EARL - API Documentation

High-Level Trace Access Library

Version 2.0 / October 1, 2004
Technical Report ICL-UT-04-03

Felix Wolf

Abstract

EARL is a high-level interface for accessing EPILOG event traces and can be used to write advanced trace-analysis software. EARL provides random access to single events and computes the execution state at the time of a given event as well as links between pairs of related events. EARL is implemented in C++ and offers a C++ and a Python class interface. This document describes the abstractions in terms of which the event trace is represented in EARL and how to efficiently access them using the C++ or Python API.

Contents

1 Introduction

An event trace is a chronologically sorted sequence of runtime events recorded during program execution that can be used to analyze program behavior. In the KOJAK performance-analysis environment [5, 6], event traces are used to identify patterns of inefficient execution.

KOJAK stores the event traces generated at runtime in the EPILOG binary trace-data format [7]. EPILOG traces consist of definition records and time-stamped event records. Event records describe the dynamic program behavior and reference objects that are defined in definition records. By letting event records store only references to those objects, trace file size can be reduced since an object, such as a region, is referenced many times.

To simplify the development of advanced trace-analysis software, KOJAK provides EARL (Event Analysis and Recognition Library), a high-level interface for accessing and processing EPILOG event traces. EARL offers the following functionality:

- Random access to single events
- Access to the execution state at the time of a given event
- Links between pairs of related events
- Various statistical functions

EARL can be used for a large variety of trace-analysis tasks. The main purpose of EARL within KOJAK is to simplify the specification of execution patterns representing performance problems within the EXPERT analyzer [8] and, thus, to allow an easy extension and customization of the pattern base used in the analysis process. The first prototype of EARL was completed in 1998 as part of a master's thesis [4].

Section 2 introduces the abstractions in terms of which an event trace is presented to the user. EARL is implemented in C++ and offers a C++ and a Python class interface. The Python interface's main advantage is the ability to use EARL interactively, which is useful especially for those unfamiliar with the abstractions it provides. Section 3 explains how to use the C++ API and gives a small code example. After that, Section 4 presents the internal buffer mechanisms that support efficient random access and tells how to configure them for maximum efficiency. Finally, Section 5 briefly describes the Python binding and how it differs from the C++ binding.

Important: This version of EARL supports EPILOG version 1.1, which is included in KOJAK version 2.0b. Please see the EPILOG 1.1 specification `$(PREFIX)/doc/epilog.ps`[1] for details.

2 Data Model

EARL is based on a simple object-oriented data model, whose simplicity is derived from the fact that all higher-level abstractions, such as execution states and links between related events, are expressed in terms of event sets or event references, thus never leaving the familiar notion of an event.

[1] `$(PREFIX)` is the KOJAK installation directory.

2.1 Abstractions

An *event trace* is a chronologically sorted sequence of events representing one program run of an MPI, OpenMP, or hybrid application. The event trace offers random access to its events including the execution state at the time of a given event, as well as information on program and system resources involved in the program execution, such as source-code regions and processes.

The central abstraction in EARL is an *event*. Every event has a type, a time stamp, and a location, which answers the questions what happened, when it happened, and where it happened, respectively. In addition, an event may provide type-specific attributes including links to related events.

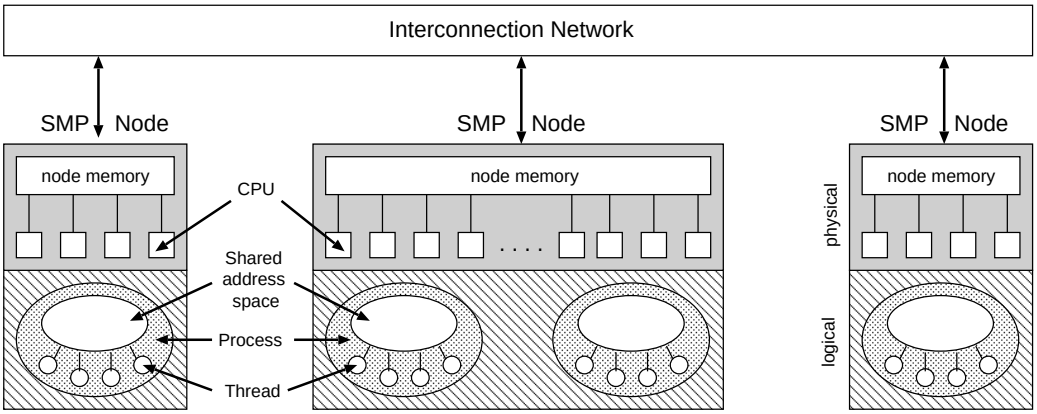

Figure 1: A parallel computer with SMP nodes.

The program resources represented in an event trace include files, regions, and call sites. A *file* is simply string containing a file name. A *region* is a source-code section that can represent a function, a loop, an OpenMP construct, or an arbitrary user-defined section. A *call site* is a source-code location where the control flow moves from one region to another. Although it is called "call site", it does not need to be involved in a function call. In EARL, a loop entry can also be a call site because the control flow moves from the enclosing region to the the loop.

The system resources associated with an event trace form a hierarchy consisting of machines, nodes, processes, and threads. *Machines* can be made up of multiple (potentially SMP) *nodes*. Each node can host multiple *processes*, which in turn can spawn multiple *threads*. This model mirrors one or more parallel computers with SMP nodes (Figure 1) and can also accommodate more traditional non-SMP, single-SMP, or simple desktop architectures. An event *location* is a tuple consisting of a machine, a node, a process, and a thread. A location is basically a thread that includes information on the process, the node, and the machine it is associated with. A single-threaded process always has one explicit thread because in EARL the thread level is mandatory. An MPI *communicator* is a special type of resource referenced by MPI communication events and is modeled as a group of processes.

Also, some events may store the values of certain system *metrics*, such as the number of floating point operations executed. A metric may represent the count of event occurrences (e.g., from a hardware counter) across an interval, an occurrence rate measured across an interval, or the current value of a metric, such as the current memory utilization.

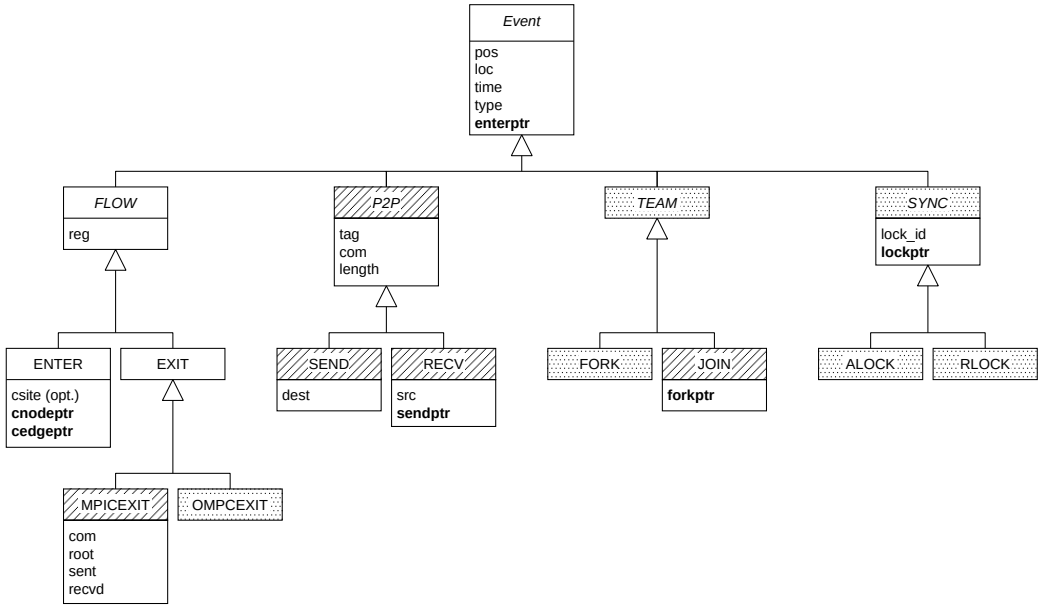

Figure 2: Hierarchy of event types.

2.2 Event Model

The event model is defined by a hierarchy of abstract and concrete event types, which is shown in Figure 2 using UML notation [1]. Abstract event types do not appear in the event trace, they are used only to isolate commonalities in the model. In the figure, abstract event types have been distinguished by writing the type names in italics. The arrows illustrate an inheritance relationship with respect to the type attributes, that is, an event type inherits all attributes from its ancestors. Hatched boxes represent MPI-specific types, whereas spotted boxes represent OpenMP-specific types. Table 1 explains the semantics of the event types and attributes.

In addition to the attributes listed in Table 1, REGION events may also carry values of system metrics, such as hardware counters. If the event trace defines metrics, every FLOW event is required to carry one value for each system metric defined in the trace.

To be able to interpret the records contained in an EPILOG event trace, EARL relies on the following validity constraints, which are already part of the EPILOG specification, but which are repeated here for clarity:

- The regions must be left in the opposite order they are entered. That is, the region that has been entered last must be left first.
- FORK and JOIN events are only generated by the master thread. The FORK event before entering and the JOIN event after leaving the parallel region.
- A SEND event must always appear before its corresponding RECV event.
- In hybrid MPI/OpenMP applications, SEND and RECV events are only allowed to be generated by the master thread.

Table 1: Event types and attributes.

Event Type	Description
EVENT	(abstract) general event
FLOW	(abstract) change of control flow
ENTER	entering a region
EXIT	leaving a region
MPI	
MPICEXIT	leaving an MPI collective communication operation including a barrier
P2P	(abstract) MPI point-to-point communication
SEND	sending a message
RECV	receiving a message
OpenMP	
TEAM	(abstract) change of parallelism
FORK	starting a parallel region
JOIN	terminating a parallel region
SYNC	(abstract) lock synchronization
ALOCK	lock acquisition
RLOCK	lock release

Attribute	Description
cedgeptr	least recent ENTER event visiting the parent call path
cnodeptr	least recent ENTER event visiting the same call path
csite	call site
enterptr	ENTER event of the enclosing region instance
loc	location
pos	relative position (1-n) within the event trace
reg	region
time	time stamp
type	event type
MPI	
com	communicator associated with a communication operation
dest	destination location of a message
length	message length
recvd	bytes received during a collective operation
root	root location of an MPI collective operation
sendptr	SEND event to a given RECV event
sent	bytes sent during a collective operation
src	source location of a message
tag	message tag
OpenMP	
forkptr	FORK event to a given JOIN event
lock_id	identifier of the lock object used for synchronization
lockptr	SYNC event that performed the last change of a lock's ownership status

2.3 Higher-Level Abstractions

To simplify the development of trace-analysis tools, EARL provides the following higher-level abstractions that are useful to easily identify related events:

- Pointer attributes
- Execution states

Pointer attributes, the first class of higher level abstractions, are event attributes that refer to another related event. For example, the attribute *sendptr* points from a RECV event to the corresponding SEND event. In Figure 2, pointer attributes are printed in bold face. They can be identified in Table 1 by a name ending in "ptr".

The second class of higher-level abstractions reflects different aspects of the program's overall execution state. The overall execution state consists of a set of (component) states, each of which represents one aspect of the overall state, such as the call stack or the message queue. EARL models each component state as a set of events. These sets are stepwise transformed by the sequence of events making up the trace file. That is, an event causes a state transition altering the event set representing the component state by either removing elements and/or adding itself to the set. Thus, for every component state, an event trace defines a *state sequence*. The initial state is always the empty set. Transition rules define how a state is transformed by an event into its successor state.

For example, EARL maintains a region (call) stack for every location. The initial stack is empty. Whenever an ENTER event occurs, it is added to the stack, and whenever an EXIT event occurs, the corresponding ENTER event is removed from the stack. Note that the state set derives its stack structure from the implicit ordering of events. For more information on the underlying theory please refer to [5]. EARL provides the following state information:

- One region stack per location that remembers all ENTER events of active region instances at a location.
- One inherited region stack per location that remembers all ENTER events of active region instances at a given location. If the location represents a slave thread, EARL adds the (inherited) region stack of the corresponding master at the time when the process starts multi-threaded execution (i.e., the time of the preceding FORK event). This is needed to track the stack of a slave all the way up to the main function even if it was created somewhere in the middle of execution.
- One message queue per location pair (*src*, *dest*) that remembers all SEND events of messages currently being transferred from *src* to *dest*. According to the restriction that MPI statements are only allowed to be executed by the master thread, there are message queues only for source and destination locations that represent master threads.
- All MPICEXIT events belonging to the same instance of an MPI collective communication operation including a barrier that has just been completed.
- All OMPCEXIT events belonging to the same instance of an OpenMP parallel construct that has just been completed.
- The call tree encoded as the set of ENTER events having visited a certain call path for the first time.

3 C++ API

Before using the C++ API, you need to include `<earl.h>` in your source code, which will be located in the following directory after installing KOJAK:

```
$(PREFIX)/include/earl
```

To generate an executable EARL application, you also need to link against the EARL and the EPILOG base library:

```
-L$(PREFIX)/lib -learl -lelg.base
```

3.1 class EventTrace

This class provides random access to all events in the trace file including the execution state at the time of a given event. EARL includes buffer mechanisms to minimize the number of file accesses and to make unavoidable file accesses faster when retrieving a particular event.

The class also provides information on program and system resources involved in the program execution. Program and system resources consist of regions, call sites, machines, (SMP) nodes, processes and threads, MPI communicators, and metrics. Except for nodes and threads, all resources have a unique identifier between zero and $n-1$, with n being the total number of resources of each type defined in the trace. The identifier of nodes and threads is also a number between zero and $n-1$, but it is local to the machine or process it belongs to and n would be the number per machine or process. A location is a tuple (*machine*, *node*, *process*, *thread*). There is one location per thread, that is, a location represents a thread and also includes the upper-level system resources the thread belongs to. Locations have unique global identifiers from zero to $n-1$, where n is the total number of locations (i.e., threads).

Events are assigned a relative *position* from 1 to n. Execution states can refer to any event position plus zero, which corresponds to the initial state.

Methods that take an identifier in order to return the corresponding object will throw an exception of type `RuntimeError` if no such identifier exists.

3.1.1 Constructor

```
EventTrace(std::string path);
```

Creates an event-trace object from an EPILOG trace file, whose path name is supplied as argument.

3.1.2 Methods to access program and system resources

```
long get_nfiles() const;
```

Returns the total number of source-code files.

```
std::string get_file(long file_id) const;
```

Returns the name of the file with identifier `file_id`.

`long get_nregs() const;`
Returns the total number of code regions.

`Region* get_reg(long reg_id) const;`
Returns the region with identifier `reg_id`.

`long get_ncsites() const;`
Returns the total number of call sites.

`Callsite* get_csite(long csite_id) const;`
Returns the call site with identifier `csite_id`.

`long get_nmachs() const;`
Returns the total number of machines.

`Machine* get_mach(long mach_id) const;`
Returns the machine with identifier `mach_id`.

`long get_nnodes() const;`
Returns the total number of (SMP) nodes across all machines.

`Node* get_node(long mach_id, long node_id) const;`
Returns the (SMP) node with machine-specific identifier `node_id` belonging to machine `mach_id`.

`long get_nprocs() const;`
Returns the total number of processes used during execution.

`Process* get_proc(long proc_id) const;`
Returns the process with identifier `proc_id`.

`long get_nthrds() const;`
Returns the total number of threads across all processes used during execution.

`Thread* get_thrd(long proc_id, long thrd_id) const;`
Returns the thread with process-specific identifier `thrd_id` belonging to process `proc_id`.

`Location* get_loc(long loc_id) const;`
Returns the location with identifier `loc_id`.

`long get_ncoms() const;`
Returns the total number of MPI communicators.

`Communicator* get_com(long com_id) const;`
Returns the MPI communicator with identifier `com_id`.

`long get_nmets() const;`
Returns the number of metrics defined in the trace.

`Metric* get_met(long met_id) const;`
Returns the metric with identifier `met_id`.

3.1.3 Methods to access events

```
long get_nevents() const;
```

Returns the total number of events in the trace (i.e., the number of event positions).

```
Event event(long pos);
```

Returns the event at position `pos`. Note that the event object is not returned as a pointer because it is a smart object containing only a reference. (See section on class `Event` for more details.)

3.1.4 Methods to query the execution state

All methods to query the execution state have an output parameter of type `std::vector<Event>&` which returns a list of events in ascending chronological order. The list reflects the state at the moment immediately after the event at position `pos` took place. The input parameter `pos` can be any number between zero and the total number of events. If zero is specified, the methods will return the initial state, which is always empty.

```
void stack(std::vector<Event>& out, long pos, long loc_id);
```

Returns in `out` all ENTER events belonging to active region instances at location `loc_id`, which corresponds to the (region) call stack at this location.

```
void istack(std::vector<Event>& out, long pos, long loc_id);
```

Returns in `out` the result of `stack()` if `loc_id` is a master thread. If `loc_id` is a slave thread, `istack()` adds (i.e., inherits) the stack of the master at the time when the process starts multi-threaded execution (i.e., the time of the preceding FORK event).

```
void queue(std::vector<Event>& out, long pos,
           long src_id = -1, long dest_id = -1);
```

Returns in `out` all SEND events of messages currently in transit from location `src_id` to location `dest_id`. Specifying `-1` for one of the end points is interpreted as from any or to any location.

```
void mpicoll(std::vector<Event>& out, long pos);
```

Returns in `out` all `mpicexit` events belonging to an instance of a MPICEXIT collective communication operation or a barrier that has been completed by the event at position `pos`. If the event at this position did not complete such an operation, `out` is left empty.

```
void ompcoll(std::vector<Event>& out, long pos);
```

Returns in `out` all `ompcexit` events belonging to an instance of a OpenMP parallel construct or a barrier that has been completed by the event at position `pos`. If the event at this position did not complete such an operation, `out` is left empty.

3.1.5 Methods to query the call tree

Similar to the call stack, the call tree is also considered as an aspect of the execution state. The call tree is empty at the beginning and evolves as execution progresses. The results returned by the methods taking a position parameter do not reflect any portions of the trace following the event at the specified position. Specifying `-1` as the position is equivalent to specifying the last position. Only methods that do not need a position parameter automatically reflect the entire trace. Note that those methods may require EARL to read the entire trace file.

The call tree is modeled as the set of ENTER events that visit a call path the first time during the entire run regardless of the location. That is, there is only one call tree for all locations. A call-tree node is an ENTER event which is part of the call tree. The call tree may have multiple roots, for example, if a parallel program was started using multiple different executables. Note that the call tree consists of a subset of all ENTER events. The functions listed below that require a parameter named `cnode`, expect the position of an ENTER event that is part of this subset.

The *cnodeptr* attribute of an ENTER event points to the ENTER event that visited the current call-tree node the fist time. Thus, a member of the call tree can be easily identified by a *cnodeptr* attribute pointing to itself. While the *cnodeptr* attribute points to the current node, the *cedgeptr* attribute points to the parent node.

```
void calltree(std::vector<Event>& out, long pos = -1);
```

Returns in `out` all ENTER events belonging to the call tree.

```
void ctroots(std::vector<Event>& out, long pos = -1);
```

Returns in `out` all ENTER events representing roots of the call tree.

```
void callpath(std::vector<Event>& out, long cnode);
```

Returns in `out` the sequence of call-tree nodes (i.e., ENTER events) from the root to the node with position `cnode`.

```
void ctchildren(std::vector<Event>& out, long cnode);
```

Returns in `out` the child nodes (i.e., ENTER events) of the node with position `cnode`.

```
long ctvisits(long cnode);
```

Returns the number of times the call-tree node with position `cnode` was visited.

```
long ctsize(long pos = -1);
```

Returns the size of the call tree as the number of events that would be returned by `calltree()`.

3.2 class Event

This class represents an event and provides methods to access event attributes. The class covers all event types, and there are no subclasses accessible to the user. Trying to access an attribute that this not defined for the type of a particular event will result in `NULL` being returned if the attribute value is supposed to be an object, `-1` if it is supposed to be an integer number, or a null event (i.e.,

an event with an empty reference) if it is supposed to be an event. Which attributes are defined for which event type can be seen in Figure 2.

From the programmer's viewpoint, the type of an event has two representations: a string representation and an enumeration type representation `etype`. The former one uses a string identical to the type name used in Figure 2 (e.g., `''ENTER''`). The latter use an enumeration constant whose name is identical to the string constant (e.g., `ENTER`).

To minimize the runtime and storage overhead of copying, instances of this class maintain only a reference to the actual object representation, which itself includes a reference counter to control its life cycle. The behavior of all methods is defined as long as the `EventTrace` object from where the event was retrieved exists. An event without a valid reference to an event object is called a *null event*. To check whether an event is a null event, use the null() method.

3.2.1 Event attributes

```
long get_pos() const;
```

Returns the relative position of the event.

```
Location* get_loc() const;
```

Returns the location of the event.

```
double get_time() const;
```

Returns the time stamp of the event.

```
etype get_type() const;
```

Returns the event type's enumeration type representation.

```
std::string get_typestr() const;
```

Returns the event type's string representation.

```
bool is_type(etype type) const;
```

Returns true if the event type is the same as or a subtype of the one supplied as argument and false otherwise.

```
Region* get_reg() const;
```

Returns the region entered or left or `NULL` if the event has no region attribute.

```
Callsite* get_csite() const;
```

Returns the call site of an ENTER event. Since the call site attribute is optional for ENTER events, an ENTER event may also return `NULL`. If the event is no ENTER event, this method returns `null`.

```
Location* get_src() const;
```

Returns the sender's location of a message.

```
Location* get_dest() const;
```

Returns the receiver's location of a message.

```
Communicator* get_com() const;
```

Returns the communicator of an MPI communication operation.

```
long get_tag() const;
```

Returns the message tag.

```
long get_length() const;
```

Returns the message length in bytes.

```
Location* get_root() const;
```

Returns the root location of an MPI collective communication operation.

```
long get_sent() const;
```

Returns the number of bytes sent during an MPI collective communication operation.

```
long get_recvd() const;
```

Returns the number of bytes received during an MPI collective communication operation.

```
long get_lock_id() const;
```

Returns the lock identifier of the OpenMP lock accessed by the event. Please note that the identifier is only used to distinguish between different locks. There are actually no lock objects in EARL. Also note that lock identifier do not need to be unique across different processes.

```
long get_nmets() const;
```

Returns the number of metrics carried by the event. The return value will be either zero or equal to the total number of metrics.

```
std::string get_metname(long i) const;
```

Returns the name of metric with identifier *i*. If this metric is not defined, a `RuntimeError` exception is thrown.

```
double get_metval(long i) const;
```

Returns the value for metric with identifier *i*. If this metric is not defined, a `RuntimeError` exception is thrown.

3.2.2 Pointer attributes

```
Event get_enterptr() const;
```

Returns the ENTER event of the currently active region instance. In the case of an EXIT event, this is the corresponding ENTER event. If there is no enclosing region instance, a null event is returned.

```
Event get_cnodeptr() const;
```

Returns the ENTER event that has visited the current call path the first time regardless of the location. The event returned represents the call-tree node currently visited.

```
Event get_cedgeptr() const;
```

Returns the ENTER event that has visited the parent call path the first time across all locations. The event returned represents the parent node of call-tree node currently visited. If there is no parent node, a null event is returned. You can use the null() method to check whether the event returned is a null event.

```
Event get_sendptr() const;
```

Returns the SEND event of the message received by a RECV event.

```
Event get_lockptr() const;
```

Returns the SYNC event that accessed the same OpenMP lock object immediately before the current event. If the lock was never accessed before, a null event is returned. You can use the null() method to check whether the event returned is a null event.

```
Event get_forkptr() const;
```

Returns the FORK event generated at the beginning of a parallel region closed by a JOIN event.

3.2.3 Miscellaneous

```
bool null() const;
```

Returns true if the event object is a null event (i.e., does not point to a valid event representation) and otherwise false.

Also, the class supports comparison operators ==, !=, <, >, <=, and => that compare events based on their position, which means they only provide meaningful results when applied to events from the same trace, since the position is only comparable across the same event trace.

3.3 class Region

This class represents a source-code region. A region is characterized by a name, a file where it is defined, begin and end line numbers, a description, and its region type.

```
long get_id() const;
```

Returns the region identifier.

```
std::string get_name() const;
```

Returns the region name.

```
std::string get_file() const
```

Returns the name of the file where the region is defined. If this information is not available, `"UNKNOWN"` is returned.

```
long get_begln() const;
```

Returns the begin line number. If the this information is not available, `-1` is returned.

`long get_endln() const;`

Returns the end line number. If the this information is not available, `-1` is returned.

`std::string get_descr() const;`

Returns the region description.

`std::string get_rtype() const;`

Returns the region type, which can be one of the following strings:

- `"FUNCTION"`
- `"LOOP"`
- `"USER_REGION"` (user defined region)
- `"OMP_PARALLEL"`
- `"OMP_LOOP"` (for/do construct)
- `"OMP_SECTIONS"` (sections construct)
- `"OMP_SECTION"` (individual section inside a sections construct)
- `"OMP_WORKSHARE"`
- `"OMP_SINGLE"`
- `"OMP_MASTER"`
- `"OMP_CRITICAL"`
- `"OMP_ATOMIC"`
- `"OMP_BARRIER"`
- `"OMP_IBARRIER"` (implicit barrier)
- `"OMP_FLUSH"`
- `"OMP_CRITICAL_SBLOCK"` (body of critical construct)
- `"OMP_SINGLE_SBLOCK"` (body of single construct)
- `"UNKNOWN"`

3.4 class Callsite

A call site is a line within a file where the control flow can move from one region to another.

`long get_id() const;`

Returns the call-site identifier.

`std::string get_file() const;`

Returns the name of the file.

`long get_line() const;`

Returns the line number.

`Region* get_callee() const;`

Returns the region to which the control flow can move from the current region.

3.5 class Metric

A metric has a name and a description. It can represent an event count, an event rate, or a sample value. An event count or rate always refers to a measurement interval, whereas a sample refers to a distinct point in time. Although for simplicity EARL returns every metric as a double, EARL indicates whether the value is meant to be an integer or a floating-point value. If the metric values refer to an interval, EARL indicates how the interval is computed. Please also read the section on performance metrics in the EPILOG specification.

`long get_id() const;`

Returns the metric identifier.

`std::string get_name() const;`

Returns the metric name. Note the EPILOG specifies predefined names for common hardware counters. Please refer to the EPILOG specification for details.

`std::string get_descr() const;`

Returns the metric description.

`std::string get_type() const;`

Tells whether the metric is an integer or a floating-point number by returning one of the following string constants:

`"INTEGER"`

`"FLOAT"`

`std::string get_mode() const;`

Tells whether the metric represents an event count, an event rate, or a sample by returning one of the following string constants:

`"COUNTER"`

`"RATE"`

`"SAMPLE"`

`std::string get_ival() const;`

If the metric represents an event count or rate, this method tells which interval the values refer to by returning one of the following string constants.

`"START"`: interval since start of measurement on a location

`"LAST"`: interval since last measurement on a location

`"NEXT"`: interval to next measurement on a location

If the metric represents a sample, `"NONE"` is returned.

Important: Note that the EPILOG library version 1.1 only generates metric information of type `"COUNTER"/"INTEGER"` measured from `"START"`.

3.6 class Location

A location is a tuple consisting of a machine, a node, a process, and a thread. There is a one-to-one mapping between locations and threads. A location is basically a thread plus information on the upper levels of the system hierarchy.

`long get_id() const;`

Returns the unique location identifier.

`Machine* get_mach() const;`

Returns the machine.

`Node* get_node() const;`

Returns the node.

`Process* get_proc() const;`

Returns the process.

`Thread* get_thrd() const;`

Returns the thread.

3.7 class Machine

The machine class constitutes the top level of the system hierarchy. A machine consist of multiple nodes.

`long get_id() const;`

Returns the unique machine identifier.

`std::string get_name() const;`

Returns the machine name. If there is no name specified, the method returns `"UNKNOWN"`.

`long get_nnodes() const;`

Returns the number of nodes associated with the machine.

`Node* get_node(long node_id) const;`

Returns the node with the identifier `node_id`. `node_id` is unique only within the machine and is a number between zero and $n - 1$, where n is the number returned by `get_nnodes()`.

3.8 class Node

A node is a physical part of a machine usually with a single address space and one or more CPUs. It can host a subset of an application's processes. A node is uniquely identified by the identifier of the machine the node belongs to in combination with the machine-local node identifier.

`long get_mach_id()`

Returns the identifier of the machine the node belongs to.

`long get_node_id() const;`

Returns the node identifier. This identifier is local to the machine the node is associated with.

`std::string get_name() const;`

Returns the node name. If there is no name specified, the method returns `"UNKNOWN"`.

`long get_ncpus() const;`

Returns the number of CPUs. Please note that the event trace might not contain the real number of CPUs, but a number greater or equal to the actual number used by the application.

`double get_clckrt() const;`

Returns the clock rate of the node in cycles per second or zero if this information is unavailable.

`Machine* get_mach() const;`

Returns the machine the node belongs to.

`long get_nprocs() const;`

Returns the number of processes hosted by the node.

3.9 class Process

A process may spawn multiple threads. A process has at least one thread.

`long get_id() const;`

Returns the unique process identifier. For MPI applications, the process identifier is equal to the rank in `MPI_COMM_WORLD`.

`std::string get_name() const;`

Returns the process name. If there is no name specified, the method returns `"UNKNOWN"`.

`Node* get_node() const;`

Returns the node the process belongs to.

`long get_nthrds() const;`

Returns the number of threads spawned by the process.

`Thread* get_thrd(long thrd_id) const;`

Returns the thread with the identifier `thrd_id`. `thrd_id` is unique only within the process and is a number between zero and $n - 1$, where n is the number returned by `get_nthrds()`.

`Location* get_loc() const;`

Returns the location corresponding to thread zero.

3.10 class Thread

A thread is uniquely identified by the identifier of the process the thread belongs to in combination with the process-local thread identifier.

`long get_thrd_id() const;`

Returns the thread identifier. This identifier is local to the machine the node is associated with. For OpenMP applications, the thread identifier is equal to the thread number returned by `omp_get_thread_num()`. Note that neither EARL nor EPILOG support nested thread parallelism.

`long get_proc_id() const;`

Returns the identifier of the process the thread belongs to.

`std::string get_name() const;`

Returns the thread name. If there is no name specified, the method returns `"UNKNOWN"`.

`Process* get_proc() const;`

Returns the process the thread belongs to.

`Location* get_loc() const;`

Returns the location of the thread.

3.11 class Communicator

EARL represents a communicator as a group of processes with an ordering defined by the rank of each process.

`long get_id() const;`

Returns the unique communicator identifier.

`long get_rank(Location* loc) const;`

Returns the rank of a location's process within the communicator. If the location is not part of the communicator, an exception of type `RuntimeError` is thrown.

`long get_nprocs() const;`

Returns the size of the communicator.

`Process* get_proc(long rank) const;`

Returns the process that corresponds to rank `rank`.

3.12 class P2Statistics

This class integrates some basic statistical functions of a data set and can be used to calculate quantiles of a very large number of values like the execution times of all instances of a particular region or the sizes of messages. The quantiles are estimates computed with the P^2 algorithm [2] which makes it unnecessary to store the complete data set. Thus, the size of an `P2Statistic` object is very small and always constant. Use of the P^2 algorithm is the reason for naming the class `P2Statistic`.

3.12.1 Constructor

`P2Statistic();`

Creates a statistics object.

3.12.2 Methods to manage the data set

`void add(double val);`

Adds a numeric value to the data set.

`void reset();`

Reinitializes the object. After applying this operation the data set is empty again.

`long count() const;`

Returns the cardinality of the data set, i.e. the number of values added so far.

3.12.3 Quantiles

`double med() const;`

Returns the median of the data set. The return value is an estimate computed with the P^2 algorithm. Requires at least five elements in the data set.

`double q25() const;`

Returns the 25% quantile of the data set. The return value is an estimate computed with the P^2 algorithm. Requires at least five elements in the data set.

`double q75() const;`

Returns the 75% quantile of the data set. The return value is an estimate computed with the P^2 algorithm. Requires at least five elements in the data set.

3.12.4 Miscellaneous

`double min() const;`

Returns the minimum of the data set. This operation requires at least one element in the data set.

`double max() const;`

Returns the maximum of the data set. This operation requires at least one element in the data set.

`double mean() const;`

Returns the mean value of the data set. This operation requires at least one element in the data set.

`double sum() const;`

Returns the sum of the elements in the data set. This operation requires at least one element in the data set.

`double var() const;`

Returns the variance of the elements in the data set. This operation requires at least one element in the data set.

3.13 Exceptions

EARL provides two different exception classes `RuntimeError` and `FatalError`, which are both subclasses of class `Error`. A runtime error is thrown if a method is used the wrong way, for example, by supplying undefined parameters. When a run-time error is thrown, the operation has failed, but the trace object is still usable. If a fatal error is thrown, the trace object has been corrupted and cannot be used anymore. Both classes provide a method to obtain an error message, which in most cases will deliver the name of the internal operation that failed.

`std::string get_msg() const;`

Returns an error message associated with the exception.

3.14 Example

The following small example illustrates how to use the EARL C++ API. The program iterates through the event trace whose name is specified as a command-line argument and prints the location identifier and the type of each event.

```
#include <earl.h>
#include <iostream>

using namespace earl;
using namespace std;

int main(int argc, char* argv[])
{
  try {
    // open trace file
    EventTrace trace(argv[1]);

    // iterate through the trace
    for ( int i = 1; i <= trace.get_nevents(); i++ ) {

      // retrieve event i
      Event event = trace.event(i);

      // print the event's location and type
      cout << event.get_loc()->get_id() << ": "
```

```
            << event.get_typestr() << endl;
    }
  }
  catch ( Error error ) {

    // print error message and exit
    cerr << error.get_msg() << endl;
    exit(EXIT_FAILURE);
  }
}
```

4 Buffer Mechanisms

While reading events from the trace file, EARL dynamically builds up a sparse index structure. State information is stored at fixed intervals in so-called bookmarks to speed up random access to events. If a particular event is requested, EARL usually need not start reading from the beginning of the trace file in order to find it. Instead, the interpreter looks for the nearest bookmark and takes the state information required to correctly interpret the subsequent events from there. Then it starts reading the trace from that position until it reaches the desired event. The distance of bookmarks can be set using the following environment variable:

`EARL_BOOKMARK_DISTANCE` (default: 10000)

To gain further efficiency, EARL automatically caches the most recently processed events in the history buffer. The history buffer always contains a contiguous subsequence of the event trace and the state information referring to the beginning of this subsequence. Thus, all information related to events in the history buffer can be completely generated from the buffer including state information. The size of the history buffer can be set using another environment variable:

`EARL_HISTORY_SIZE` (default: 1000 * number of locations)

Note that choosing the right buffer parameters is usually a trade-off decision between access efficiency and memory requirements. In particular, for very long traces with many events or very wide traces with many processes or threads, a readjustment of these parameters might be recommended.

5 Python API

The Python API is a wrapper around the C++ API that has been generated using SWIG [3]. The main advantage of the Python interface is that it enables rapid prototyping as well as interactive programming. To install the files needed for the Python interface, follow the steps explained in the KOJAK installation instructions. Before using it, make sure that your `PYTHONPATH` includes the KOJAK library directory `$PREFIX/lib`. After completing these steps, you can use EARL from Python by importing EARL using the Python import command.

```
from earl import *
```

```
t = EventTrace("trace.elg")          # open trace file
for i in range(1, t.get_nevents()):  # iterate through the trace
    print e                          # print event i
del t                                # close trace file
```

5.1 Differences between the C++ and Python API

The easiest way to become familiar with the Python API is to use it interactively from the Python shell. The Python API differs form the C++ API in that only the classes `EventTrace` and `P2Statistics` have a corresponding Python classes. Objects of all other classes are represented as Python dictionaries. The dictionary keys correspond to C++ method names and the dictionary values to the C++ methods' return values.

The dictionaries are not nested. Instead, references to other objects are expressed using identifiers. For example, a C++ `Location` object holds pointers to a machine, a node, a process, and a thread. The equivalent Python dictionary would hold a machine identifier, a node identifier, a process identifier, and a thread identifier.

```
>> print t.get_loc(1)
{'thrd_id': 2, 'mach_id': 0, 'node_id': 0, 'id': 3, 'proc_id': 0}
```

In most cases, you can translate from the C++ method name to the Python dictionary key simply by using the method name without the preceding `get_`. Python attributes holding an identifier generally end with `_id`. `-1` is used to indicate the absence of a certain object. Also, C++ methods taking an argument are not represented in the dictionary. Table 2 and 3 list the Python dictionary keys used to resemble the C++ methods of the various object types.

Events. In dictionaries representing events, pointer attributes are expressed in terms of event positions. That is, wherever a method of C++ class `Event` returns an object of type `Event`, the Python dictionary contains only the event's position. Null events are represented as `-1`.

```
>>> t.event(32)
{'reg_id': 69, 'loc_id': 15, 'csite_id': -1, 'cedgeptr': 1,
'time': 0.11698729544878006, 'enterptr': 28, 'type': 'ENTER',
'pos': 32, 'cnodeptr': 3}
```

As shown here, the event attributes referring to other objects just contain their identifiers. Table 3 includes a list of all dictionary keys representing event attributes. Also, metric values carried by an event can be accessed using the metric name as the key.

Event vectors. Some methods of the C++ class `EventTrace` use an output argument of type `std::vector<Event>` to return a list of events. The equivalent Python method does not use an output parameter. Instead, a Python list with event positions is returned. The events themselves can be obtained later using the `event()` method.

```
>>> t.stack(100, 0)
[2, 94]
```

Table 2: Python dictionary keys representing event attributes.

Event	
`cedgeptr`	position of the least recent ENTER event visiting the parent call path
`cnodeptr`	position of the least recent ENTER event visiting the same call path
`csite_id`	call site identifier
`enterptr`	position of the ENTER event of the enclosing region instance
`loc_id`	location identifier
`pos`	relative position (1-n) within the event trace
`reg_id`	region identifier
`time`	time stamp
`type`	event type as a string
< *metname* >	value of metric < *metname* >
MPI	
`com_id`	identifier of the communicator associated with a communication operation
`dest_id`	destination location identifier of a message
`length`	message length
`recvd`	bytes received during a collective operation
`root_id`	root location identifier of an MPI collective operation
`sendptr`	position of the SEND event to a given RECV event
`sent`	bytes sent during a collective operation
`src_id`	source location of a message
`tag`	message tag
OpenMP	
`forkptr`	position of the FORK event to a given JOIN event
`lock_id`	identifier of the lock object used for synchronization
`lockptr`	position of the SYNC event that performed the last change of a lock's ownership status

MPI communicators Communicators are represented in Python as lists holding one or more process identifiers. Hence, wherever a pointer to a communicator object is returned in C++, a list with process identifiers is returned in Python.

References

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language User Guide*. Addison Wesley, October 1998.

[2] R. Jain and I. Chlamtac. The P^2 Algorithm for Dynamic Calculation of Quantiles and Histograms Without Storing Observations. *Communcations of the ACM*, 28(10), October 1985.

[3] SWIG. Simplified Wrapper Interface Generator. `http://www.swig.org/`.

[4] F. Wolf. EARL - Eine programmierbare Umgebung zur Bewertung paralleler Prozesse auf Message-Passing-Systemen. Master's thesis, RWTH Aachen, Forschungszentrum Jülich, Jül-Bericht 3551, June 1998.

Table 3: Python dictionary keys corresponding to C++ methods.

Metric	
`id`	metric identifier
`name`	metric name
`descr`	metric description
`type`	metric data type
`mode`	metric mode
`ival`	metric interval
Callsite	
`id`	call-site identifier
`file`	file name
`line`	line number
`callee_id`	identifier of the callee region
Region	
`id`	region identifier
`name`	region name
`file`	file name
`begln`	begin line number
`endln`	end line number
`rtype`	region type
`descr`	region description
Machine	
`id`	machine identifier
`name`	machine name
`nnodes`	number of nodes
Node	
`mach_id`	machine identifier
`node_id`	node identifier
`name`	node name
`ncpus`	number of CPUs
`clckrt`	clock rate
`nprocs`	number of processes
Process	
`id`	process identifier
`name`	process name
`nthrds`	number of threads
`loc_id`	location identifier of the master thread
`node_id`	node identifier
Thread	
`proc_id`	process identifier
`thrd_id`	thread identifier
`name`	thread name
`loc_id`	location identifier

[5] F. Wolf. *Automatic Performance Analysis on Parallel Computers with SMP Nodes*. PhD thesis, RWTH Aachen, Forschungszentrum Jülich, February 2003. ISBN 3-00-010003-2, `http://www.fz-juelich.de/nic-series/volume17/`.

[6] F. Wolf and B. Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. *Journal of Systems Architecture*, 49(10-11):421–439, 2003. Special Issue "Evolutions in parallel distributed and network-based processing".

[7] F. Wolf and B. Mohr. EPILOG Binary Trace-Data Format. Technical Report FZJ-ZAM-IB-2004-06, Forschungszentrum Jülich, May 2004.

[8] F. Wolf, B. Mohr, J. Dongarra, and S. Moore. Efficient Pattern Search in Large Traces through Successive Refinement. In *Proc. of the European Conference on Parallel Computing (Euro-Par)*, Lecture Notes in Computer Science, Pisa, Italy, August - September 2004. Springer.