Effective Data Management for Interactive Trace Analysis

Generoso Pagano, Vania Marangozova-Martin

To cite this version:

Generoso Pagano, Vania Marangozova-Martin. Effective Data Management for Interactive Trace Analysis. [Technical Report] RT-0460, Inria - Research Centre Grenoble – Rhône-Alpes; INRIA. 2015, pp.26. hal-01155518v2

HAL Id: hal-01155518

https://hal.inria.fr/hal-01155518v2

Submitted on 27 May 2015

Effective Data Management for Interactive Trace Analysis

Generoso Pagano, Vania Marangozova-Martin

TECHNICAL REPORT
N° 460
May 2015
Project-Team MESCAL

ISSN 0249-0803 ISRN INRIA/RT--460--FR+ENG

Effective Data Management for Interactive Trace Analysis

Generoso Pagano [*], Vania Marangozova-Martin [†]

Équipe-Projet MESCAL

Rapport technique n° 460 — May 2015 — 26 pages

Résumé : Ce rapport technique décrit les améliorations proposées et mises en œuvre dans Framesoc, l'infrastructure de gestion de traces du projet SoC-Trace [2, 3], par rapport à ce qui est décrit dans le RT-447 [4], dont le contenu est considéré comme connu ici. Le thème central du présent document est la façon dont Framesoc gère la grande quantité de données de trace pour permettre une analyse de trace interactive, tout en abordant les questions de stockage dans une base de données relationnelle.

Mots-clés : Traces d'exécution, gestion de traces, infrastructure, représentation de données, interface utilisateur, interaction utilisateur, ergonomie, mécanisme publication/souscription, conception de logiciel, base de données, chargement de données.

This research is supported by the FUI SoC-TRACE project [1]

[*] INRIA, generoso.pagano@inria.fr
[†] UJF, Vania.Marangozova-Martin@imag.fr

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Effective Data Management for Interactive Trace Analysis

Abstract: This technical report describes the enhancements proposed and implemented in Framesoc, the SoC-Trace project trace management infrastructure [2, 3], with respect to what is described in the RT-447 [4], whose content is considered as known here. The central topic of the present document is how Framesoc manages huge trace data to enable interactive trace analysis, while tackling the issues of a relational database storage.

Key-words: Execution traces, trace management, infrastructure, data representation, user interface, user interaction, ergonomics, publish-subscribe, software design, relational database, data loading.

Table of contents

1 Introduction

Framesoc [5] provides a graphical user environment with several views for execution trace management and analysis. A detailed description of this environment, presented from the user point of view, is available in the Framesoc user guide [1]. In this document, we provide a more technical description of the issues, and the corresponding solutions, related to effective data management for interactive trace analysis.

The document is structured as follows. Section 2 describes the general principles adopted for data management, dealing with a relational database storage. Section 3 describes how the general principles are implemented for the different analysis views; we highlight here the view-specific solutions aimed at improving the user experience and at reducing the analysis pitfalls. Section 4 presents a performance evaluation study, concerning the technical choices implemented in Framesoc for data management. Section 5 presents our conclusions and perspectives.

2 Framesoc Data Management Principles

2.1 Data Loading and Visualization Architecture

Framesoc analysis views aim at providing different visualizations over trace data, trying to ensure a good interactivity and responsiveness. Considering that trace databases typically contain a lot of data (i.e., millions of events), data management is a major issue in maximizing user experience and supporting effective analysis. When dealing with millions of events, the simple pattern consisting in loading all relevant information in memory, computing an analysis and finally visualizing it in a view, does not scale. Indeed, on one hand, memory limits may simply prevent the loading or the analysis phases from completing successfully. On the other hand, manipulating big data requires long periods of time and the analyst may find herself waiting for the final result.

This kind of considerations motivated the choice of creating a pipeline between data loading and result visualization treatments for all Framesoc views. Figure 1 shows a generic representation of this *pipeline* pattern.

There are two active entities, a Loader Entity and a Drawer Entity. The Loader Entity is in charge of submitting a series of queries to the Framesoc database, receiving for each query a new chunk of raw data. Using this raw data, the Loader Entity updates a Shared Data Structure. The update may directly push the raw data in the data structure or require its preprocessing. On the other side, the Drawer Entity waits for new data to come into the Shared Data Structure, retrieves it and updates the view. The loading process continues until all requested data (e.g., the data in a given time interval) is loaded and displayed, or until the user stops it.

The above architecture ensures that partial results are available for the analyst almost immediately, i.e., after the first query. This partial result gets updated without freezing the user interface. The analyst can therefore start observing the trace data or stop the loading process.

Given a global data request, the pipeline pattern needs a strategy for creating the partial queries. This strategy should fulfill the following objectives:

- Limit the pitfalls in the analysis that are due to the display of partial results.
- Ensure a low latency and a high interactivity.
- Limit the overhead due to request partitioning.

The first point may easily benefit from the fact that a simple and intuitive way to provide partial trace visualizations is to use the time chronology of the execution trace. The global

[1] https://github.com/soctrace-inria/framesoc/blob/master/src/fr.inria.soctrace.maven.repository/archive/doc/framesoc_user_guide.pdf?raw=true

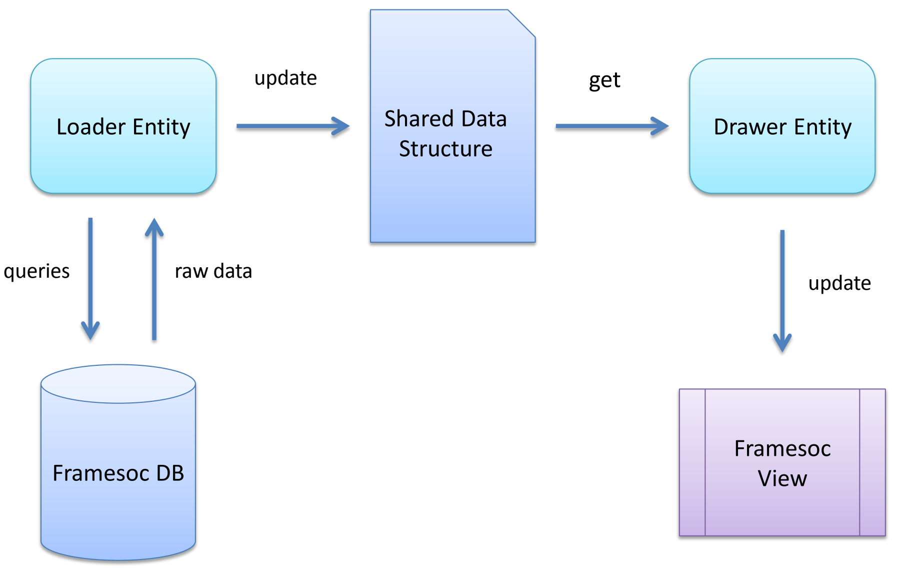


Figure 1 – General architecture of pipelined data loading and display.

request is partitioned using subsequent time intervals (query intervals), and the data related to the different intervals is displayed sequentially.

To address the other two points, we have to use a query interval that is small enough to ensure low latency, but big enough to ensure low partitioning overhead. Considering that different traces have different density of events for a given time interval, this kind of "query interval" selection must be done in terms of "number of events in the result set". A translation between this number (N) and the actual time interval (T) is done passing through the average event density (d) in the trace, according to the following relation:

$$T = N/d$$

where d is computed as:

$$d = number\_of\_events/trace\_duration$$

Using this approach we ensure that each intermediate query returns a result set containing N elements, on the average. The value of N differs for the different views and targets a good trade-off between low latency and low overhead.

This generic pipeline is instantiated for each Framesoc analysis view. The corresponding details will be described in Section 3.

2.2 Management of the Displayed Time Interval

The implemented data loading mechanism is generic and imposes no constraint on the time interval to be displayed. As a consequence, each Framesoc analysis view is capable of loading *whatever* sub interval of the whole trace. A view therefore loads only the data actually necessary for its particular analysis.

The user controls the portion of the trace displayed in the analysis view using a time management bar, located at the bottom of each Framesoc analysis view (Figure 2).

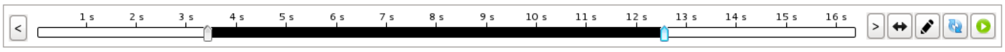


FIGURE 2 – Time management bar present in all Framesoc analysis view.

The main element of this bar is a double range time slider, whose length represents the whole trace duration. The colored part of the slider represents the trace portion that is already loaded or that we want to load by pressing the rightmost button, as explained below. Two arrow buttons on the sides of the time bar can move the blackened part to the left or to the right. The four buttons to the right respectively select the whole trace duration, specify the start and end timestamps of the selection, resynchronize the selection with the displayed trace portion and load the current selection into the view.

2.3 View Synchronization

The possibility to load any time interval in each view enables the synchronization of different views. In other terms, views can simply display the same time interval. Each Framesoc analysis view has in fact a button for each one of the other analysis views (Figure 10). The button switches to the other view while keeping the same time interval.

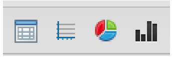


FIGURE 3 – Toolbar buttons used to switch from one analysis view to another.

In terms of implementation, this mechanism is based on the Framesoc Bus [2]. When pressing a view button, a special *visualization request* message is sent on the Framesoc Bus. In the topic, the message indicates the desired visualization (e.g., Gantt, Pie, etc.). The message content specifies the target trace, the target time interval, and the *group* of the view sending the request. A group defines a family of different types of views (i.e., Gantt, Pie, etc.) for a given trace. Framesoc, in fact, allows to open several instances of the same type of view for a given trace (e.g., several instances of the Gantt for trace t). Each different instance will belong to a different group. The time sychronization takes place only within the different views of the same group.

With this view sychronization mechanism it is easy to implement an iterative analysis workflow, where the base iteration could be following:

- Load the whole trace in the Event Density Chart, identify an interesting time zone and zoom on it using the time bar.
- Switch to the Gantt Chart visualization using the specific button, loading only the zoomed time interval.
- Reduce again the time interval using the time bar.

[2] The Framesoc Bus is a simple Publish-Subscribe bus available in the Framesoc runtime. All the details are available in the RT-447 [4]

- Visualize a given statistical metric for that time interval in the Statistics Pie Chart view, switching to it using the specific button.
- Finally switch to the Event Table visualization, to see all the details of the events in the interval.

3 Framesoc Interactive Views

In this section, we detail the implementation of the data management principles described in Section 2. We consider the different Framesoc analysis views and discuss their respective implementations. All the cited Java packages, interfaces or classes can be found in the Framesoc source code on GitHub[3].

3.1 Gantt Chart

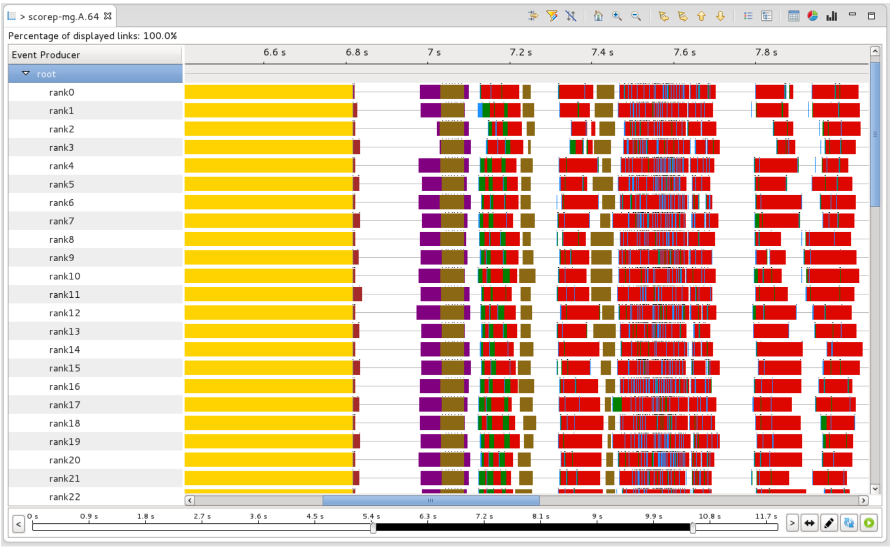


FIGURE 4 – Gantt Chart analysis view.

The Framesoc Gantt Chart analysis view (Figure 4) has been developed using the Time Graph viewer provided by the Trace Compass project[4]. It is implemented as a Gantt Chart Framesoc plugin (`fr.inria.soctrace.framesoc.ui.gantt`).

Data loading

To instantiate the general data loading architecture (cf. Figure 1), the Gantt Chart Framesoc plugin proceeds as follows.

The Loader Entity is an Eclipse `Job`. The job uses an object implementing the `fr.inria.soctrace.framesoc.ui.gantt.model.IEventLoader` interface. The Drawer Entity is a normal Java `Thread`. The thread uses an object implementing the

[3] `https://github.com/soctrace-inria/framesoc`
[4] `https://projects.eclipse.org/projects/tools.tracecompass`

fr.inria.soctrace.framesoc.ui.gantt.model.IEventDrawer interface. The IEventLoader declares the loadWindow() method, which takes as input the global time interval we want to load. The partitioning of the global request in several queries to the database event table is done in the method implementation. The results of the partial queries are stored in a fr.inria.soctrace.framesoc.ui.model.LoaderQueue, which corresponds to the Shared Data Structure in Figure 1. The elements of this queue are lists of ReducedEvent objects. A ReducedEvent contains the minimal information to draw an event (state, link or punctual event) in the Gantt Chart. When a new list is ready, the drawer thread uses the IEventDrawer to transform the ReducedEvent objects in a model understandable by the Time Graph viewer. This model is basically a hierarchy of *lines*, where a line corresponds to an event producer and contains a list of graphical events.

The Gantt Chart plugin defines an extension point[5] allowing other plugins to provide a concrete IEventLoader and IEventDrawer implementations for a given trace type. Framesoc provides default implementations of the two interfaces. These default implementations rely on the Framesoc data model semantics, supporting states, links and punctual events as they are stored in the Framesoc trace database. The default implementation of the IEventLoader performs a partitioning of the global request loading 100 000 events, on the average, for each partial query. This number has been chosen as a good compromise between low visualization latency and small partitioning overhead.

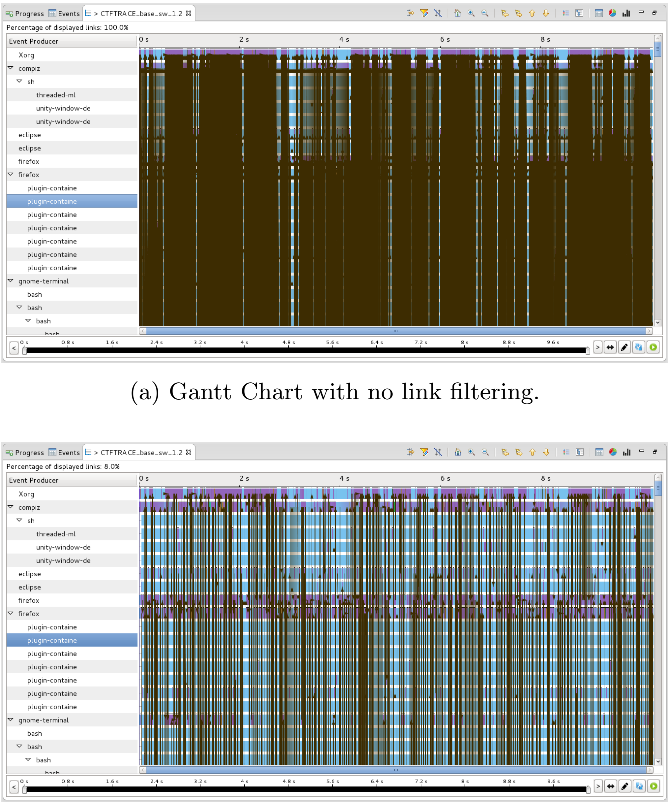
Data manipulation

The Gantt Chart viewer is capable of doing some manipulations over displayed data, either automatically, or guided by the user. The automatic manipulations performed by the Gantt Chart are state aggregation and link filtering. The guided manipulations are event type filtering and event producer filtering.

State aggregation consists in aggregating different states in a given producer time line, where there are not enough pixels to draw all of them. If the level of zoom is such that more than one state is to be represented in a given pixel, those states get aggregated (only one graphical object is represented) and a black dot is displayed on top of the point where aggregation took place. This dot informs the user that the graphical object under the dot is not an actual state, but simply the aggregate of several states. With this piece of information the analyst knows that, to draw correct conclusions over a given point of the trace, he has to zoom in more, to show all the states. This state aggregation technique limits the pitfalls of the uncontrolled graphical aggregation, which would be performed anyway by the computer graphic card or by the screen if we tried to draw all the states. Furthermore, this technique actually draws less graphical objects on the screen (at most one per pixel), so it is more efficient in both time and memory.

Link filtering consists in applying a similar idea to links, i.e., the arrows that may exist between two points of the Gantt Chart. Given that typically links connect two points belonging to two different event producer lines, it is not straightforward to easily apply a *merge* among links as we do for states. For this reason, a simpler technique is used, which consists in displaying only a subset of links, based on the amount of pixels we have. The heuristic currently implemented acts as follows. Links whose duration exceeds the time interval corresponding to a single pixel, are always displayed (non vertical arrows). For all the other links (vertical arrows), at most one link is displayed for a given pixel: the link chosen for displaying is the first one in the time interval corresponding to the current displayed pixel. Link filtering enables the control of the amount of information lost due to screen limitation: in fact, the percentage of links actually drawn in the

[5] https://wiki.eclipse.org/FAQ_What_are_extensions_and_extension_points%3F



Percentage of displayed links: 100.0%

(a) Gantt Chart with no link filtering.

Percentage of displayed links: 8.0%

(b) Gantt Chart with link filtering.

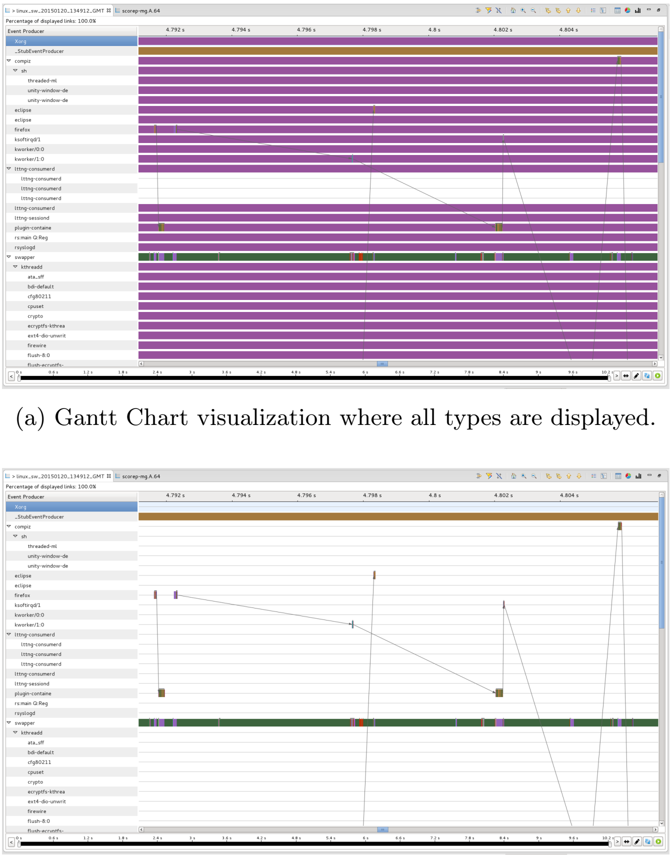
FIGURE 5 – Gantt Chart visualization without and with link filtering.

displayed time interval is shown at the top left corner of the Gantt Chart view. It is worth noting that, without link filtering, information would be lost as well, but this would be out of control, depending only on the graphic card or the screen. Furthermore the analyst would not be aware of this loss. In this case too, with link filtering we actually draw less graphical objects, thus being more efficient in memory consumption and achieving a more fluid and interactive display.

In Figure 5, we can see the difference between having or not the link filtering feature. Figure 5a shows a time interval where all links are displayed on the screen. From this representation, the only information we have is that *there are a lot of links*, but almost any other information is lost. Even the states behind the links are almost completely hidden. Figure 5b, on the contrary, provides a more useful information. Here we are still able to conclude that *there are a lot of links* (a lot of arrows are displayed even if only 8% of the actual number of links is displayed), but we can also see the different states connected by the links. Besides, having 92% less graphical objects on the screen has a significant positive impact on the latency of zoom in/zoom out operations.

One of the well-known problems of Gantt Chart representations is the lack of spatial scalability. Indeed, when a high number of event producers is present in the trace, a high number of time lines will be displayed in the Gantt chart, thus obliging the analyst to scroll over these lines to analyze its trace. This easily leads to context loss, thus making trace analysis way harder, if not impossible. A simple step to limit this problem is offering to the analyst the possibility to explicitly **filter event producers**, saying which ones she wants see and which ones she wants to hide. For example, if we are interested only in the events occurring on two producers, it is not necessary nor useful to show all the trace producers. The Gantt Chart view provides this kind of filter: pressing the corresponding button in the view toolbar, a dialog is displayed where the user can check/uncheck event producers according to her needs.

With a similar dialog, accessible through another toolbar button, the user can explicitly **filter event types**. It is possible to check/uncheck which event types we want to see and which ones we want to hide. Figure 6 shows the difference between the Gantt Chart representation where all types are shown, and the one where a dominating type has been hidden, to better focus on the other types of events.



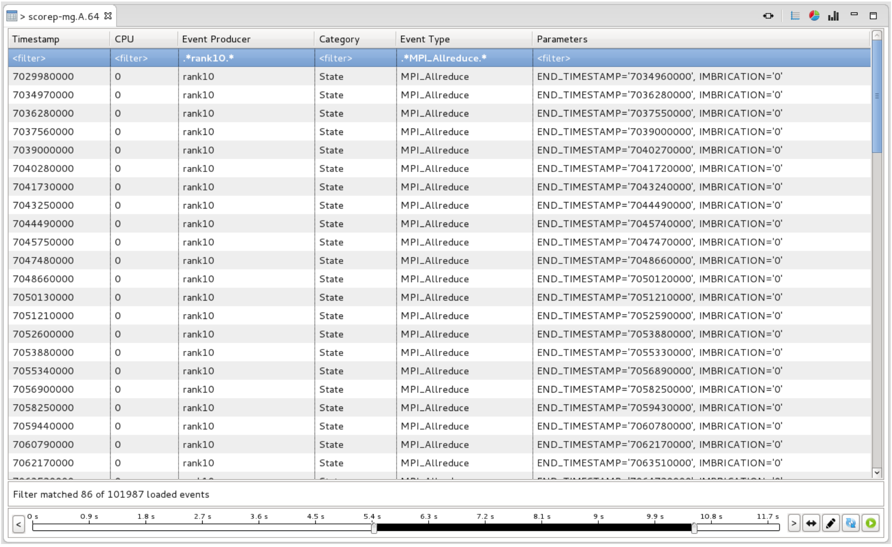
(a) Gantt Chart visualization where all types are displayed.

(b) Gantt Chart visualization after hiding a type.

FIGURE 6 – Event type hiding in Gantt Chart.

3.2 Event Table

Framesoc Event Table analysis view (Figure 7) has been developed using the Virtual Table viewer provided by the Trace Compass project. We chose the Trace Compass viewer, over the default Eclipse Table viewer, since it is more lightweight and scales better. The basic idea behind this viewer is that only displayed rows have a corresponding graphical object in memory. For all the other rows, only the information related to the row content is kept in memory in a cache.



Timestamp	CPU	Event Producer	Category	Event Type	Parameters
<filter>	<filter>	.*rank10.*	<filter>	.*MPI_Allreduce.*	<filter>
7029980000	0	rank10	State	MPI_Allreduce	END_TIMESTAMP='7034960000', IMBRICATION='0'
7034970000	0	rank10	State	MPI_Allreduce	END_TIMESTAMP='7036280000', IMBRICATION='0'
7036280000	0	rank10	State	MPI_Allreduce	END_TIMESTAMP='7037550000', IMBRICATION='0'
7037560000	0	rank10	State	MPI_Allreduce	END_TIMESTAMP='7039000000', IMBRICATION='0'
7039000000	0	rank10	State	MPI_Allreduce	END_TIMESTAMP='7040270000', IMBRICATION='0'
7040280000	0	rank10	State	MPI_Allreduce	END_TIMESTAMP='7041720000', IMBRICATION='0'
7041730000	0	rank10	State	MPI_Allreduce	END_TIMESTAMP='7043240000', IMBRICATION='0'
7043250000	0	rank10	State	MPI_Allreduce	END_TIMESTAMP='7044490000', IMBRICATION='0'
7044490000	0	rank10	State	MPI_Allreduce	END_TIMESTAMP='7045740000', IMBRICATION='0'
7045750000	0	rank10	State	MPI_Allreduce	END_TIMESTAMP='7047470000', IMBRICATION='0'
7047480000	0	rank10	State	MPI_Allreduce	END_TIMESTAMP='7048660000', IMBRICATION='0'
7048660000	0	rank10	State	MPI_Allreduce	END_TIMESTAMP='7050120000', IMBRICATION='0'
7050130000	0	rank10	State	MPI_Allreduce	END_TIMESTAMP='7051210000', IMBRICATION='0'
7051210000	0	rank10	State	MPI_Allreduce	END_TIMESTAMP='7052590000', IMBRICATION='0'
7052600000	0	rank10	State	MPI_Allreduce	END_TIMESTAMP='7053880000', IMBRICATION='0'
7053880000	0	rank10	State	MPI_Allreduce	END_TIMESTAMP='7055330000', IMBRICATION='0'
7055340000	0	rank10	State	MPI_Allreduce	END_TIMESTAMP='7056890000', IMBRICATION='0'
7056900000	0	rank10	State	MPI_Allreduce	END_TIMESTAMP='7058250000', IMBRICATION='0'
7058250000	0	rank10	State	MPI_Allreduce	END_TIMESTAMP='7059430000', IMBRICATION='0'
7059440000	0	rank10	State	MPI_Allreduce	END_TIMESTAMP='7060780000', IMBRICATION='0'
7060790000	0	rank10	State	MPI_Allreduce	END_TIMESTAMP='7062170000', IMBRICATION='0'
7062170000	0	rank10	State	MPI_Allreduce	END_TIMESTAMP='7063510000', IMBRICATION='0'

Filter matched 86 of 101987 loaded events

FIGURE 7 – Event Table analysis view.

This cache is used to get the data needed to create graphical rows on demand: only when, using the scrollbar, the row must be actually displayed, the graphical object is created.

Data loading

The data loading architecture implemented in the Event Table Framesoc plugin (`fr.inria.soctrace.framesoc.ui.eventtable`) is an instantiation of the generic one (Figure 1).

The Loader Entity is an Eclipse `Job` internally using an object implementing the `fr.inria.soctrace.framesoc.ui.eventtable.loader.IEventLoader` interface. The `IEventLoader` declares the `loadWindow()` method, which takes as input the global time interval we want to load. Concrete implementations of this method are in charge of partitioning this global request in several partial queries on the database event table. The default implementation of the `IEventLoader` loads 20000 events, on the average, for each partial query. This number has been chosen as a good compromise between low visualization latency and small partitioning overhead. The results of the partial queries are stored in a `fr.inria.soctrace.framesoc.ui.model.LoaderQueue`, which corresponds to the Shared Data Structure in Figure 1. The elements of this queue are lists of `Event` objects, and not `ReducedEvent` objects as it was in the Gantt Chart case. In fact, in this case a complete `Event` object is necessary, since all event information must be displayed. In particular, the table also displays event custom parameters, not visible in the Gantt Chart.

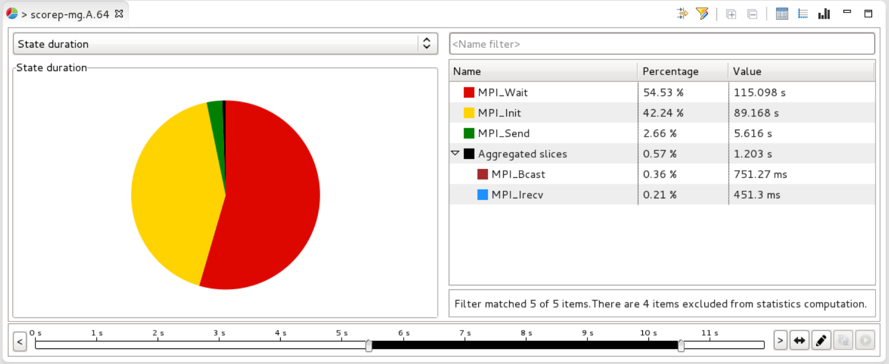
The Drawer Entity is a normal Java `Thread`, in charge of filling the `fr.inria.soctrace.framesoc.ui.eventtable.view.EventTableCache`. The drawer thread waits for new lists of `Event` to be pushed into the `LoaderQueue`. When a new list is ready, the drawer thread uses the `Event` information to fill the cache. Both this cache and the table viewer are rank based. This means that each table row corresponds to an index, and this index is used to retrieve the information related to one row from the cache. When the table is first loaded, the row at index k in the viewer, corresponds to the information accessible in the cache at index k.

Column Filtering

The Framesoc Event Table allows for filtering over the different column values, using regular expressions. To do so, the first row of the table viewer contains editable text fields. For each column, it is therefore possible to specify a regular expression to be matched by the different rows. This kind of filtering is performed within a background thread and the filtering results are progressively shown in the view.

Given the fact that both the cache and the table viewer are rank based, filtering makes it necessary to re-index the cache. In fact, when no filter is present, there is an identity relation between the table viewer index and the cache index: e.g., the row corresponding to index k in the table, corresponds to the row information accessible in the cache at index k. On the contrary, when we apply a filter, some rows will probably not be present in the table, so this identity does not stand any more. For example, if we apply a filter and the result contains only rows 3, 45 and 48, in the resulting table there will be only 3 rows whose indexes will be 0, 1 and 2 for the table viewer. When the table viewer will ask the cache for index 0, the cache will answer with what was before at index 3, and so on. To achieve this, the cache must be re-indexed. The implementation of the cache simply support this feature using an intermediate index for the remapping.

3.3 Statistics Pie Chart



Name	Percentage	Value
🟥 MPI_Wait	54.53 %	115.098 s
🟨 MPI_Init	42.24 %	89.168 s
🟩 MPI_Send	2.66 %	5.616 s
▽ ■ Aggregated slices	0.57 %	1.203 s
🟥 MPI_Bcast	0.36 %	751.27 ms
🟦 MPI_Irecv	0.21 %	451.3 ms

Filter matched 5 of 5 items.There are 4 items excluded from statistics computation.

Figure 8 – Statistics Pie Chart analysis view.

The Framesoc Statistics Pie Chart analysis view provides the possibility to compute several statistical metrics over trace events and display them in the form of a Pie Chart. Beside the Pie Chart, a tabular representation is provided as well, in order to offer some data manipulation capabilities to the analyst.

Data loading

The data loading architecture implemented in the Statistics Pie Chart Framesoc plugin (`fr.inria.soctrace.framesoc.ui.piechart`) is an instantiation of the one presented in Figure 1.

The Loader Entity is a normal Java `Thread`, internally using an object implementing the `fr.inria.soctrace.framesoc.ui.piechart.model.IPieChartLoader` interface. Concrete implementations of this interface correspond to actual statistical operators. The interface, therefore, declares a set of methods defining the specific behavior for the computation and the representation of a specific statistical variable. First of all, the interface offers a `load()` method, taking as input the trace and the time interval of interest. Concrete implementations of this method are in charge of partitioning the global request into partial requests, storing a snapshot of the computed statistical metric in a `fr.inria.soctrace.framesoc.ui.piechart.model.PieChartLoaderMap`, which corresponds to the Shared Data Structure in Figure 1. For the existing implementations, each partial request processes 100 000 events on the average, to get a good view responsiveness and a small partitioning overhead. The `PieChartLoaderMap` provides the abstraction of a map between entity names and the statistical metric value for that entity. If, for example, we are computing the state duration, this map will link each state name with the corresponding duration. This data structure differs from the queues seen for the Gantt Chart and the Event Table, since it does not store incremental data, but simply a snapshot of the computed statistical metric at a given time. If, for example, the global request over a time interval is partitioned in 10 requests over 10 subsequent time intervals, each time the data from a new time interval are processed by the `IPieChartLoader`, a new snapshot of the metric is created: this new snapshot contains, of course, also the information coming from previous time intervals. So, when a new snapshot is available, it replaces the previous one in the `PieChartLoaderMap`.

The Drawer Entity is an Eclipse `Job` waiting for new snapshots to be set into the `PieChartLoaderMap`. When a new snapshot is available, this job simply translates the content of the map into something that we can visualize directly in a pie chart and in a table. To do so, it uses two methods still provided by the `IPieChartLoader`, since they depend on the specific statistical operator considered: `getPieDataset()` and `getTableDataset()`.

The `IPieChartLoader` offers also some methods to get the colors for the different entities, the numerical format for the values, and how aggregation of small values is performed [6]. In particular, for the aggregation, the loader implementation specifies if it is supported, which is the aggregation threshold and which is the label for the aggregated slice.

The Statistics Pie Chart plugin defines an extension point allowing other plugins to provide concrete `IPieChartLoader` implementations for different statistical metrics. The Statistics Pie Chart plugin itself currently provides four extensions to this extension point, providing the following statistical operators:

- Event Producer instances: each pie slice represents the number of events having a given event producer.
- Event Type instances: each pie slice represents the number of events having a given event type.
- Link duration: each pie slice represents the duration of a given type of links.
- State duration: each pie slice represents the duration of a given type of state.

Event Type and Event Producer Filtering

The Statistics Pie Chart analysis view offers some filtering capabilities, to reduce the scope of the analysis and focus only on relevant events. In particular it is possible to interactively filter over event types and event producers, specifying which ones must be used in the computation of the selected metric. The user can check/uncheck types and producers of interest using the corresponding dialogs accessible via two toolbars buttons. When a filtering is done using this mechanism, the chart is immediately updated for the current metric. When changing the metric, the filters are kept: if necessary, the metric is recomputed to match the current filters.

[6] As described in RT-447 [4], entities whose values are smaller than a given threshold are aggregated in the pie chart in a single slice, and grouped in the table into a dedicated folder.

Data manipulation

The Statistics Pie Chart analysis view allows for some data manipulation aimed at extracting meaningful information from the computed statistical metric.

First, it is possible to **exclude from the metric computation** one or more of the entities. In fact, the default behavior when computing a metric is to consider all the corresponding trace entities: if we are computing state duration, all the state types will be considered by default. Opening the possibility to exclude entities interactively, allows the analyst to progressively ignore uninteresting information and let interesting information emerge.

Second, the user has also the possibility to **define custom groups** of entities, to immediately see their aggregated value. The user can select two or more entities and group them, defining the color and the name for the new aggregated entity.

Note that these data manipulations take place *after* the type/producer filtering described above: if an entity has been excluded because of a filter, it will not be possible to remove it or to put it in a group, since it will not be shown in the first place. In other words, it is possible to manipulate only not filtered entities.

Figure 9 shows the application of the two data manipulation techniques for the State Duration operator. In Figure 9a, all the state types are considered in the metric computation. We observe that there is a dominating state, active more than 99% of the time, which prevents to have a good understanding of what else is happening. First of all, we exclude this state from metric computation, obtaining the result shown in Figure 9b. While before all the other state types were in the aggregated slice, now we can better understand their proportions in the trace. Going even further, we want to group all the states related to the real execution: that is the states of type `*RUN*`. To this purpose, we create a new group, called `RUN_STATUS`, containing the above mentioned states. We chose the red color for this aggregated slice. The result of this manipulation is shown in Figure 9c.

Note that excluded entities can be restored, reintroducing them in the computation of the metric. Similarly, aggregated entities can be disaggregated.

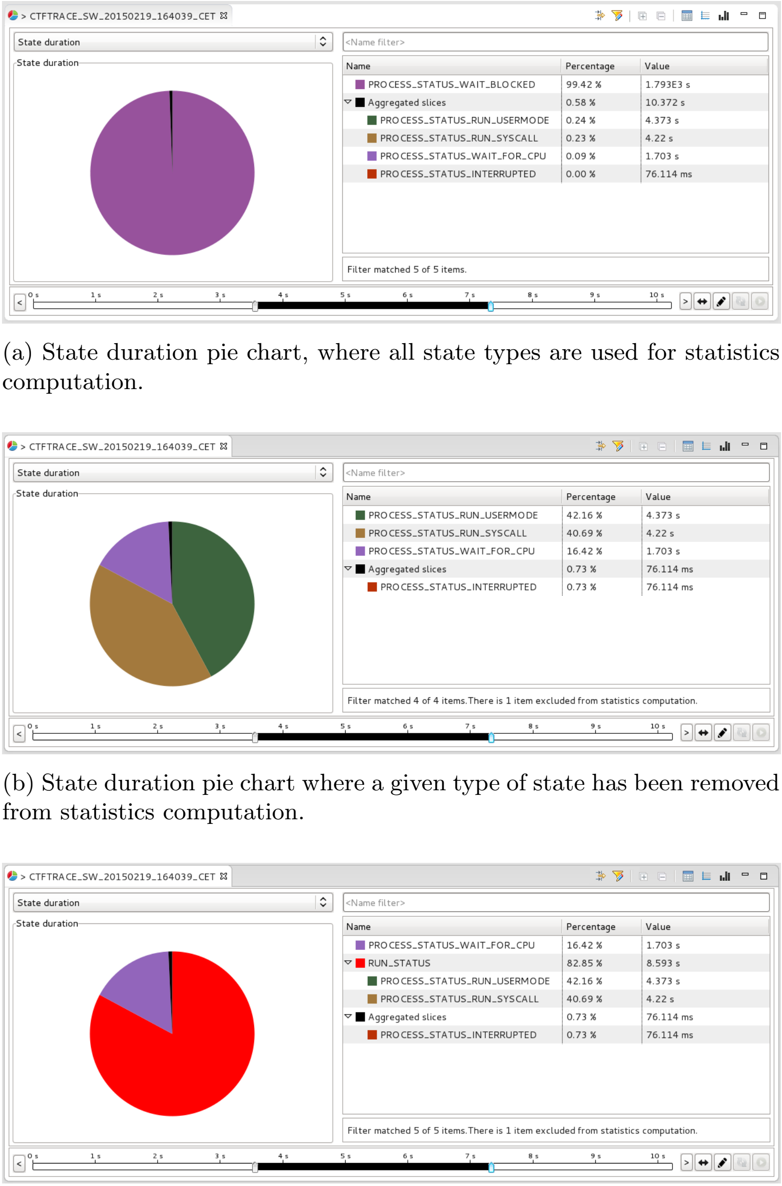
3.4 Event Density Chart

The Framesoc Event Density Chart analysis view provides a histogram representation of the event density over time. Each histogram bin corresponds to the number of events produced in the corresponding time interval. This analysis view provides the possibility to filter over time, space and types.

Data loading

The data loading architecture implemented in the Event Density Chart Framesoc plugin (`fr.inria.soctrace.framesoc.ui.histogram`) is an instantiation of the one presented in Figure 1.

The Loader Entity is a normal Java `Thread`, internally using a `fr.inria.soctrace.framesoc.ui.histogram.loaders.DensityHistogramLoader`. This class offers a `load()` method, taking as input the trace, the time interval of interest, the event types to consider and the event producers to consider. This method is in charge of partitioning the global request into partial requests, storing a snapshot of the computed event density in a `fr.inria.soctrace.framesoc.ui.histogram.model.HistogramLoaderDataset`, which corresponds to the Shared Data Structure in Figure 1. The implementation of the `load()` method partitions the requests over subsequent time intervals. Each sub-interval contains on the average



Name	Percentage	Value
PROCESS_STATUS_WAIT_BLOCKED	99.42 %	1.793E3 s
Aggregated slices	0.58 %	10.372 s
PROCESS_STATUS_RUN_USERMODE	0.24 %	4.373 s
PROCESS_STATUS_RUN_SYSCALL	0.23 %	4.22 s
PROCESS_STATUS_WAIT_FOR_CPU	0.09 %	1.703 s
PROCESS_STATUS_INTERRUPTED	0.00 %	76.114 ms

Filter matched 5 of 5 items.

(a) State duration pie chart, where all state types are used for statistics computation.

Name	Percentage	Value
PROCESS_STATUS_RUN_USERMODE	42.16 %	4.373 s
PROCESS_STATUS_RUN_SYSCALL	40.69 %	4.22 s
PROCESS_STATUS_WAIT_FOR_CPU	16.42 %	1.703 s
Aggregated slices	0.73 %	76.114 ms
PROCESS_STATUS_INTERRUPTED	0.73 %	76.114 ms

Filter matched 4 of 4 items.There is 1 item excluded from statistics computation.

(b) State duration pie chart where a given type of state has been removed from statistics computation.

Name	Percentage	Value
PROCESS_STATUS_WAIT_FOR_CPU	16.42 %	1.703 s
RUN_STATUS	82.85 %	8.593 s
PROCESS_STATUS_RUN_USERMODE	42.16 %	4.373 s
PROCESS_STATUS_RUN_SYSCALL	40.69 %	4.22 s
Aggregated slices	0.73 %	76.114 ms
PROCESS_STATUS_INTERRUPTED	0.73 %	76.114 ms

Filter matched 5 of 5 items.There is 1 item excluded from statistics computation.

(c) State duration pie chart where two types of state have been merged in a given group.

FIGURE 9 – Statistics Pie Chart interactive data manipulation for a given trace, considering the state duration metric.

the timestamps of 1 000 000 events: once again, this number has been chosen to minimize the partitioning overhead and improve the reactiveness of the view. The `HistogramLoaderDataset` directly contains a dataset understandable by the histogram viewer. Setting a snapshot into the data structure simply means setting an array containing all the timestamps found so far for the events of interest. As for the Statistics Pie Chart case, each new snapshot replaces the previous

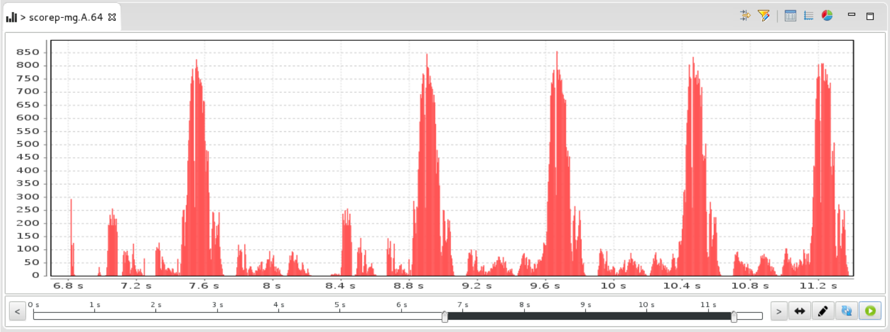


FIGURE 10 – Event Density Chart analysis view.

one.

The Drawer Entity is an Eclipse `Job` waiting for new snapshots to be set into the `HistogramLoaderDataset`. When a new snapshot is available, this job simply gets the histogram dataset and refreshes the histogram viewer.

Event Type and Event Producer Filtering

The Event Density Chart analysis view offers the possibility to interactively filter over event types and event producers, specifying which ones must be used in the computation of the event density. The user can check/uncheck types and producers of interest using the corresponding dialogs, as it happens for the Statistics Pie Chart view. When a filtering is done using this mechanism, the chart is immediately updated.

4 Performance Evaluation

This section presents some performance measurements related to Framesoc data management. The experiments have been designed in order to clarify some of the concepts anticipated in the previous sections. The obtained results motivate the data management design choices made in Framesoc.

The following analysis supposes a preliminary knowledge of the Framesoc data model [7].

Vocabulary Used

- Parameter: something influencing the behavior of the system.
- Factor: parameter chosen to variate during experiment.
- Level: value given to a factor.
- Metric: what is to be measured.
- Repetition: one execution of a given experimental configuration.

System Environment

- Framesoc version: `1.0.3`

[7] We refer in particular to the relational implementation of the *Self Defining Pattern*. Please refer to the RT-427 [3] for all the details.

- DBMS used: `SQLite 3.7.2`
- Operating System: `Linux Fedora 17, 64 bit (3.9.10-100.fc17.x86_64)`
- CPU Details:
  - model name: `Intel(R) Xeon(R) CPU E5-1660 0 @ 3.30GHz`
  - number of cores: `6`
  - hyperthreading: `active, 2 threads per core`
  - scaling governor: `performance`
- RAM: `16360588 kB`
- Hard Disk Details (output of `hdparm -i`)

```
Model=Hitachi HDS721010CLA630, FwRev=JP4OA41A, SerialNo=JP2940N03MSP5V
Config={ HardSect NotMFM HdSw>15uSec Fixed DTR>10Mbs }
RawCHS=16383/16/63, TrkSize=0, SectSize=0, ECCbytes=56
BuffType=DualPortCache, BuffSize=29999kB, MaxMultSect=16, MultSect=16
CurCHS=16383/16/63, CurSects=16514064, LBA=yes, LBAsects=1953525168
IORDY=on/off, tPIO={min:120,w/IORDY:120}, tDMA={min:120,rec:120}
PIO modes:  pio0 pio1 pio2 pio3 pio4
DMA modes:  mdma0 mdma1 mdma2
UDMA modes: udma0 udma1 udma2 udma3 udma4 udma5 *udma6
AdvancedPM=no WriteCache=enabled
Drive conforms to: unknown:  ATA/ATAPI-2,3,4,5,6,7
```

Experiments

The following subsections present three main analyses. The experiments presented in Subsection 4.1 measure the impact of database indexing on trace import time. The experiments presented in Subsection 4.2 and Subsection 4.3 consider the performance of the Gantt Chart and Event Table views respectively. The performance reflects the impact of database indexing and pipelined data loading/visualization. No experiment will be presented for the Statistics Pie Chart or the Event Density Chart use-cases, since from a data-loading point of view these use-cases are small variations on the Gantt Chart use-case, where only a subset of the event table columns is considered.

Concerning database indexing, three configurations will be considered:

- No Index: no custom index is created at trace import time.
- Timestamp Index: an index on the `TIMESTAMP` column of `EVENT` table is created at trace import time.
- Timestamp and Event-ID Index: beside the above Timestamp Index, another index on `EVENT_ID` column of `EVENT_PARAM` table is created at trace import time.

Custom indexes, when present, are created just after the actual writing of all trace events into the database, since this is more efficient than updating the index while importing the trace. Indexing time is included in the import time when custom indexes are created.

In the different experiments, we will always use the trace size, in number of events, as a factor. All trace events have 2 event parameters: this number has been chosen as an average value over different trace formats. The number of events, once the number of event parameters is fixed, translates directly into the trace database size. This translation depends on the indexing configuration used as well, as shown in Table 1. For ease of description, we will classify traces as `small`, `medium` and `big` (Table 1).

The traces used for the experiments are *fake* traces, generated and imported using the tool

Trace Alias	**Number of Events**	**Database Size**		
		No Index	**Timestamp Index**	**Timestamp and Event-ID Index**
Small	1 Million	66.3 MB	81.2 MB	121.8 MB
Medium	10 Millions	692.6 MB	843.9 MB	1.3 GB
Big	100 Millions	7.5 GB	9.3 GB	13.2 GB

Table 1 – Used traces

Temictli[8]. The event timestamps are uniformly distributed over trace duration. The database IDs are assigned incrementally.

4.1 Impact of Indexing on Trace Import

Description

When dealing with a database for data storage, indexes play a key role to have good performance. The pipelined data loading and visualization, resulting in continuous database queries requesting events from different time intervals, calls for the creation of an index on the event timestamp. Furthermore, the Event Table view use-case adds the complexity of loading event parameters too, thus calling for an index on the event ID, in the table of parameters.

The following experiments measure the impact of database indexing on import time. The detailed experimental configuration is the following:

- Factors: trace size, indexing configuration.
- Levels:
  - trace size: small, medium, big.
  - indexing: no index, timestamp index, timestamp and event ID index.
- Metric: import time.
- Number of repetitions:
  - small traces: 30.
  - medium traces: 10.
  - big traces: 5.

Analysis

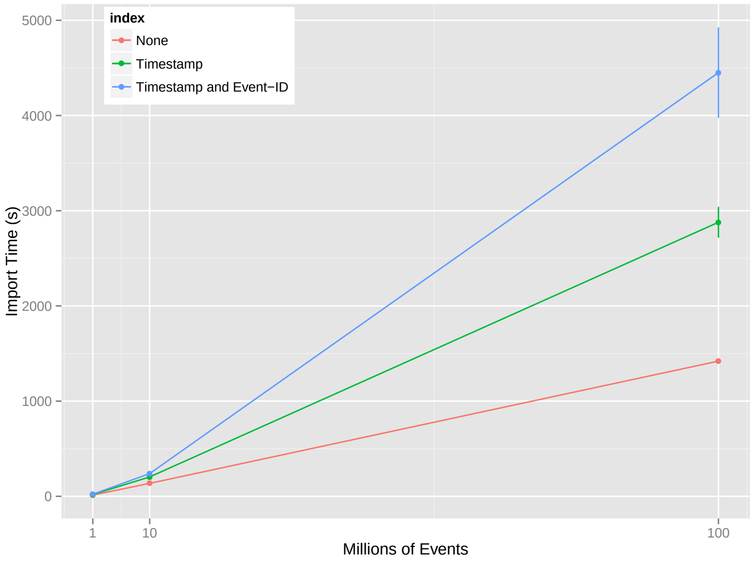
Figure 11 shows the import times for small, medium and big traces, obtained using all three indexing configurations. Figure 12 is only a detail of the previous one, showing only the results for small and medium traces. From Figure 11, we have a global overview of trace import time.

We observe that, without indexes, the import time grows linearly with the trace size. Adding indexes, the import time growth becomes more than linear: the more the indexes, the bigger the non linearity.

The increase factors in import times due to indexing on timestamp or event ID are summarized in Table 2. These factors are computed with respect to the case where no index is created.

We can summarize the whole analysis saying that for small and medium traces, indexing increases import times between 40% and 80%. The actual import times are by the way always relatively small (less than 4.5 minutes for the medium trace with both indexes). On the other

[8] Temictli is available on `https://github.com/soctrace-inria/framesoc.various`.



;

FIGURE 11 – Import times for small, medium and big traces, using different indexing configurations.

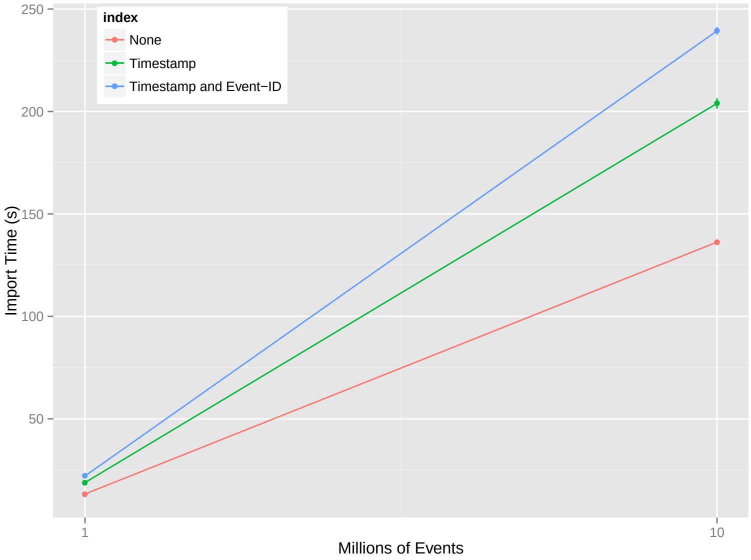


FIGURE 12 – Import times for small and medium traces, using different indexing configurations.

hand, import times for big traces are quite big even without indexing (about 24 minutes), and double or triple when adding one or two more indexes.

Trace	Timestamp Index	Timestamp and Event-ID Index
Small	1.4×	1.6×
Medium	1.5×	1.8×
Big	2.0×	3.1×

Table 2 – Import time increase factor using different indexing configurations.

4.2 Gantt Chart Loading

Description

Gantt Chart display is done in Framesoc by pipelining data loading and screen drawing. The data loading part is thus implemented with several queries for events in subsequent time intervals.

The following experiments show the advantages of pipelined loading, as opposite to *all-or-nothing* loading[9], measuring the impact of timestamp indexing on load time. Pipelined loading type will be referred as `pipelined` while the *all-or-nothing* loading type will be referred as `all`. The screen drawing part of Gantt display is not taken into account: only the loading part times are measured. The detailed experimental configuration is the following:

- Factors: trace size, indexing configuration, loading type.
- Levels:
  - trace size: small, medium, big (not complete tests, see after).
  - indexing: no index, timestamp index.
  - loading type: all, pipelined.
- Metric: import time, time to get the first interval.
- Number of repetitions:
  - small traces: 30.
  - medium traces: 10.
  - big traces: 5 repetitions (1000 intervals per repetition) for the `pipelined` loading type, to get a good estimation of the time to read an interval; no repetition for the `all` loading type, because of reached memory limits.

Analysis

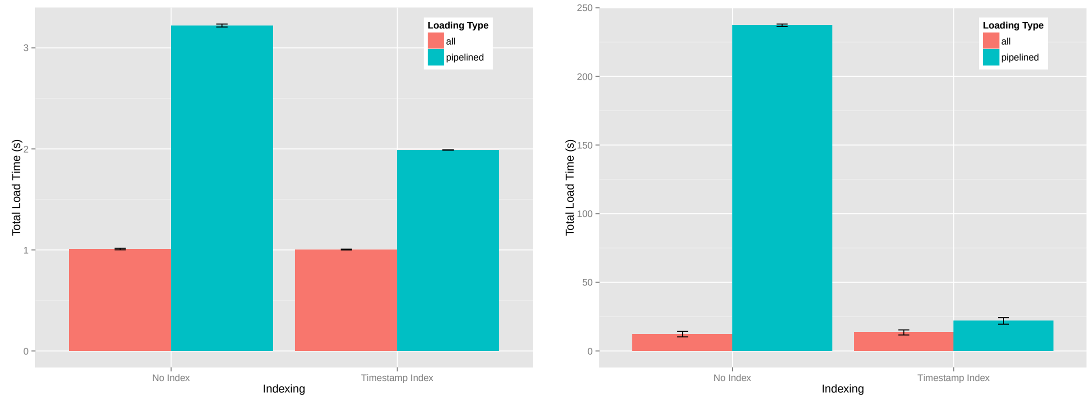
Figure 13 shows the total Gantt Chart load time for small and medium traces, using `all` and `pipelined` loading types, and using or not the timestamp index.

We observe that, using a pipelined loading type adds a significant overhead if we do not use timestamp indexing. This was expected since pipelined loading relies on several queries over different time intervals. Adding the timestamp indexing, a small overhead is still present, but it becomes smaller and smaller as trace size increases (the total load time for medium traces increases by a factor 1.6).

Paying this small overhead has two major benefits. The first one can be appreciated in Figure 14, which shows the time the user has to wait before starting to see meaningful information on the screen. Here we can see that pipelined loading type provides a better user experience, since the user starts seeing meaningful information almost immediately (the time to load an interval). On the contrary, with the `all` loading type, the user has to wait the loading of the

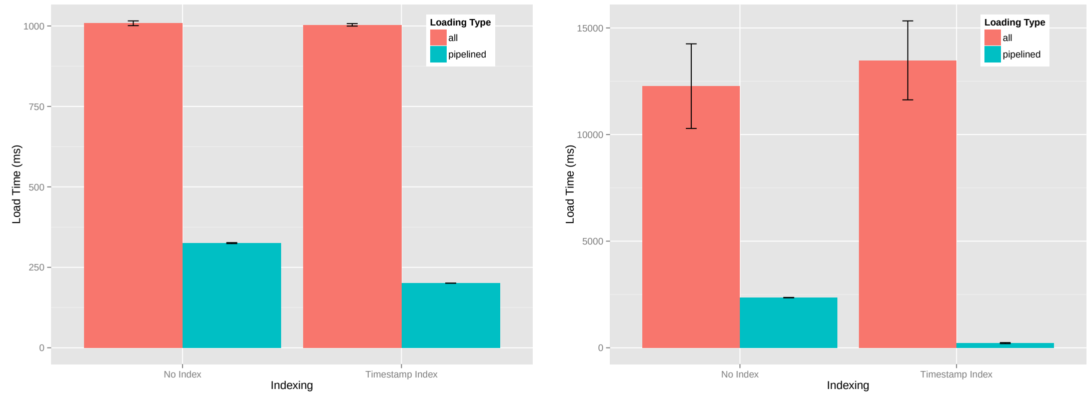
[9] With the expression *all-or-nothing* loading, we mean a solution where either the whole trace is loaded or nothing can be displayed.

whole trace, before seeing anything. The possibility to immediately start the analysis, basically erases the overhead paid for pipelined loading and visualization.



(a) Small Trace (b) Medium Trace

FIGURE 13 – Total Gantt Chart load time for small and medium traces, using different loading types, and using or not timestamp indexing.



(a) Small Trace (b) Medium Trace

FIGURE 14 – Gantt Chart load time needed to get the first meaningful information for small and medium traces, using different loading types, and using or not timestamp indexing.

The second benefit is that pipelined loading is a more scalable solution, compared to *all-or-nothing* loading. This is clear from Table 3, which shows interval and total load time for big traces. First of all, we observe that *all-or-nothing* loading could not be tested on big traces, since the memory limit (16 GB) was reached during tests. Furthermore, using timestamp indexing, the time to start seeing meaningful information (time to load an interval) is basically the same as small or medium traces. The whole trace is then loaded in about 4 minutes (without indexing, it would be 17 hours, i.e., about 250 times slower).

We conclude this analysis saying that pipelined loading, used in combination with timestamp indexing, provides the best user experience (high responsiveness) with small overhead. This kind

Indexing	**Pipelined Loading**		***All-Or-Nothing* Loading**
	Load interval	**Load whole trace**	**Load whole trace**
No Index	61 s	17 hours	memory limit reached
Timestamp Index	250 ms	4 min 10 s	memory limit reached

Table 3 – Gantt Chart loading time for big traces, using or not time index.

of loading is also the only one possible when dealing with big traces, since performing a single huge request to the database to get all the events would not scale.

4.3 Event Table Loading

Description

As for the Gantt Chart, the Event Table display is done in Framesoc by pipelining data loading and screen drawing. The data loading part is also implemented with several queries for events in subsequent time intervals. The Event Table use-case queries differ from the Gantt Chart ones, since here we have to load also all the event parameters. This peculiarity motivates the presence of an index on the event ID in the table of event parameters.

The following experiments show the advantages of pipelined loading, as opposed to *all-or-nothing* loading, measuring the impact of indexing on event timestamp and on event ID. Pipelined loading type will be referred as `pipelined` while the *all-or-nothing* loading type will be referred as `all`. The screen drawing part of Event Table display is not taken into account: only the loading part times are measured. The detailed experimental configuration is the following:

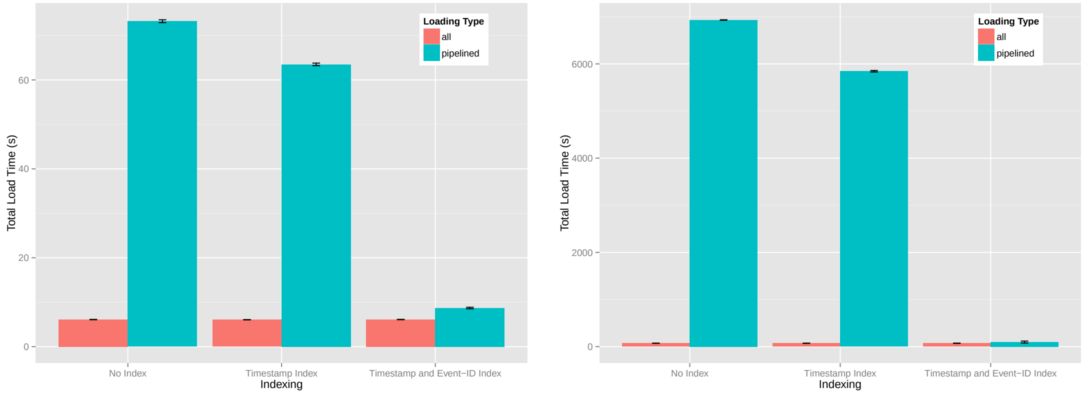
- Factors: trace size, indexing configuration, loading type.
- Levels:
  - trace size: small, medium, big (not complete tests, see after).
  - indexing: no index, timestamp index, timestamp and event ID index.
  - loading type: all, pipelined.
- Metric: import time, time to get the first interval.
- Number of repetitions:
  - small traces: 30.
  - medium traces: 10.
  - big traces: 1 repetition (5000 intervals) for the `pipelined` loading type, to get a good estimation of the time to read an interval; no repetition for the `all` loading type, because of reached memory limits.

Analysis

Figure 15 shows the total Event Table load time for small and medium traces, using `all` and `pipelined` loading types, using three different indexing configurations.

For the `all` loading type, changing the indexing configuration does not affect the performance, since we always load all the events and all the parameters, without any specific query condition. On the other hand, with the `pipelined` loading type we have huge gains in performance using both timestamp and event ID indexing. Using no index, the overhead compared to the `all` loading type is not acceptable. Using only the timestamp index, we gain a bit (load time reduced by about 15% for medium traces) but the overhead is still huge, since, if the query on the table of events is optimized, the query on the table of parameters is still very slow. Adding the index on the event ID in the table of parameters, we manage to optimize all the queries, thus reducing the load time drastically (we gain about 98% in time, compared with the configuration without indexes for medium traces).

Even with both indexes, a small overhead compared to the `all` loading type is still present. However, this overhead becomes smaller and smaller as trace size increases (the total load time for medium traces increases by a factor 1.3). Furthermore, as we have seen for the Gantt Chart use-case, paying this small overhead has two major benefits. The first one can be appreciated in Figure 16, which shows the time the user has to wait before starting to see meaningful information on the screen. Here we can see that pipelined loading type, combined with the use of timestamp and event ID indexes, provides a better user experience, since the user starts seeing meaningful information almost immediately (the time to load an interval). On the contrary, with the `all` loading type, the user has to wait the loading of the whole trace, before seeing anything. The possibility to immediately start the analysis, basically erases the small overhead paid for pipelining loading and visualization.



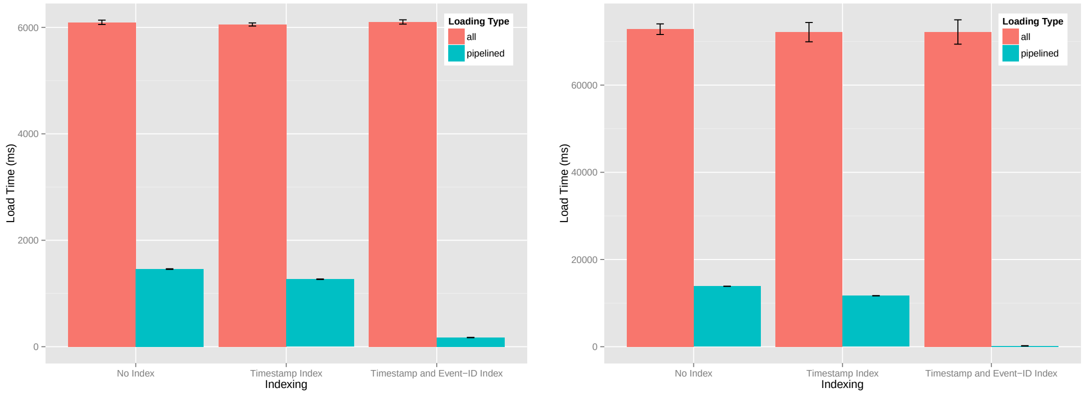
(a) Small Trace

(b) Medium Trace

FIGURE 15 – Total Event Table load time for small and medium traces, using different loading types and different indexing types.

Also for the Event Table use-case, the second benefit is scalability. Pipelined loading is a more scalable solution, compared to *all-or-nothing* loading. This is clear from Table 4, which shows interval and total load time for big traces. As it happened for the Gantt Chart, *all-or-nothing* loading could not be tested on big traces, since the memory limit (16 GB) was reached during tests. Furthermore, using timestamp and event ID indexing, the time to start seeing meaningful information (time to load an interval) is basically the same as small or medium traces. The whole trace is then loaded in about 25 minutes (without any indexing it would be 202 hours, i.e., about 485 times slower).

We conclude this analysis saying that pipelined loading, used in combination with timestamp and event ID indexing, provides the best user experience (high responsiveness) with small overhead. As for the Gantt Chart use-case, this kind of loading is also the only one possible when dealing with big traces. For the Event Table use-case, this point is even more critical, since also the event parameters are loaded: performing a single huge request to the database to get all the events and all their parameters would not scale.



(a) Small Trace

(b) Medium Trace

FIGURE 16 – Event Table load time needed to get the first meaningful information for small and medium traces, using different loading types and different indexing types.

Indexing	**Pipelined Loading**		***All-Or-Nothing* Loading**
	Load interval	**Load whole trace**	**Load whole trace**
No Index	146 s	202 hours	memory limit reached
Timestamp Index	118 s	163 hours	memory limit reached
Timestamp and Event-ID Index	300 ms	25 minutes	memory limit reached

TABLE 4 – Event Table loading time for big traces, using different indexing configurations.

5 Conclusions

In this technical report, we described the solutions proposed by Framesoc for an effective and efficient management of huge traces.

The data management has been built using a pipeline architecture between data loading and data visualization phases. This enables an interactive analysis on partial data and provides a good performance with low latency, especially when dealing with huge traces.

Framesoc ensures a scalable trace analysis through an interactive visualization workflow and data filtering. Indeed, the view synchronization over time intervals offers the possibility to link the different analysis views in an interactive workflow, where at each step we can reduce the scope of the analysis through filtering. To get an overview of the whole trace, we can use memory-cheap visualizations like the Event Density Chart, or more sophisticated aggregated view as the one provided by Ocelotl [6]. Then we can load only portions of the trace in more detailed and memory-expensive views, like the Gantt Chart or the Event Table.

Currently, all Framesoc analysis views are able to filter over the three dimensions of space, time and types. As for the near future, it would be interesting to provide the possibility to synchronize the different views not only over the time dimension, but also over the other two (space and types), thus obtaining a multi-dimensional view synchronization. Among our other perspectives, we plan to investigate new forms of storage, based on *non relational* distributed solutions. This could allow for a parallelization of data writing (import) and reading (analysis), thus increasing the global performance and improving scalability. Then, in the long term, we would like to enrich Framesoc in order to better support new use cases related to parallel and distributed application analysis. In particular, it would be interesting to have new analysis views supporting a scalable visualization of huge task-dependency graph.

References

[1] SoC-TRACE project. `http://tinyurl.com/minalogic-soc-trace`.

[2] Generoso Pagano, Damien Dosimont, Guillaume Huard, Vania Marangozova-Martin, and Jean-Marc Vincent. Trace Management and Analysis for Embedded Systems. In *Proceedings of the IEEE seventh International Symposium on Embedded Multicore SoCs (MCSoC-13)*, Tokyo, Japan, sep 2013.

[3] Generoso Pagano and Vania Marangonzova-Martin. SoC-Trace Infrastructure. Rapport Technique RT-0427, INRIA, November 2012.

[4] Generoso Pagano and Vania Marangozova-Martin. FrameSoC Workbench: Facilitating Trace Analysis through a Consistent User Interface. Technical Report RT-0447, Inria, April 2014.

[5] Framesoc website. `http://soctrace-inria.github.io/framesoc/`.

[6] Damien Dosimont, Robin Lamarche-Perrin, Lucas Mello Schnorr, Guillaume Huard, and Jean-Marc Vincent. A Spatiotemporal Data Aggregation Technique for Performance Analysis of Large-scale Execution Traces. In *IEEE Cluster 2014*, Madrid, Spain, September 2014.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-0803