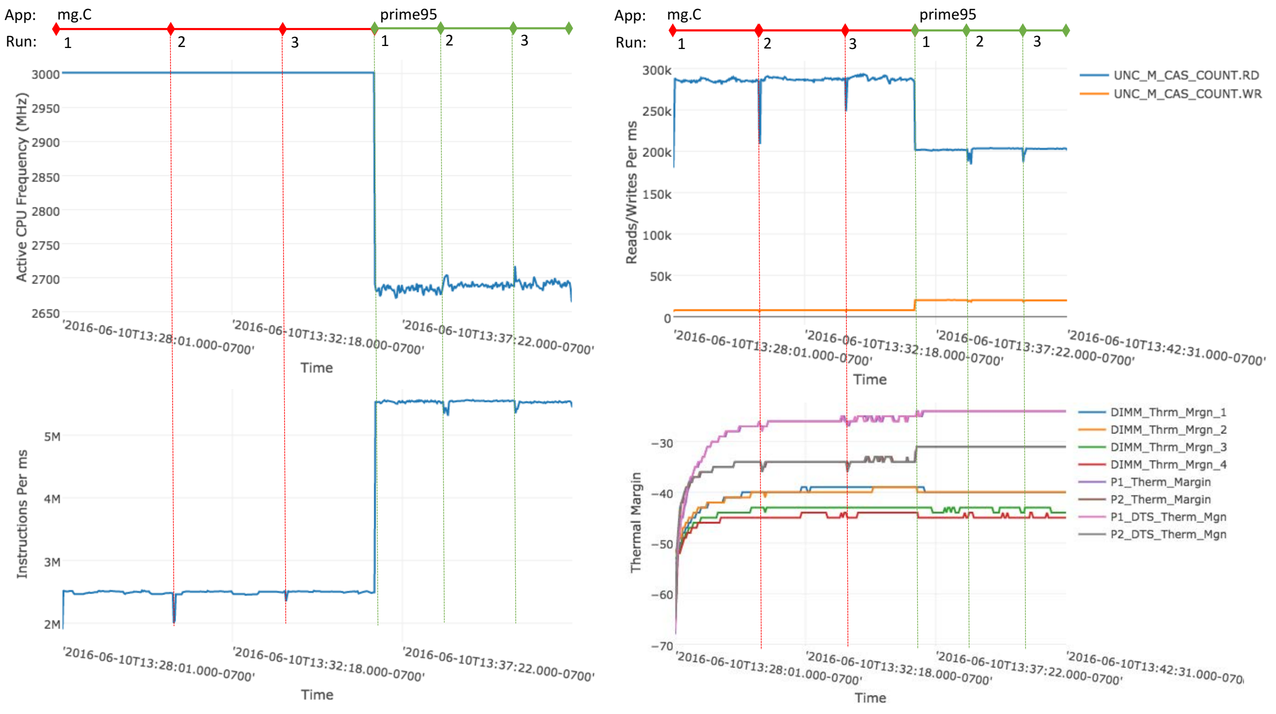


Figure 6: Plots of derived values describing CPU (left) and node (right) performance during the execution of the memory-intensive workload mg.C followed by the compute-intensive workload prime95.

generation in the facility in turn has consequential effects on the power throttling of cooling units, heat dissipated to neighboring subsystems, and generation of heat-related errors or malfunctions.

Using the collected facility data, we defined a derivation to quantify heat generation on racks. The facility monitoring infrastructure provides 6 temperature sensors on each rack of Cab, localized to the top, middle, and bottom of both the hot and cold aisles. Instantaneous temperature measurements are recorded from all sensors every two minutes continuously. Our heat derivation function takes hot and cold aisle temperature readings at one instant in time and calculates their difference as an approximation of instantaneous heat generated. Thus, we obtain a measure for heat at the top, middle, and bottom of the rack.

With this information in place, we simply requested the value "application names" associated with the domain "jobs" and the value "heat" associated with the domain "racks". ScrubJay returned the derivation sequence shown in figure 5.

The derivation sequence first transforms the job queue log data with *explode discrete* and *explode continuous*, yielding a dataset describing the application executing on every node at every instant in time. This result is then combined with the node layout dataset, matching elements with the same node identifiers and thus describing the rack that the application was executing on. Independently, our defined heat derivation function extracts heat information from the rack temperature dataset. Finally, the previous two derived datasets are joined interpolatively, matching application execution instances with heat measurements that occurred closely in time and interpolating values that are not exactly matched.

We collected the results and unwrapped them into a tabular file for analysis. We sorted the results with respect to heat and quickly identified an outlier. The most heat was being generated on rack 17 while executing the adaptive mesh refinement application AMG on 60 of its respective nodes. We plotted the collective heat generated on the top, middle, and bottom locations of rack 17 over time to visualize the heat profile during this time in figure 4. Here we are able to characterize the thermal signature of not only AMG but all applications. AMG demonstrates a fairly regularly increasing heat curve, while other applications rise and fall over time, presumably as they enter different application phases.

7.3 CPU Frequency Throttling Impact on Node Power Consumption

For the second DAT, we increased the quantity and complexity of data sources by an order of magnitude. Whereas temperature readings are collected on a two-minute interval, we additionally collected CPU, motherboard, and node performance data on one- to three-*second* invervals. The discrepancy between measurement granularities and the sheer volume of data generated over multiple hours make manual analysis of this information particularly difficult. Furthermore, in the previous case we utilized static information about the node and rack layouts, and here we take it a step further to include static CPU specifications to enable derivation of CPU-specific performance characteristics.

We ran two types of workloads during this DAT: a compute-intensive application prime95 and a memory-intensive arithmetic

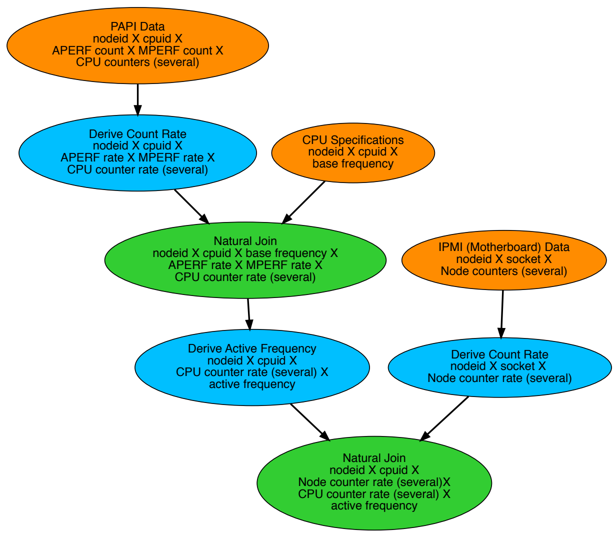


Figure 7: The resulting derivation sequence from querying CPU active frequency and several counter rates for CPU and node events, from data collected during the second DAT.

application mg.C. We also enabled on-demand CPU throttling throughout the duration of the workloads. Because of their contrasting instruction mixes, we hypothesized that they would incur vastly different CPU throttling characteristics, and we sought to determine what impacts this would have on the temperatures and power draw of different subsystems on the node.

In contrast to the previous facility-level measurements, much of the available CPU and node data sources record cumulative counts of particular events, e.g. instructions, memory accesses, rather than instantaneous values. Furthermore, these counter values reset at some arbitrary interval, making their absolute values irrelevant by themselves. We therefore developed a derivation to calculate the rate of change of counter values with respect to a window of time, effectively measuring the instantaneous frequency of events.

Accurate and precise measurements of active CPU frequency are not directly available from CPU data collection mechanisms however. Instead, each CPU provides a counter that increments at the base processor frequency (MPERF) and another that does so at the active frequency (APERF). Therefore, we must derive the active frequency at a point in time by calculating the rate of change of each counter and multiplying their ratio by the base frequency specified for that particular CPU. Our aforementioned derivation effectively calculates the rate of change for counter values, thus we developed one additional derivation to calculate active frequency for a dataset containing rates of change for APERF and MPERF and the base frequency of their associated CPU. The latter data element, while not immediately available from the counters recorded in PAPI, is available in the static CPU specification dataset. Thus, we rely on ScrubJay to infer a relation to this dataset in order to derive active frequency.

With these new derivations in place, we queried the value "active CPU frequency" for the domain "CPU" along with counter values from both the CPU and the motherboard, including memory read and write rates, power draw from different processor sockets, and temperature and thermal margins for CPUs and memory banks. The resulting derivation graph is shown in Figure 7.

We plotted several of the resulting derived data elements in figure 6 for a specified set of application executions. During this time, 6 workloads were executed, 3 instances of mg.C and 3 of prime95, in that order. The left plots show derived CPU-level performance data; active CPU frequency and instructions per millisecond, while the right shows derived node-level data; memory reads/writes per millisecond, and thermal margins of different processor sockets and memory controllers. We observe that mg.C operated at full CPU frequency and lower instruction rate, while prime95 incurred high instruction rates and experienced aggressive CPU throttling.

8 RELATED WORK

Performance databases ScrubJay is not the first tool to perform analysis on performance data in a database. Huck et al.'s PerfExplorer and PerfDMF [12, 13, 24], store data from TAU [26] and other parallel performance systems in a relational database, and they allow the user to perform queries on a predetermined, typed schema. Mucci's PerfMiner [25], Gallo et al.'s machine learning experiments [7], and PerfTrack [15], and PerfExpert [5] are similar in that they use relational databases with strict schemata. ScrubJay stores unstructured but *annotated* performance data. It can ingest arbitrary data without the need to first transform it to a particular schema. ScrubJay can also query over both raw data fields *and* completely derived, on-demand transformations of raw fields. To our knowledge, this capability of ScrubJay is completely unique among performance mining systems. General log analysis tools like Splunk[6] and Graylog are the closest analog to ScrubJay's aggregations, joins, and projections, but these tools do not automate the generation of transformations like ScrubJay.

Parallel performance analysis Other tools have performed parallel analysis of performance data. Gamblin's AMPL [10], Libra [8], and CAPEK [9] tools performed parallel, in-situ sampling, wavelet, and cluster analysis on performance data, respectively. AutomaDeD and Prodometer [4, 17–21, 23] perform scalable, in-situ outlier detection to discover bugs at runtime. The Vampir Next-Generation (VNG) [16] tool uses parallel processing to analyze and visualize very large MPI trace volumes. None of these tools, however, implement parallel join semantics to automatically transform annotated, schemaless performance data. ScrubJay's on-demand transformation capability is unque among performance tools. Prior work on MemAxes [11] and Boxfish [22] allow for projecting performacne data across domains, but they do not provide a fully general system for query derivation; users of these tools must manually explore the projection space.

Monitoring infrastructure Finally, ScrubJay is not a measurement framework; it is a scalable, offline analysis tool. It relies on tools such as LDMS [2], HPCToolkit [1], TAU [26], and Caliper [3] to provide performance data to ingest, and it relies on scalable database and analysis systems (such as Cassandra and Spark) for data measurement and analysis infrastructure.

9 CONCLUSION

Our system effectively enables us to characterize derived performance information and relationships between different domains in the HPC ecosystem. ScrubJay now contains the relevant knowledge base to perform these derivations automatically, allowing us to perform this analysis over historical and incoming data in a consistent and reproducible way. This work, used in collaboration with HPC facility engineers, currently fuels next-generation efforts in designing resource-aware job schedulers and determining optimal hardware configurations for specific workloads.

We are extending our uses of ScrubJay to analyze applications and hardware at finer granularities, by quantifying relationships between individual kernels, loops, and lines of code with specific CPU cores, threads, and caches. We also see enormous potential in using ScrubJay to relate application behaviors to utilization across the network, an area of increased nondeterministic behavior due to interference.

While we have developed ScrubJay and built its backend data cluster with data scalability in mind, we still see potential for improvement. We can cache intermediate derivation results in ScrubJay, but too large a cache could deplete our available storage. We envision having a storage cache hierarchy in the future, where old entries may be compressed and stored in separate long-term storage devices.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-735962).

REFERENCES

[1] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. 2010. HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.

[2] Anthony Agelastos, Benjamin Allan, Jim Brandt, Paul Cassella, Jeremy Enos, Joshi Fullop, Ann Gentile, Steve Monk, Nichamon Naksinehaboon, Jeff Ogden, Mahesh Rajan, Michael Showerman, Joel Stevenson, Narate Taerat, and Tom Tucker. 2014. The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications. In *Supercomputing*.

[3] David Böhme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Giménez, Matthew LeGendre, Olga Pearce, and Martin Schulz. 2016. Caliper: Performance Introspection for HPC Software Stacks. In *Supercomputing 2016 (SC'16)*. Salt Lake City, UT. LLNL-CONF-699263.

[4] Greg Bronevetsky, Ignacio Laguna, Saurabh Bagchi, Bronis R. de Supinski, Dong H. Ahn, and Martin Schulz. 2010. AutomaDeD: Automata-Based Debugging for Dissimilar Parallel Tasks. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010)*. Chicago, IL.

[5] Martin Burtscher, Byoung-Do Kim, Jeff Diamond, John McCalpin, Lars Koesterke, and James Browne. 2010. PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, Washington, DC, USA, 1–11. https://doi.org/10.1109/SC.2010.41

[6] David Carasso. 2012. Exploring splunk. *published by CITO Research, New York, USA, ISBN* (2012), 978–0.

[7] S. M. Gallo, J. P. White, R. L. DeLeon, T. R. Furlani, H. Ngo, A. K. Patra, M. D. Jones, J. T. Palmer, N. Simakov, J. M. Sperhac, M. Innus, T. Yearke, and R. Rathsam. 2015. Analysis of XDMoD/SUPReMM Data Using Machine Learning Techniques. In *2015 IEEE International Conference on Cluster Computing*. 642–649. https://doi.org/10.1109/CLUSTER.2015.114

[8] Todd Gamblin, Bronis R. de Supinski, Martin Schulz, Robert J. Fowler, and Daniel A. Reed. 2008. Scalable Load-Balance Measurement for SPMD Codes. In *Supercomputing 2008 (SC'08)*. Austin, Texas, 46–57.

[9] Todd Gamblin, Bronis R. de Supinski, Martin Schulz, Robert J. Fowler, and Daniel A. Reed. 2010. Clustering Performance Data Efficiently at Massive Scale. In *International Conference on Supercomputing*. Tsukuba, Japan.

[10] Todd Gamblin, Robert J. Fowler, and Daniel A. Reed. 2008. Scalable Methods for Monitoring and Detecting Behavioral Classes in Scientific Codes. In *Proceedings of the 22nd International Parallel and Distributed Processing Symposium (IPDPS 2008)*. Miami, FL, 1–12.

[11] Alfredo Giménez, Todd Gamblin, Barry Rountree, Abhinav Bhatele, Ilir Jusufi, Peer-Timo Bremer, and Bernd Hamann. 2014. Dissecting On-Node Memory Access Performance: A Semantic Approach. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Computer Society. LLNL-CONF-658626.

[12] Kevin A. Huck, Oscar Hernandez, Van Bui, Sunita Chandrasekaran, Barbara Chapman, Allen D. Malony, Lois Curfman McInnes, and Boyana Norris. 2008. Capturing Performance Knowledge for Automated Analysis. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC '08)*. IEEE Press, Piscataway, NJ, USA, Article 49, 10 pages. http://dl.acm.org/citation.cfm?id=1413370.1413420

[13] Kevin A. Huck and Allen D. Malony. 2005. PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing (SC '05)*. IEEE Computer Society.

[14] Katherine E. Isaacs, Peer-Timo Bremer, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, and Bernd Hamann. 2014. Combing the Communication Hairball: Visualizing Parallel Execution Traces using Logical Time. *IEEE Transactions on Visualization and Computer Graphics* (Dec. 2014). LLNL-JRNL-657418.

[15] K. L. Karavanic, J. May, K. Mohror, B. Miller, K. Huck, R. Knapp, and B. Pugh. 2005. Integrating Database Technology with Comparison-based Parallel Performance Diagnosis: The PerfTrack Performance Experiment Management Tool. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*. 39–39. https://doi.org/10.1109/SC.2005.36

[16] Andreas Knüpfer, Holger Brunst, and Wolfgang E. Nagel. 2005. High Performance Event Trace Visualization. In *Euro Workshop on Parallel, Distributed and Network-Based Processing (PDP 2005)*.

[17] Ignacio Laguna, Dong H. Ahn, Bronis R. de Supinski, Saurabh Bagchi, and Todd Gamblin. 2012. Probabilistic Diagnosis of Performance Faults in Large Scale Parallel Applications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. Minneapolis, MN. http://www.cs.unc.edu/~tgamblin/pubs/2012/laguna-automaded-diagnosis-pact12.pdf LLNL-PROC-548642.

[18] Ignacio Laguna, Dong H. Ahn, Bronis R. de Supinski, Saurabh Bagchi, and Todd Gamblin. 2014. Diagnosis of Performance Faults in Large Scale MPI Applications via Probabilistic Progress-Dependence Inference. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (May 22 2014). LLNL-JRNL-643939.

[19] Ignacio Laguna, Dong H. Ahn, Bronis R. de Supinski, Todd Gamblin, Gregory L. Lee, Martin Schulz, Saurabh Bagchi, Milind Kulkarni, Bowen Zhou, Zhezhe Chen, and Feng Qin. 2015. Debugging High-Performance Computing Applications at Massive Scales. *Commun. ACM* (September 2015). LLNL-JRNL-652400.

[20] Ignacio Laguna, Todd Gamblin, Bronis R. de Supinski, Saurabh Bagchi, Greg Bronevetsky, Dong H. Ahn, Martin Schulz, and Barry Rountree. 2011. Large Scale Debugging of Parallel Tasks with AutomaDeD. In *Supercomputing 2011 (SC'11)*. Seattle, WA. https://engineering.purdue.edu/dcsl/publications/papers/2011/debugging_ded_supercom11.pdf LLNL-CONF-486911.

[21] Ignacio Laguna, David F. Richards, Todd Gamblin, Martin Schulz, Bronis R. de Supinski, Kathryn Mohror, and Howard Pritchard. 2016. Evaluating and Extending User-Level Fault Tolerance in MPI. *International Journal of High Performance Computing Applications (IJHPCA)* (January 11 2016). LLNL-JRNL-663434.

[22] Aaditya G. Landge, Joshua A. Levine, Katherine E. Isaacs, Abhinav Bhatele, Todd Gamblin, Martin Schulz, Steve H. Langer, Peer-Timo Bremer, and Valerio Pascucci. 2012. Visualizing Network Traffic to Understand the Performance of Massively Parallel Simulations. In *IEEE Symposium on Information Visualization (INFOVIS'12)*. Seattle, WA. LLNL-CONF-543359.

[23] Subrata Mitra, Ignacio Laguna, Dong H. Ahn, Saurabh Bagchi, Martin Schulz, and Todd Gamblin. 2014. Accurate Application Progress Analysis for Large-Scale Parallel Debugging. In *Programming Langauge Design and Implementation (PLDI'14)*. Edinburgh, UK. LLNL-CONF-646258.

[24] Shirley Moore, David Cronk, Felix Wolf, Avi Purkayastha, Patricia Teller, Robert Araiza, Maria Gabriela Aguilera, and Jamie Nava. 2005. Performance profiling and analysis of dod applications using papi and tau. In *Users Group Conference, 2005*. IEEE, 394–399.

[25] Philip Mucci, Daniel Ahlin, Johan Danielsson, Per Ekman, and Lars Malinowski. 2005. PerfMiner: Cluster-wide collection, storage and presentation of application level hardware performance data. *Euro-Par 2005 Parallel Processing* (2005), 612–612.

[26] S. Shende and A. D. Malony. 2006. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications* 20, 2 (2006), 287–311. https://doi.org/10.1177/1094342006064482

[27] Jeffrey Vetter and Chris Chambreau. 2005. mpiP: Lightweight, Scalable MPI Profiling. (2005). http://www.llnl.gov/CASC/mpip