Eiger Performance Modeling Toolkit

Eric Anger - eanger@gatech.edu
Andrew Kerr - akerr@gatech.edu
Benjamin Allan - baallan@sandia.gov
Sudhakar Yalamanchili - sudha@ece.gatech.edu

Contents

1 Overview

1.1 Introduction

Eiger is a framework for the construction, analysis, and use of statistical performance models of computer systems. It is designed with the goals of flexibility and understandability above that of model construction speed. As such, the framework uses as little expert knowledge about the modeled system as possible, as well as generalized notions of independent and dependent metrics. The models it generates are concise, explicit functional representations of the performance metric, allowing for insight into those metrics that most contribute to performance. As well, a set of additional analysis tools are provided to aid users in validating, improving, and understanding the models they generate. This document will present a brief lay out of key components within Eiger, common utilization methodology, and resources for more information.

1.2 Modeling Procedure

As a statistical modeling tool, Eiger requires training data. This data is what informs the performance models, as well as guides their construction and accuracy. Eiger will generate performance models that only follow the training data it is provided with; this implies that Eiger must be trained on data representative of the types of models the user wishes to generate. As a rule of thumb, the more data that Eiger has to work with, the better quality the models. The pathway that Eiger sets up for modeling has several phases:

Collection of Instrumented Data Instrumentation API implementations in Python and C++ for sending collected input data to the Eiger internal data store.

Storage MySQL database schema and integration with instrumentation APIs.

Generation Flexible, semi-automated statistical model generation.

Analysis Performance and structure visualization tools, as well as validation procedures.

Export Standardized model object export, integration with SST/macro large-scale simulator.

1.3 Further Information

For more information on the motivation, implementation details, experimental results, and exposition on future work, see the publication "Eiger: A Framework for the Automated Synthesis of Statistical Performance Models" in the 1st Workshop on Performance Engineering and Applications, held in conjunction with the 2012 High Performance Computing (HiPC) conference. It is also included in this directory as `eiger-WPEA2012.pdf`.

Should any questions or concerns arise, feel free to contact the authors:

Eric Anger - eanger@gatech.edu
Andrew Kerr - akerr@gatech.edu

2 Dependencies

This document specifies software dependencies for running Eiger. The indicated versions are the ones tested with; your luck may vary with other (including newer) versions.

2.1 Framework

- Python 2.7.3
- Matplotlib 1.1.1
- Numpy 1.6.1
- SciPy 0.10.1
- Scikit-learn 0.11
- MySQLdb 1.2.3

2.2 C++ API

- MySQL 5.5.24
- MySQL++ 3.1.0
- g++ 4.6.3
- scons 2.1.0

2.3 SST/macro Interface

- SST/macro default branch, changeset 755f2b136d67 or newer

3 Implementation Details

This document walks through the implementation details of the Eiger components, giving an overview of major structures and programming practices.

3.1 Class Hierarchy

Eiger employs a rigid structure for the description of training data. This structure, enforced by the API specification as well as the interal storage mechanism, is intended to be as flexible and understandable as possible, so that users are able to formulate their model generation in whichever way is easiest. This document will lay out the basic hierarchy and interactions between classes; more information can be found in the API specification.

DataCollection Performance models are built from data contained in a DataCollection. A DataCollection serves as an identifier for a set of training data points, known as *Trials*.

Trial A Trial represents a single run of a program to collect training data. Each Trial is a single run of a particular *Application*, fed with data from a particular *Dataset*, executed on a particular *Machine*, modified by a particular environmental *Property*. A Trial serves as a unifying structure under which all Metrics are contained, including those from the Machine, the Application, and the performance metric.

Execution An execution represents the individual instrumentation tools used to collect metrics from a single Trial. The union of Metrics from all of a Trial's Executions comprize the entire set of Metrics that describe that Trial. This allows for running several different tools (emulator, simulator, etc) to collect metrics that will all go together to describe a Trial.

Application An Application is a program run to collect training data. It can be a full application, a kernel, or a microbenchmark. Several different input Datasets can be run through the same Application, i.e. different sized input matrix Datasets can be run through the same matrix multiplication Application.

Dataset A Dataset describes a particular set of input data run through an Application. A Dataset may not be shared across several Applications. DeterministicMetrics are associated with a Dataset; any run of the same Application with the same Dataset should result in the same values for DeterministicMetrics.

Machine The Machine class specifies the hardware upon which an Application is run. This class is used to contain all Metrics that are invariant across Applications that run on the same hardware.

Property The Property class is used to associate any extra environmental details associated with the state of execution of a particular Trial. This may include things such as concurrent workloads on the system or existing traffic in the network.

Metric A Metric describes a measurable quantity of the system. There are several types of Metrics:

result Performance metric the model will try to predict. Runtime, energy use, etc.

deterministic Any value that is invariant across executions of a particular execution. Static instruction counts, input data size, etc.

nondeterministic Values that may change from execution to execution. This includes measures such as dynamic instruction counts, cache miss rates, etc.

machine Values that pertain only to the Machine an Application is run on. Memory capacity, clock speeds, etc.

other Miscellanea used to describe the system by the Property class.

DeterministicMetric This class acts as a wrapper for a single result of a measured Metric that has deterministic type. Each Dataset will have one DeterministicMetric for every Metric of deterministic type.

NondeterministicMetric Analogous to DeterministicMetrics but for Metrics of nondeterministic type. This includes the result type, as by definition any performance result is dependent upon the execution of the workload on hardware.

MachineMetric Analogous to DeterministicMetric but for Metrics of machine type.

3.2 Internal Data Store

By design the API abstracts away the requirement for any particular database technology, although at this time this interface is only implemented for the MySQL database. There is a table for each of the major classes, as well as extra tables used for model serialization. Eiger does not require that the database be stored locally; rather, the API provides a way to indicate the IP address or hostname of the machine, allowing for nonlocal data storage. Further details can be found in the API documentation in Section 4. The schema describing the database is in `eiger/database/schema.sql`.

3.3 Model File Format

For integration with the SST/macro simulator, Eiger can serialize its performance models to files. Each file is formatted so that it can be read by the appropriate module within SST/macro, however there is nothing inherent to this format that prevents it from being used by different sources. More details on the exact file format can be found in `eiger/PerformanceModel.py` under the `toFile()` function. See the FAQ in Section 7 for details on exporting performance models.

4 Application Programming Interface

This document defines the API to Eiger. The primary implementation of this API uses SQL queries; an alternate implementation (with some restrictions) that preserves application code almost unchanged is described in section 8.

4.1 Requirements

- Provide the interface into and out of Eiger internal storage mechanism.
- Hide all implementation details of Eiger's storage (eg SQL commands, etc).
- API implementations can be written to map any object- oriented language to any internal storage mechanism (eg C++ to MYSQL, python to Hadoop, etc).
- The API reflects the extensible nature of Eiger.
- The API includes the specification of the Eiger storage schema.
- Provide error reporting.

4.2 Setup/Teardown

`Connect` Connects to the eiger database for this session.

Parameters

databaseLocation string indicating the location of the database
databaseName Database name
username Username with read/write access to this database
password Password for this username

4.3 Classes

`Metric` A metric is something measured about the system. There are five types:

result Result from a run of an `Application`. Runtime, energy use, etc.
deterministic Metric that can be found statically. Pertains to a particular `Dataset`.
nondeterministic Metric that can only be found by running. Pertains to a `Trial`.
machine Machine parameter. Number of cores, cache size, etc.
other Miscellanea.

Members

ID Unique identifier for each `Metric` object.

type `Metric` type. Enumeration of: result, deterministic, nondeterministic, machine, other.

name `Metric` name.

description `Metric` description.

Constructors

`Metric(type, name, description)` Creates new `Metric` from provided `type`, `name`, and `description`.

`Metric(ID)` Constructs new `Metric`, but loads members from Eiger database belonging to `ID`.

Methods

`toString()` Returns human readable string describing this object.

`getID()` Returns this object's unique ID.

`commit()` Sends this object to the Eiger database for storage.

`NondeterministicMetric` A nondeterministic metric is a metric inherent to an execution; a particular run on a particular machine. This class is used to store one nondeterministic metric from a single run.

Members

execution Points to the `Execution` this `DynamicMetric` pertains to.

metric Points to the `Metric` describing this `DynamicMetric`.

value Actual value of this metric.

Constructors

`NondeterministicMetric(execution, metric, value)` Creates a new `NondeterministicMetric` from the provided `execution`, `metric`, and `value`.

Methods

`toString()` Returns human readable string describing this object.

`commit()` Sends this object to the Eiger database for storage.

`DeterministicMetric` A deterministic metric is a metric inherent to a dataset of an application. This class is used to store one deterministic metric from an input dataset.

Members

dataset Points to the `Dataset`this `DeterministicMetric` pertains to.

metric Points to the `Metric` describing this `DeterministicMetric`.

value Actual value of this metric.

Constructors

`DeterministicMetric(dataset, metric, value)` Creates a new `DeterministicMetric` from the provided `dataset`, `metric`, and `value`.

Methods

`toString()` Returns human readable string describing this object.

`commit()` Sends this object to the Eiger database for storage.

`MachineMetric` A machine metric is a metric inherent to a particular hardware configuration. This class is used to store one machine metric from a single machine.

Members

machine Points to the `Machine` this `MachineMetric` pertains to.

metric Points to the `Metric` describing this `MachineMetric`.

value Actual value of this metric.

Constructors

`MachineMetric(machine, metric, value)` Creates a new `MachineMetric` from the provided `machine`, `metric`, and `value`.

Methods

`toString()` Returns human readable string describing this object.

`commit()` Sends this object to the Eiger database for storage.

`Trial` A trial is a run of a particular application, using a particular input dataset, on a particular machine. *Members*

ID Unique identifier for each `Trial` object.

dataCollection Points to the `DataCollection` this `Trial` is a member of.

machine Points to the `Machine` this `Trial` was run on.

application Points to the `Application` run for this run.

dataset Points to the input `Dataset` for this run.

properties Points to the `Properties` of this run.

Constructors

`Trial(dataCollection, machine, application, dataset, properties)` Creates a new `Trial` from the provided `dataCollection`, `machine`, `application`, `dataset`, and `properties`.

Methods

`toString()` Returns human readable string describing this object.

`getID()` Returns this object's unique ID.

`commit()` Sends this object to the Eiger database for storage.

`Execution` An Execution is an individual run of an instrumentation tool that collects nondeterministic metrics or result metrics. Several combine to form a trial. *Members*

ID Unique identifier for each `Execution` object.

trial Points to the trial which this `Execution` is a member of.

machine Points to the machine on which this `Execution` was run.

Constructors

`Execution(trial, machine)` Creates a new `Execution` from the provided `trial` and `machine`.

Methods

`toString()` Returns human readable string describing this object.

`getID()` Returns this object's unique ID.

`commit()` Sends this object to the Eiger database for storage.

`Property` Specifies any extra information about the trial *Members*

ID Unique identifier for each `Property` object.

name `Property` name.

description `Property` description.

Constructors

`Properties(name, description)` Creates a new `Property` from the provided `name` and `description`.

Methods

`toString()` Returns human readable string describing this object.

`getID()` Returns this object's unique ID.

`commit()` Sends this object to the Eiger database for storage.

`Machine` A machine is the hardware applications are run on. *Members*

ID Unique identifier for each `Machine` object.

name `Machine` name.

description `Machine` description.

Constructors

`Machine(name, description)` Creates a new `Machine` from the provided `name` and `description`.

Methods

`toString()` Returns human readable string describing this object.

`getID()` Returns this object's unique ID.

`commit()` Sends this object to the Eiger database for storage.

`Application` An Application is a workload or kernel run on a machine. *Members*

ID Unique identifier for each `Application` object.

name `Application` name.

description `Application` description.

Constructors

`Application(name, description)` Creates a new `Application` from the provided `name` and `description`.

Methods

`toString()` Returns human readable string describing this object.

`getID()` Returns this object's unique ID.

`commit()` Sends this object to the Eiger database for storage.

`Dataset` A Dataset is the input information given to a particular Application. For example the input matrices to a matrix-matrix multiplication application. *Members*

ID Unique identifier for each `Dataset` object.

application Points to the `Application` this `Dataset` belongs to.

name `Dataset` name.

description `Dataset` description.

created Date and time this `Dataset` was created.

url Hyperlink to website of the `Dataset`.

Constructors

`Dataset(application, name, description, url)` Creates a new `Dataset` from the provided `application`, `name`, `description`, and `url`.

Methods

`toString()` Returns human readable string describing this object.

`getID()` Returns this object's unique ID.

`commit()` Sends this object to the Eiger database for storage.

`DataCollection` A DataCollection is the set of all trials that go into the construction of a single performance model. *Members*

ID Unique identifier for each `DataCollection` object.

name `DataCollection` name.

description `DataCollection` description.

created Date and time this `DataCollection` was created.

Constructors

`DataCollection(name, description)` Creates a new `DataCollection` from the provided `name` and `description`.

`DataCollection(ID)` Creates new `DataCollection` object, but loads members from Eiger database belonging to this `ID`.

Methods

`toString()` Returns human readable string describing this object.

`getID()` Returns this object's unique ID.

`commit()` Sends this object to the Eiger database for storage.

5 Model File Format

This section describes the text format for model files generated by Eiger. The function of these files are to easily marshall and unmarshall performance model data for use in other simulators, such as SST/macro. The goal is ease of use and readability over compactness and bit-correctness.

5.1 File Format

The model captures each effect required to compute the value of a prediction. It begins with the combined principal component analysis transformations from the application and machine metrics, followed by the loadings of each predictor function, followed by an encoding of the functions themselves. The last is a list of each input metric the model requires, printed in order, i.e. the first metric listed is considered the first dimension for PCA, the second metric is the second dimension, and so on. The file is formatted as follows. Each element is separated by a newline character.

- `[`# rows in PCA matrix,# cols in PCA matrix`]` $((val_{00}, val_{01}, ...), (val_{10}, val_{11}...), ...)$
- `[`# functions in model`]` $(\beta_0, \beta_1, ...)$
- encoded functions in model, separated by newline
- name of each input metric for the model, ordered, separated by newline

5.2 Function Encodings

Each predictor function must be reevaluated when a prediction is being made. To do so, an encoding is established representing the function evaluated. This is inherently tied to the type of regression being performed, as well as the model pool used to feed the regression process. For parametric linear regressions, the function encoding is as follows.

Encoding	Function
`0`	$f(\mathbf{x}, i) = 1$
`1 i n`	$f(\mathbf{x}, i, n) = x_i^n$
`2 i j`	$f(\mathbf{x}, i, j) = x_i * x_j$
`3 i`	$f(\mathbf{x}, i) = \sqrt{x_i}$
`4 i`	$f(\mathbf{x}, i) = \log_2(x_i)$
`5 i`	$f(\mathbf{x}, i) = 1/x_i$

6 Example

In this section we will walk through the entire pipeline of data acquisition, model generation, model analysis, and output. Included in the `experiment/` directory are:

- A small matrix multiplication application already annotated with the C++ API.
- Golden reference versions of all the models created in this tutorial.
- A dump of the database containing all measurements used to create the reference models.

We will use this as the sandbox to demonstrate Eiger's functionality.

6.1 Getting Started

The process by which models are created works as follows:

1. Data is collected and stored in Eiger.
2. Models are generated.
3. Models are analyzed.
4. Models are output in a final format.

This exmple will walk through each of these steps. Before beginning, the C++ API, modeling framework, and internal database have to be set up. For instructions on how to do this, see the `README`.

If you wish to use the provided data, set up a new, empty database, then issue the following command from within the MySQL command prompt with the empty database selected:

```
source /path/to/eiger/examples/database_dump.sql;
```

This will populate the new database with the provided data.

6.2 The Application

The application we will be modeling is a very simple matrix-matrix multiplication, where each matrix is square. Progressively larger matrices are multiplied together and timed, the presumption being that the size of these matrices will be a good indicator for the performance of the matrix multiplication kernel. The included code is already annotated with the appropriate Eiger API calls so that the data can be sent to the internal Eiger data store.

To begin, the execution environment is set up. The Eiger database is connected to, the specifics of the machine, application, and workload are established. In this example, there are only two `Metrics`: the size of the input matrix(i.e. number of rows and columns), and the execution time per data point. Each iteration of the outer loop then marks a new `Trial` and `Execution`, as well as marking down the values for the individual `Metrics`. Clean up at the end disconnects from the Eiger internal data store.

It is important to note that each object needs to be send to the database, performed by the `commit()` member functions. This flushes the data to the database so that it can be used for modeling.

Compiling the application requires indicating the location of the C++ API include directory, typically `/usr/local/include/eiger`, and the API library, e.g.

```
g++ -I/usr/local/include/eiger matmul.cc -o matmul -leigerInterface
```

Run the application like this:

```
./matmul
```

It will ask you for the information necessary to connect to the Eiger database.

6.3 Modeling

All the modeling functionality is built into the `Eiger.py` script. All the options and their descriptions can be found by issuing the following command:

```
Eiger.py -h
```

Since the data for the models resides in the database, the connection flags are the most commonly used:

`--db` Name of the database.

`--user` Username to get access to the database.

`--passwd` Password associated with the user.

`--host` Hostname or IP address of the machine the database is hosted on.

Making models requires the name of the `DataCollection` and the metric to predict. Here's how you make a basic model:

```
Eiger.py --db test --user username --passwd abc123 --host
localhost -t example --performance-metric runtime
```

Most analytics require an experiment to be run, as they demonstrate how well the model performs. There are two ways to explore the quality of the model produced: how closely it fits to the training data, and how well it extrapolates to unseen data.

To explore how well the model fits the training data, and to allow us to play around with some settings, set fit test and turn on the statistics:

```
Eiger.py --db test --user username --passwd abc123 --host
localhost -t example --performance-metric runtime --test-fit
--show-prediction-statistics
```

This will print out some statistics about how well the trained model fits to the training data. The golden version of this model is in the `gold.model` file.

Let's turn on some analysis figures to explore the quality of the program. Most plotting functions pertain to the principal component analysis; while PCA is performed in this example, there is only one training variable (the size of the model), which will result in only one principal component identical to this variable. Let's turn on a scatter plot of predicted versus actual performance:

```
Eiger.py --db test --user username --passwd abc123 --host
localhost -t example --performance-metric runtime --test-fit
--show-prediction-statistics --plot-performance-scatter
```

One important setting that affects the quality of the model is the *threshold*. This determines how much additional benefit an added term to the model must have to be retained. Compare difference in root mean squared error when the threshold goes from 0.01:

```
Eiger.py --db test --user username --passwd abc123 --host
localhost -t example --performance-metric runtime --test-fit
--show-prediction-statistics --plot-performance-scatter --threshold
0.01
```

up to 0.1:

```
Eiger.py --db test --user username --passwd abc123 --host
localhost -t example --performance-metric runtime --test-fit
--show-prediction-statistics --plot-performance-scatter --threshold
0.1
```

The golden model files for these models are `gold-threshold-0.01.model` and `gold-threshold-0.1.model`, respectively. For more information on threshold, see the Eiger WPEA12 paper, included in the `documentation` directory.

There are many more flags for specifying subsets of `DataCollections` to use, how to vary principal components, as well as many more plotting functions. Please see the Eiger help command for more details:

```
Eiger.py -h
```

6.4 Saving Models

Models are only saved by using the *output* flag to Eiger, like so:

```
Eiger.py OTHER_FLAGS -o
```

Here the model, which is constructed under the constraints in the other flags (not shown), is output into a file with a name of the format `DATACOLLECTION.model`. This file is a human-readable text file. In general, it contains the principal component analysis rotation matrix, the model functions, their weights, and the names of the metrics that this model requires. For more details on the file format of the models, see Section 5.

7 Frequently Asked Questions

Frequently used functionality, errors, and questions.

Question How do I install the model generation tools?

Answer All the modeling and visualization tools are written in Python. See the README in the Eiger root directory for more information on installing.

Question How do I generate a model?

Answer All model generation, visualization, and serialization tasks are performed through the `Eiger.py` script. Model generation tasks require the DataCollection ID of the data within the Eiger database used to train the model. As well there are many flags that can be set to specify settings and procedures for building a model, experimenting on that model, performing analyses, and visualizing results.

This command will generate a model and do nothing else:

```
Eiger.py -t <Training DataCollection ID>
```

You can then specify a DataCollection to test the model against:

```
Eiger.py -t <Training DataCollection ID> -e <Experiment DataCollection
ID>
```

A full listing of flags and their functions can be found by running the following command:

```
Eiger.py -h
```

Question How do I visualize the performance or structure of my models?

Answer There are many analysis and visualization tools built into the `Eiger.py` script.

To draw the scree graph illustrating the distribution of explained variance across application principal components:

```
Eiger.py <other flags> --plot-application-scree
```

To draw a scatter graph of the predicted versus actual values for a given experiment:

```
Eiger.py <other flags> --plot-performance-scatter
```

A full list of analysis flags and their functions can be found by running:

```
Eiger.py -h
```

Question How do I export my model for use in SST/macro?

Answer The following flag can be used to output a generated model for consumption by SST/macro:

```
Eiger.py <other flags> --output MODEL_FILENAME
```

8 Non-SQL Application Programming Interface

This document defines the Non-SQL API to Eiger, for convenience named *fakeeiger*.

8.1 Requirements

- Provide as close to the SQL-based Eiger API as possible without using SQL where mysql is unavailable.
- Permit multiple executions to be accumulated into the same data set.
- Provide a mechanism to load the non-sql results into mysql.
- Generate obvious errors (preferably at compile time) where incompatibilities exist.

8.2 Differences with MySQL-based Eiger

8.2.1 Runtime behavior

Libfakeeiger replaces interprocess communication (and possible database file reads) with a lightweight in-process approximation of the database behavior and filesystem writes to a pair of fixed files "*fakeeiger.log*" and "*fakeeiger.offsets*". Fakeeiger.log is written as eiger object commits occur and contains the data needed to read a collected result set into a mysql database using the feloader utility. Fakeeiger.offsets is written at the end of the run and contains the integer id of the next execution and trial; these ids are needed if multiple runs are to be appended to the same fakeeiger.log. Clearing the offsets (and existing data) is done by removing or renaming the fakeeiger.* files. If fakeeiger.offsets is not present, fakeeiger.log is overwritten.

Because of the node-local, in-process nature of fakeeiger, it is not inherently thread-safe. Eiger calls should typically be made only from the lead thread when profiling OpenMp applications with fakeeiger. Fakeeiger log files generated by multiple processes in an MPI parallel job are *independent* and will contain identifier conflicts. Useful analysis of compute-bound code sections can still occur under these conditions.

8.2.2 Identifiers

Using MySQL permits nearly arbitrary combining of performance experiments into a single database. Part of this functionality involves providing unique integer identifiers for metadata items. In some cases, creating the same kind of item by the same name results in a database query to check if the metadata combination pre-exists; if the item exists, the integer ID is read from the database rather than generated. Obviously, this cannot be done efficiently without MySQL (or even with MySQL on very slow parallel file systems). The fakeeiger solution is to recognize that a restricted, but still useful, pattern of Eiger use is for a

given application (across distinct runs) to declare all the metadata in the same order every time, resulting in the same integer identifiers. Within a run, where metadata definitions are repeated, a map is used to ensure appropriate common identifiers are used. Where identifiers must be distinct across all runs, offsets are stored at the end of a run and read at the beginning as previously noted.

8.2.3 Compilation

As the Eiger API is based on concrete classes rather than a functional interface, fakeeiger reimplements those classes in an alternate header. Include statements must be changed from eiger.h to fakeeiger.h. As the use of eiger itself is likely to be conditional, we provide an example of controlling both the use and flavor of eiger by adding -D options to the compiler invocation.

```
%# compile line

% g++ -D_USE_EIGER -D_USE_FAKEEIGER $OTHERFLAGS app.cxx

%# include block from app.cxx

  #ifdef _USE_EIGER
  #ifdef _USE_FAKEEIGER
  #include "fakeeiger.h"
  #else
  #include "eiger.h"
  #endif
  #endif
```

8.2.4 Linking

As the Eiger API is based on concrete classes rather than a functional interface, fakeeiger provides an alternate library, *libfakeeiger*, which implements all the data writing functions of libeiger. Lacking mysql support, libfakeeiger provides none of the data reading functionality of libeiger, except where it is faked as described under Runtime behavior. In particular, the object constructors which read an object by its integer Id from the database are not available; this should produce obvious errors at compile and link times. These functions are clearly marked with macro NEEDSQLREAD in fakeeiger.h.

8.2.5 Output

The output always goes to fakeeiger.log, which normally will be in the directory from which the eigerized program is executed. The output may be edited at the top to adjust the name of the database or other parameters before loading into the database if needed. Loading from fakeeiger.log is done by running feloader while in the directory where fakeeiger.log resides.

8.3 Possible improvements

Filename fakeeiger.log is hardcoded. This could be changed but has not thusfar proved necessary for practical work.

Format Fakeeiger uses a human-readable, line-oriented format. A speed improvement (at the expense of debugging ease) would be to convert to a binary file format in fakeeiger.log.

Build The fakeeiger header arrangements could be shifted and the libraries unified so that application code requires no changes except the addition of -D_USE_FAKEEIGER to the compilation. This would require defining eiger in terms of a macro or namespace trick.

Thread safety The fakeeiger api is not intentionally thread-safe. To date, it has only been used to collect data from single processes or from multiple threads/processes where a leader handles performance data logging. A simple improvement would be to open one log-file per process, perhaps by suffixing the process id to fakeeiger.log. To support multithread or multi-process use, a revised FakeEigerLoader class will be needed to coordinate merging of various IDs. Alternately, analysis could be done rankwise and fakeeiger would need to be slightly revised to incorporate process rank into the log filename.

9 Parallel Non-SQL Application Programming Interface

This document defines the MPI non-SQL API, for convenience named *mpifakeeiger*.

9.1 Requirements

- Provide as close to the SQL-based Eiger API as possible without using SQL where mysql is unavailable.
- Permit multiple executions and process ranks to be accumulated into the same data set.
- Provide a mechanism to load the non-sql results into mysql.
- Generate obvious errors (preferably at compile time) where incompatibilities exist.

9.2 Differences with serial fake Eiger

9.2.1 Runtime behavior

Libmpifakeeiger replaces libfakeeiger and writes to a fixed file set "*mpifakeeiger.*.log*" where * is the process rank in the mpi job. The assumptions needed to prop-

erly apply mpifakeeiger are less restrictive than the assumptions for fakeeiger. Instead of requiring metadata commits always happen in the same sequence, mpifakeeiger allows any sequence but places a restriction on the naming of eiger::Dataset objects. Mpifakeeiger.log is written as eiger object commits occur and contains the data needed to read a collected result set into a mysql database using the mfeloader utility. There is no offsets file needed as in the simpler fakeeiger case, as the process of merging the data into a single database automatically handles output from additional runs. Log files generated by multiple ranks in an MPI parallel job are *independent* and will contain Dataset identifier conflicts. The mfeloader application handles resolving these conflicts if (and only if) Dataset names map equivalently to experimental parameters on different MPI ranks.

Because of the in-process nature of mpifakeeiger, it is not inherently thread-safe. Eiger calls still should typically be made only from the lead OMP thread within an MPI rank when profiling OpenMp hybrid applications with mpifakeeiger.

9.2.2 Identifiers

Using MySQL permits nearly arbitrary combining of performance experiments into a single database. Part of this functionality involves providing unique integer identifiers for metadata items. In some cases, creating the same kind of item by the same name results in a database query to check if the metadata combination pre-exists; if the item exists, the integer ID is read from the database rather than generated. Obviously, this cannot be done efficiently without MySQL (or even with MySQL on very slow parallel file systems).

The mpifakeeiger solution is to recognize that a restricted, but still useful, pattern of Eiger use is for an MPI application (across runs or ranks) to declare all Dataset objects with names that encode the experimental parameters, resulting in the same integer identifiers. This encoding can be consistently and almost effortlessly applied with machine-based techniques [**?**] beyond the scope of this section. For completeness, a hand-crafted example follows:

```
eiger::ApplicationID aid = ...;
...
// build the dataset name based on the upper bounds array
std::ostringstream dsbuf;
dsbuf << "force_loopnest_3d";
for (int i=0;i<3;i++) {
  dsbuf << "_" << upper[e];
}
std::string dsname = dsbuf.str();
eiger::Dataset ds(aid, dsname, "noDesc", "noURL");
ds.commit();
```

9.2.3 Compilation

As the Eiger API is based on concrete classes rather than a functional interface, mpifakeeiger reimplements those classes in an alternate header. Include statements must be changed from eiger.h to mpifakeeiger.h. As the use of eiger itself is likely to be conditional, we provide an example of controlling both the use and flavor of eiger by adding -D options to the compiler invocation.

```
%# compile line

% g++ -D_USE_EIGER -D_USE_FAKEEIGER $OTHERFLAGS app.cxx

%# include block from app.cxx

  #ifdef _USE_EIGER
  #ifdef _USE_FAKEEIGER
  #include "mpifakeeiger.h"
  #else
  #include "eiger.h"
  #endif
  #endif
```

In this example, OTHERFLAGS must contain the options needed to find the mpifakeeiger library.

Construction of libmpifakeeiger is not presently handled by the eiger SConstruct file. A GNUmakefile is provided, which will need minor modifications in the variables defining the flags for MySQL and MPI dependencies.

9.2.4 Linking

As the Eiger API is based on concrete classes rather than a functional interface, fakeeiger provides an alternate library, *libmpifakeeiger*, which implements all the data writing functions of libeiger. Lacking mysql support, libmpifakeeiger provides none of the data reading functionality of libeiger, except where it is faked as described under Runtime behavior. In particular, the object constructors which read an object by its integer Id from the database are not available; this should produce obvious errors at compile and link times. These functions are clearly marked with macro NEEDSQLREAD in mpifakeeiger.h.

9.2.5 Output

The output always goes to mpifakeeiger.*.log, which normally will be in the directory from which the eigerized program is executed. The output may be edited at the top to adjust the name of the database or other parameters before loading into the database if needed. Loading from mpifakeeiger.*.log is done by running mfeloader while in the directory where mpifakeeiger.*.log resides.

9.2.6 Output Format

To address scalability and file size issues, the mpifakeeiger log format (while remaining ASCII for data readability) reduces the keywords to single characters, as documented in fakekeywords.h. This reduction is optional: building the mpifakeeiger library with -DFAKE_KEYS_FULL will enable the long keywords for debugging.

9.3 Possible improvements

Filename mpifakeeiger.*.log is hardcoded. This could be changed but has not thusfar proved necessary for practical work.

Format Fakeeiger uses a human-readable, line-oriented format. A potential speed improvement (at the expense of debugging ease) would be to convert to a binary file format in the log. This would not necessarily lead to smaller output files.

Build The mpifakeeiger header arrangements could be shifted and the libraries unified so that application code requires no changes except the addition of -D_USE_FAKEEIGER to the compilation. This would require defining eiger in terms of a macro or namespace trick.

Thread safety The mpifakeeiger api is not intentionally thread-safe. To date, it has only been used to collect data from single processes and MPI processes where ranks are single-threaded.