HAL
open science

# SoC-Trace Infrastructure Benchmark

Generoso Pagano, Vania Marangozova-Martin

## To cite this version:

Generoso Pagano, Vania Marangozova-Martin. SoC-Trace Infrastructure Benchmark. [Technical Report] RT-0435, INRIA. 2013, pp.25. hal-00830008

**HAL Id: hal-00830008**

**https://inria.hal.science/hal-00830008v1**

Submitted on 4 Jun 2013

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# SoC-Trace Infrastructure Benchmark

Generoso Pagano, Vania Marangozova-Martin

TECHNICAL REPORT
N° 435
June 2013
Project-Team MESCAL

ISSN 0249-0803 ISRN INRIA/RT--435--FR+ENG

# SoC-Trace Infrastructure Benchmark

Generoso Pagano *, Vania Marangozova-Martin †

Équipe-Projet MESCAL

Rapport technique n° 435 — June 2013 — 25 pages

**Résumé :** Ce document présente la comparaison de performances que nous avons effectuée entre deux implémentations de l'Infrastructure SoC-Trace. La première implémentation, basée sur l'utilisation de bases de données distribuées, est décrite dans le rapport technique RT-427 [2]. La deuxième implémentation a été réalisée dans la suite et fait usage d'une base de données centralisée. Ce document décrit également les dernières évolutions de l'Infrastructure SoC-Trace.

**Mots-clés :** Traces d'exécution, gestion de traces, infrastructure, modèle de données, base de données, format de trace, banc d'essai, évaluation de performances.

This research is supported by FUI [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

# SoC-Trace Infrastructure Benchmark

**Abstract:** This document presents the performance benchmark performed on two different implementations of the SoC-Trace Infrastructure. The first implementation, based on a distributed database, is described in the technical report RT-427 [2]. The second implementation has been realized later in order to explore the alternative centralized approach. This document describes also the latest evolutions of the SoC-Trace Infrastructure.

**Key-words:** Execution traces, trace management, infrastructure, data-model, database, trace formats, benchmark, performance evaluation.

# Table of contents

**1 Introduction** **4**

**2 SoC-Trace Infrastructure Centralized Implementation** **4**
2.1 Multi-Trace DB . . . . . 4
2.1.1 Conceptual Model . . . . . 4
2.1.2 Database Architecture . . . . . 5
2.1.3 Database Schema : Replicated versus Catalog Solution . . . . . 6
2.2 SoC-Trace Library . . . . . 8
2.2.1 Model . . . . . 8
2.2.2 Storage . . . . . 8
2.2.3 Query . . . . . 9
2.2.4 Search . . . . . 10

**3 Benchmark Plan** **10**
3.1 Benchmark Goals . . . . . 10
3.2 Infrastructure Services . . . . . 10
3.3 Benchmark Metrics . . . . . 11
3.4 Parameters . . . . . 11
3.5 Factors and Levels . . . . . 12
3.6 Workloads . . . . . 12

**4 Benchmark Experiments** **12**
4.1 Database Storage . . . . . 13
4.2 Trace Import . . . . . 14
4.3 Search and Save . . . . . 15

**5 Conclusions** **18**

**A Database performance** **19**

**B SoC-Trace Infrastructure Current State** **22**
B.1 Database Management . . . . . 22
B.2 Tool Management . . . . . 23
B.3 Graphical User Interface . . . . . 23

# 1 Introduction

One of the main objectives of SoC-Trace project [2] is the development of a trace management infrastructure, storing traces and facilitating the access to trace data for analysis tools. The first implementation of SoC-Trace Infrastructure deals with the storage problem using a distributed database solution, where each trace has its own database for storing events and analysis results, while a central database is used to store trace metadata and organize trace databases. In order to avoid the logical replication of this distributed solution (each trace database has the same schema) and the creation of a new database for each new trace, a centralized design seemed to be an interesting option. We therefore designed a centralized version of the database and implemented the corresponding version of the SoC-Trace Infrastructure software library, in order to compare the two different approaches.

This report is organized as follows. Section 2 presents the centralized implementation of the SoC-Trace Infrastructure (referred as CDB[1]). Section 3 presents the defined performance parameters and the benchmark plan. Section 4 gives the corresponding experimental results. Section 5 presents the conclusion of this benchmark. Two appendices are provided at the end of this document : Appendix A is a study of DB performance and Appendix B briefly presents the latest developments of SoC-Trace Infrastructure.

Note that the description presented in Section 2 is done by highlighting the differences between CDB and the distributed database implementation (referred as DDB[2]). It is therefore strongly recommended to read the technical report describing DDB [2] in order to have a good understanding of the present document.

# 2 SoC-Trace Infrastructure Centralized Implementation

From a high level perspective, the greatest difference between CDB and DDB is in the way multiple traces are managed at the database level. In the following subsections we therefore first present the database differences before discussing the related repercussions at the software interface level.

## 2.1 Multi-Trace DB

In both versions, the MySQL DBMS is used to store trace general information, raw trace data and analysis result. The following subsections describe the differences among the two considered approaches.

### 2.1.1 Conceptual Model

Figure 1 shows the data model of CDB. The only difference compared to DDB is that the relation between `TRACE` and `ANALYSIS_RESULT` entities is many-to-many, instead of one-to-many. Indeed, at the moment CDB allows the storage of analysis results related to more than one trace (multi-trace analysis). This possibility is an ongoing work for DDB. In any case, this difference is not relevant to the benchmark tests presented in this report, as only common features are compared.

[1] CDB for Centralized DataBase implementation
[2] DDB for Distributed DataBase implementation

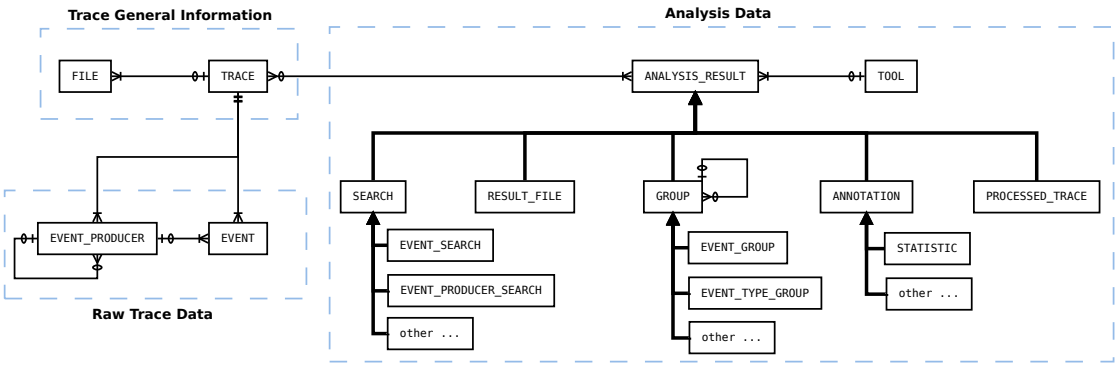


FIG. 1 – CDB Conceptual Model

### 2.1.2 Database Architecture

The principal difference between DDB and CDB is in the database architecture, as shown in Figure 2 and Figure 3. In DDB, the Multi-Trace DB is designed as a two-level hierarchy of databases. The root of the hierarchy is the System DB, which contains general information about traces and the tool registry. The leafs are the Trace DBs, each containing the information related to a single trace (raw events, event producers, file references and analysis results for that trace).

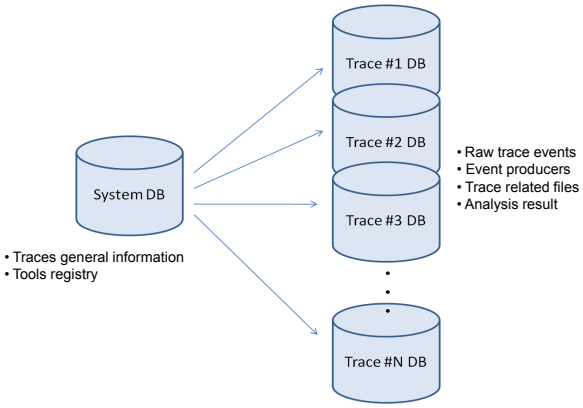


FIG. 2 – DDB Architecture

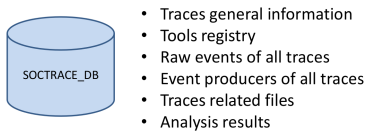


FIG. 3 – CDB Architecture

The DDB solution has the following issues :

- A new Trace DB needs to be created for each new managed trace.
- There is a model replication, since all Trace DBs have the same database schema (Figure 5).
- Trace DBs always store the trace format, even if the system already contains other traces with the same format : we have therefore some data redundancy.

These observations motivated the reflection about the CDB solution (Figure 3). However, a naive approach in designing a centralized database, such as putting all events of all traces are in the same table, would lead to the following issues :

- The space overhead to store a trace identifier in all rows of almost all tables (especially the ones containing trace event data).
- The time overhead to perform queries in presence of the above mentioned trace identifier.
- The presence of huge tables, which can lead to the necessity of table partitioning [3].

Therefore, we designed CDB with the aim of solving DDB issues, without introducing the problems just mentioned and trying to keep the complexity of the DB schema low. All the details of CDB schema are given below.

### 2.1.3 Database Schema : Replicated versus Catalog Solution

As specified in [2], in DDB System DB and Trace DB have the schemas shown in Figure 4 and Figure 5 respectively.

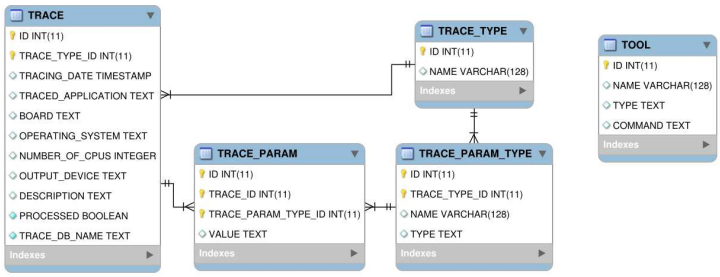


FIG. 4 – DDB System DB schema

To get the CDB schema, we modified the DDB schemas in order to :

- merge the information of the two schemas in a single one, thus minimizing replication at the logical level ;
- replicate only a subset of tables for each trace, adopting a *catalog* solution, as described in the following.

In the catalog solution we do not replicate all the database tables for each trace. We only replicate the tables containing trace-specific raw information, namely `EVENT`, `EVENT_PARAM` and `EVENT_PRODUCER`. For each trace, these tables have a different name, having as a suffix the trace identifier (e.g. `EVENT_1`). The `TRACE` table, with its list of trace IDs, becomes the catalog identifying the set of three tables related to a single trace. With this approach, when a new trace is imported into the infrastructure, only these three tables are created and not a whole database. Thus it is no longer necessary to rewrite the trace format description (`EVENT_TYPE` and `EVENT_PARAM_TYPE` table information) for each trace of a given type : it is enough to write the format once, then all the traces of that format refer to it.

Figure 6, shows an example where three different traces have been imported to the system. There are three instances of the `EVENT`, `EVENT_PARAM` and `EVENT_PRODUCER` tables, one for each trace.

Let us look at the whole CDB schema given in Figure 7. For the sake of clarity, we have represented a single instance per replicated table.

We can highlight the following differences between DDB and CDB :

- The `TRACE_DB_NAME` field has been removed from the `TRACE` table, since in CDB there a single DB for all traces.
- The `EVENT_TYPE` table contains the additional field `TRACE_TYPE_ID`, since in CDB all the event types of all trace formats are stored in this table.

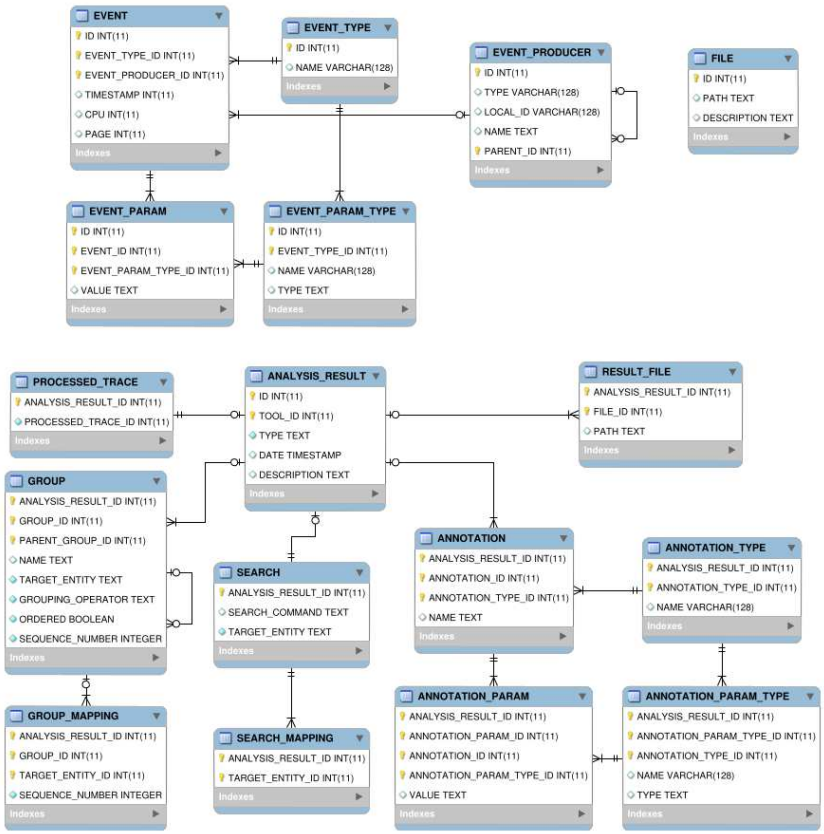


FIG. 5 – DDB Trace DB schema

- The `FILE` table contains the additional field `TRACE_ID`, since in CDB file information related to all the traces are stored in this table.
- There is an additional `TRACE_ANALYSIS_MAPPING` table, necessary to implement the many-to-many relationship between TRACE and `ANALYSIS_RESULT` entity of the data-model. At the logical level this relationship is decomposed in two different relationships : a one-to-many and a many-to-one relationship.
- The `PROCESSED_TRACE` table contains the field `SOURCE_ID`, since the link with the source trace is no longer implicit.

It is important to point out that in CDB, all the analysis results of all traces are stored in the same tables, since the tables related to concrete analysis result entities are not replicated. For example, the file results related to trace `X` and trace `Y` are saved in the same `RESULT_FILE table`. However, some of the predefined analysis results are currently considered only as *single-trace* results (grouping, searching and processed trace results). On the other hand, annotations and file results can be both *single-trace* and *multi-trace* results. This constraint is enforced by the SoC-Trace Library in CDB.

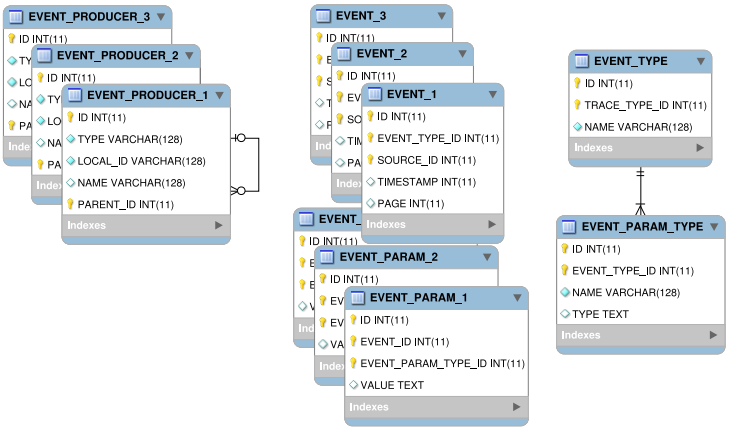


FIG. 6 – CDB Catalog Solution

## 2.2 SoC-Trace Library

CDB implementation of the SoC-Trace Library takes into account the DB changes, while keeping almost the same interface for the user. The following sections briefly describe the main differences compared to DDB.

### 2.2.1 Model

In the CDB `com.inria.soctrace.lib.model` plugin we find only minor changes, reflecting the changes at the DB schema level. For example, a `TraceType` reference has been added to the `EventType` class and a `traceId` field has been added to the `File` class. Also, the `AnalysisResult` class contains a list of `Trace` objects, in order to support the possibility of having multi-trace analysis results.

### 2.2.2 Storage

The `com.inria.soctrace.lib.storage` plugin is the one where we can find the most important differences. Given that in CDB we deal with a single database, the `SystemDBObject` and `TraceDBObject` classes have been replaced by a single `DBObject` class. This point slightly impacts the user interface : instead of creating a `SystemDBObject` instance and then have different `TraceDBObject` instances, the user works with a single instance of the `DBObject`. The list of services offered by this `DBObject` is basically a union of the services of DDB `SystemDBObject` and `TraceDBObject` objects.

At the implementation level, there are several other changes, but with no impact on the user interface. The major ones are :

- The `SQLConstant` class has been updated with constants reflecting the new DB schema.
- The `Visitor` class has been enriched with a map that links a trace ID with an instance of the `SingleTraceStatements` class. This inner class is used to manage the prepared statements related to the three trace-specific tables (`EVENT`, `EVENT_PARAM`, `EVENT_PRODUCER`). It also manages the naming convention for these three tables (trace ID as suffix).
- The format cache mechanism has been updated in order to allow the `DBObject` to manage several trace formats at the same time. Basically, the `DBObject` contains a `TraceFormatsCache`

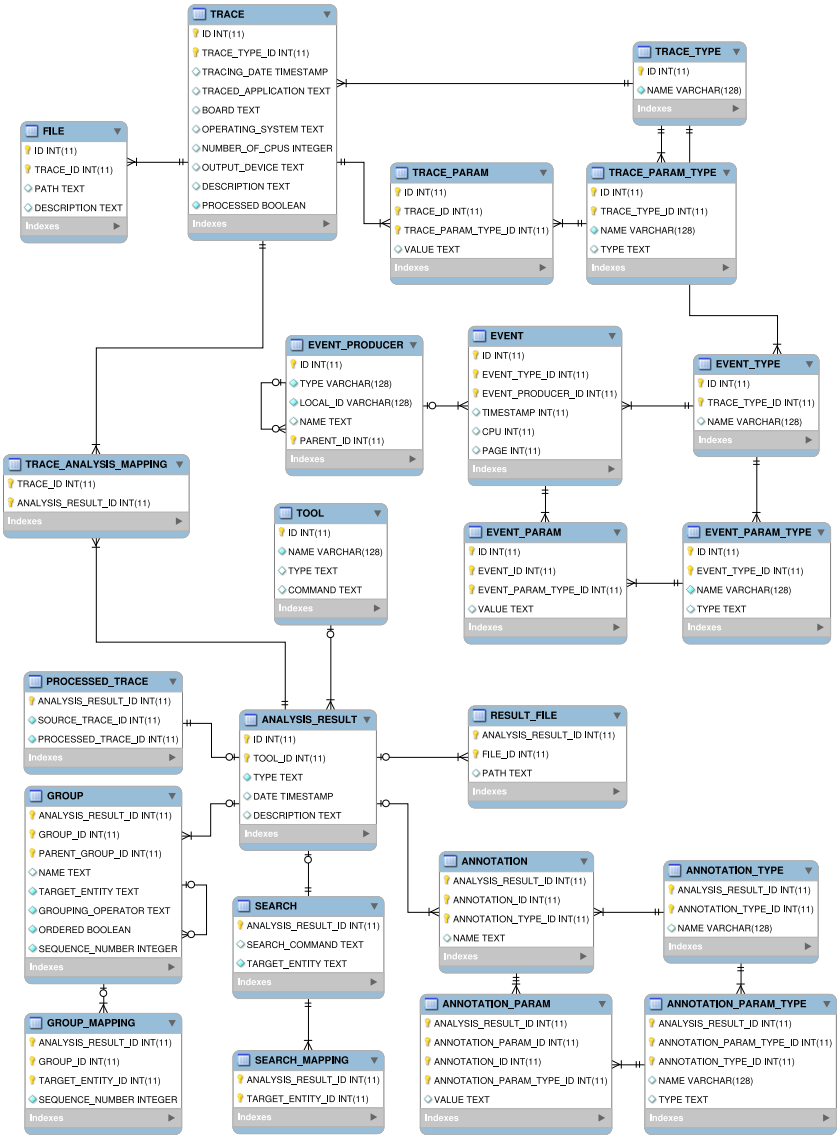


FIG. 7 – CDB Database Schema

object, which contains the `TraceType` and `TraceParamType` objects and a map of `TraceFormatCache` objects.

### 2.2.3 Query

The query classes provided by the `com.inria.soctrace.lib.query` plugin are only marginally affected by the changes in the DB architecture in CDB. The only real difference with DDB is in the instantiation of these classes. In DDB, the constructor of a query class typically takes either the `SystemDBObject` or a `TraceDBObject` instance, depending on where the entity being the object of the query is stored. For CDB there is a single DB, so the constructor of a query

class always uses a `DBObject` instance.

We note that for both `EventQuery` and `EventProducerQuery` classes, a `Trace` object is passed to the constructor, in order to specify the trace we want to work with. We also note that for `EventTypeQuery` class, the constructor needs a `TraceType` object in addition to the `DBObject`, since in CDB all the `EventType` of all formats are stored in the same table.

Apart from these initialization differences, the logic of dealing with query and condition classes is the same for CDB and DDB.

#### 2.2.4 Search

As explained in [2], the search interface provided by the `com.inria.soctrace.lib.search` plugin can be used in a database-agnostic fashion. Therefore the changes in the DB architecture do not affect the provided user interface.

On the other hand, the change in the data-model regarding the possibility of having multi-trace analysis results does have a consequence on the user interface. Indeed, in DDB, to retrieve analysis results, it is always necessary to specify the `Trace` the results are related with. In CDB, this is no more the case, since an analysis result is not necessarily related to a single trace. This tiny difference is only temporary, since it will disappear when also DDB will support multi-trace results.

## 3 Benchmark Plan

In order to design the benchmark for comparing the performances of the two different implementations of SoC-Trace Infrastructure, we followed the guidelines described in [4]. The vocabulary is also taken from the cited book.

We summarize below the basic concepts we based our benchmark on.

- Parameter : something influencing the behavior of the system.
- Factor : parameter you decide to variate during experiment.
- Level : value given to a factor.
- Workload : a set of service requests to the system (e.g. a program executing queries).

We performed a benchmark based on actual measurements of the real system implementations, namely DDB and CDB.

### 3.1 Benchmark Goals

Our goal is to compare the DDB and CDB implementations in order to chose the one with better performances for the SoC-Trace project.

The architecture of the considered system is the one presented in Figure 8. The system is composed of a DBMS and a software library to access the DB. As the SoC-Trace Library is implemented in Java, the interaction with the MySQL DBMS is performed through its JDBC driver.

In this layered architecture, only the library implementation layer changes between DDB and CDB.

### 3.2 Infrastructure Services

The main services provided by the system under test are the following :

- Import a trace into the system
- Perform searches on traces

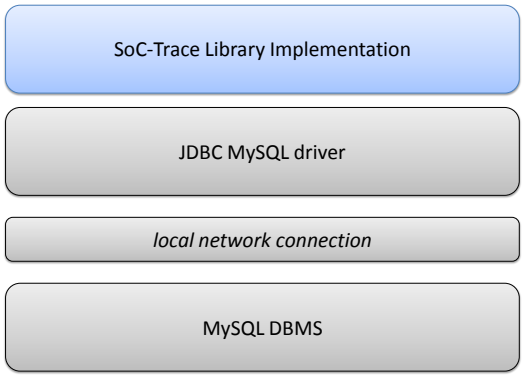


FIG. 8 – System Layers

  - Search for traces respecting some criteria
  - Search in a trace for events or event producers respecting some criteria
- Save and retrieve analysis results

## 3.3 Benchmark Metrics

The metrics chosen for our experiments are the following :

- Speed of execution : the time needed to perform a given operation.
- Disk space usage : the size of the database on the disk.
- Memory usage : the maximum size of the used Java Virtual Machine heap during the execution of a given operation.

Depending on the specific experiment, the principal metrics will be pointed out and analyzed.

## 3.4 Parameters

We make the distinction between the system parameters which are related to the hardware and software configuration of the system, and workload parameters which characterize users' requests. The main system and workload parameters identified for this study are reported below.

The system parameters are :

- Type and speed of the disk
- Processor
- RAM
- OS
- Kernel
- MySQL version and engine
- MySQL JDBC driver version
- SoC-Trace Library type

The workload parameters are :

- Number of managed trace formats
- Number of managed traces
- Size of a trace
- Request type
- Number of saved results

- Size of saved result

All these parameters, if varied, may alter the behavior of the system in performing given operations. Among these parameters, we chose to vary only a subset, thus identifying benchmark factors.

### 3.5 Factors and Levels

The only system parameter varied during the experiments is the SoC-Trace Library type : DDB or CDB. All the other system parameters are fixed as follows :

- Type and speed of the disk
  - SSD disk model MTFDDAK256MAM-1K1
  - device size : 256 GB
  - Measured timing buffered disk reads (avg) : 461.86 MB/s
  - Measured timing write (avg) : 92.1 MB/s
- Processor : Intel® Xeon(R) CPU E5-1660 0 @ 3.30 GHz x 12
- RAM : 16 GB DDR3
- OS : Fedora Release 17 (BeefyMiracle) 64-bit
- Kernel : Kernel Linux 3.6.2-4.fc17.x86_64
- MySQL : server version 5.5.28, InnoDB engine
- MySQL driver version : 5.1.22

Regarding workload parameters, only a subset of them has been chosen as factors.

- The number of different formats managed : 1, 5, 30 (depending on the experiment)
- The number of traces managed : 1, 5, 30 (depending on the experiment)
- The size of the trace : small (1 MB), large (100 MB) (all experiments)

Note that all the different formats are actually *fictitious*, meaning that the same format (KPTrace) is stored several times with different names. From the SoC-Trace Infrastructure point of view, the formats are different, so stored separately. Furthermore, in our experiments all the traces having the same size are actually identical, in order to ensure that exactly the same events and event producers are managed by the system. Note also that the experiments described below are designed so that there are dependencies between different parameters : this way we can keep the number of factors small. In fact, as it will be clearer later, in Section 4.3 the number of results is a function of the number of managed traces and the size of the results is a function of the size of the trace.

### 3.6 Workloads

The workloads used to carry out our experiments are generated with the KPTrace parser tool and an *ad hoc* test application. The KPTrace parser tool is used to import traces into the system. As anticipated above, different trace type names are used to produce *fictitious* different formats. The *ad hoc* application is a simple program that performs search queries and saves the results. More details about this last program are given in Section 4.3.

## 4 Benchmark Experiments

In this section we present the experiments we designed and implemented, discussing the results obtained.

## 4.1 Database Storage

In this experiment we import traces into the system using the KPTrace parser tool in order to gather information about disk space usage.

The factors considered in this experiment are the trace size and the number of traces. The number of formats is not relevant, so no explicit scenario will be described on this topic and only analytic considerations will be done. In fact, regarding the number of trace formats, the greatest expected gain of CDB over DDB is when only a single format is considered, since CDB does not replicate format storage : all the scenarios will therefore deal with a single format.

We consider two scenarios, where we import into the system 5 traces with the same format. In the first scenario we import small traces (1 MB) while in the second we import large traces (100 MB). In each scenario, after importing each trace, we measure the whole disk space used for the DB storage. In the CDB case, we consider the single DB size. In the DDB case, we take the sum of the sizes of the System DB and the various Trace DBs.

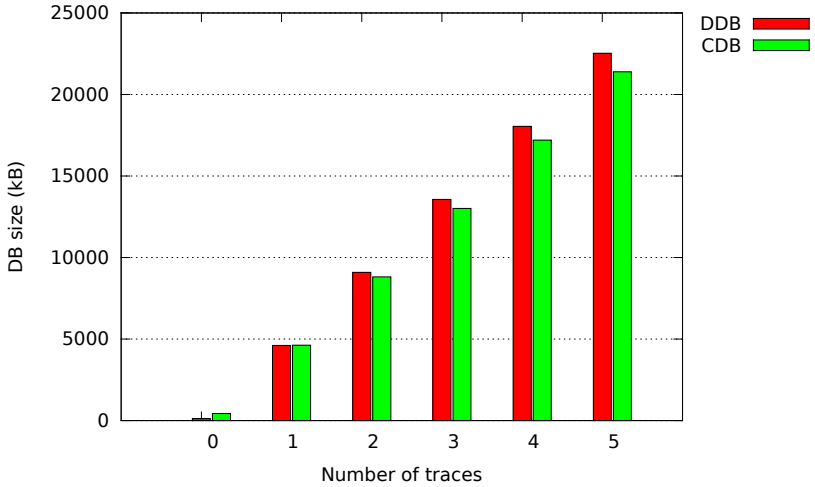


FIG. 9 – Storage size for small traces having the same format (lower is better)

Figure 9 shows the result for small traces. When in the system there is more than 1 trace, the CDB solution produces a slightly smaller disk occupation (space gain ranges from 3% to 5%). The difference is due to the presence of empty tables in each DDB Trace DB prepared for analysis results [3].

This result is conservative (though to a lesser extent) even when analysis result related tables are not empty, since CDB uses the same set of tables for all traces, while each DDB Trace DB has its own set of tables. A similar argument applies to the case of traces having different formats : if exactly the same set of formats is used in both DDB and CDB case, CDB size on disk will be always smaller or equal, since all the formats are stored in the same couple of `EVENT_TYPE` / `EVENT_PARAM_TYPE` tables.

Figure 10 shows the result for large traces. The same logic applies here but the gain obtained by CDB is so small that is completely negligible (the space gain here ranges from 0.04% to 0.07%).

In our experiments, we actually observed that if exactly the same number of traces of the same formats is imported in the system, the space gain of CDB over DDB is not dependent on

[3] The MySQL InnoDB engine uses 16 kB of disk space for empty tables

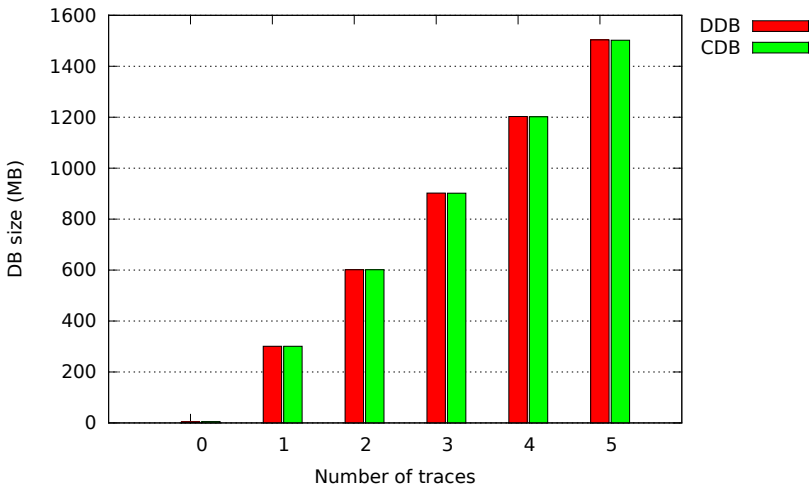


FIG. 10 – DB size for large traces having the same format (lower is better)

the size of the traces, but only on their number. This gain is less than 300 kB for each imported trace. It is for this reason that, especially in the case of large traces, the gain is negligible.

## 4.2 Trace Import

In this experiment we measure the time needed to import a trace into the system using the KPTrace parser tool. We measure the average time needed to import a KPTrace trace in DDB and CDB, varying the trace size and the number of managed formats. For this purpose, we have explored two groups of scenarios : in the first we consider small traces (1 MB) and in the second large traces (100 MB). For each group, we consider the cases described below.

- CDB :
  - Import 1 trace of a given format, being the format already stored in the DB (30 repetitions).
  - Import 30 traces of 30 different formats, being these formats not stored yet in the DB. This case can be seen as "importing 1 trace whose format is still not stored in the DB, with 30 repetitions".
- DDB :
  - Import 1 trace of a given format (30 repetitions).
  - The import of 30 traces of 30 different formats is not necessary for DDB, since even in the above case (single format) at each repetition the format is saved anyway in the specific Trace DB. The result obtained would be exactly the same.

The idea of this experiment is to evaluate if and how much we gain in CDB avoiding the creation of a Trace DB for each trace import and, in the single format case, avoiding the replication of the format done in DDB.

Figure 11 shows the result obtained for small traces. Considering only CDB (first two bars) we observe that managing different formats increases the import time of only 0.7%. This confirm that the format replication is not really a performance issue in DDB. Considering DDB results (third bar) we distinguish between the time needed for the actual trace import and the time needed to create, in the first place, the Trace DB. We observe that the difference in time between CDB results and DDB ones is actually due to the time spent in creating the DB. This time is

less than 700 ms and is independent from the trace size. We can summarize our observations saying that for small traces the maximum gain of CDB over DDB is obtained dealing with a single format and is about 12.6% in time.

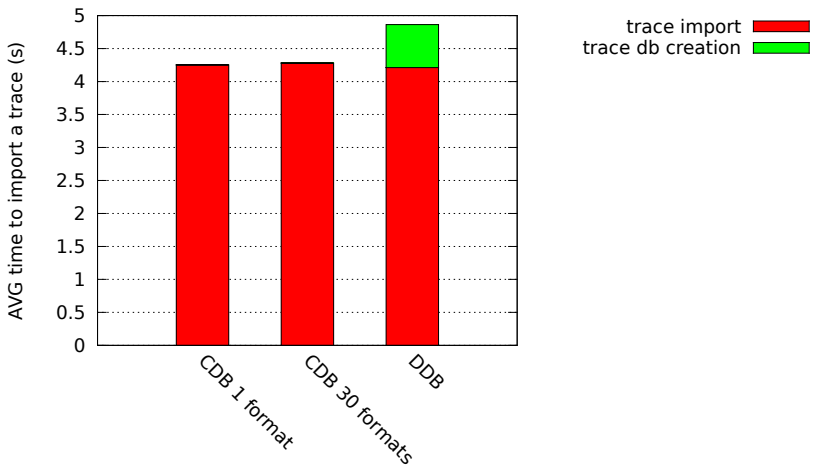


FIG. 11 – Import of Small Traces

These considerations are basically confirmed in Figure 12, which shows the results obtained with large traces. Considering CDB we observe that managing several formats increases import times only by 1.3%. Considering DDB, we confirm that the Trace DB creation time does not change comparing with previous scenario and stays around 700 ms. Dealing with large traces the weight of this operation is therefore clearly negligible : the figure shows only the last portion of the y axis (from 200 s) and it is still hard to appreciate the green part of the third bar. We can conclude that for large traces the maximum gain of CDB over DDB is obtained dealing with a single format and is about 2.1% in time, so negligible.

## 4.3 Search and Save

In this experiment we measure the execution time and some memory metrics for an *ad hoc* application performing searches and saving results, working with multiple traces. Namely, the application searches for the events of a given type within all the traces satisfying some given criteria, and saves the corresponding results.

The application is multithreaded with one thread per search. Note that multithreading is only at the application level, while SoC-Trace Library is sequential. The only portion of the library (both for CDB and DDB) being concurrency aware is the one dealing with DB write operations and transactions : the storage part of the library ensures that save operations as well as commits on a given connection are done in mutual exclusion. Considering DB read operations, when using a single connection concurrent queries on that connection are sequentialized by MySQL ; on the contrary, with different connections, different MySQL threads serve concurrent requests. In both cases concurrency is managed at DBMS level. In this experiment the single connection case is significant only for CDB, since DDB has different DBs for different traces, so there are always different connections.

The application is run for both DDB and CDB in two distinct scenarios : one for small traces (1 MB) and one for big traces (100 MB). For each scenario we have two different configurations

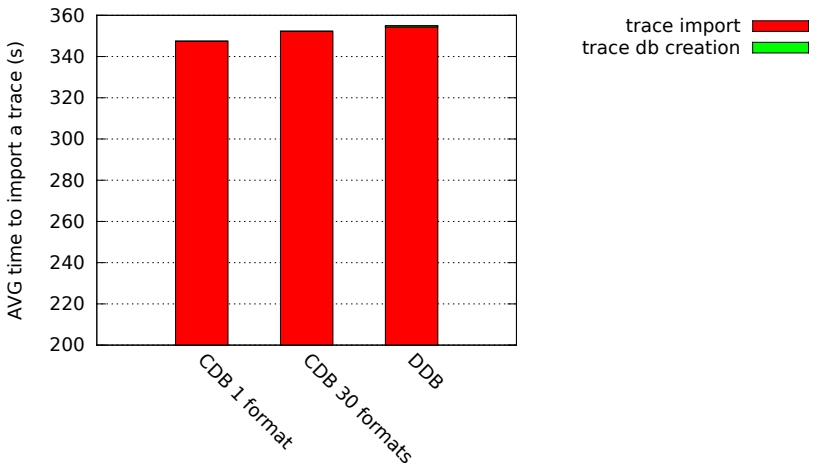


FIG. 12 – Import of Large Traces

considering respectively 5 traces having 5 different formats and 5 traces having the same format.

For all configurations, we have used trace selection criteria that result in returning all the 5 traces. As the traces are actually identical, the event searches return the same set of events.

Considering CDB, actually we tested all the configurations both using a single DB connection for all threads and opening a distinct DB connection for each thread. Strictly speaking, we are introducing here a new factor, the number of DB connections, for CDB. As explained before, for DDB there is always a distinct DB connection for each thread, since each thread works on a different trace, and each trace has its distinct DB. For each configuration we performed 60 runs.

In the figures presenting the results we will use the following naming conventions :

- `sf/mf` to distinguish the case with a single format and the case with 5 different formats.
- `sc/mc` to distinguish in CDB tests the case with a single DB connection and the case with one connection per thread.

In Figure 13 we have the aggregated results for small traces : for each configuration, the red bar is the average time while the green bar is the maximum amount of memory used. The two metrics are presented together in order to have a better overall vision of the system behavior.

For CDB, when we use a single DB connection shared by all threads (first two configurations), the average time spent to run the application is about the double comparing with all the other configurations. This is explained by the fact that MySQL manages each connection with a single thread, so the requests coming from the application threads are sequentialized by the DBMS.

Still considering the CDB single-connection configurations, when dealing with several formats the time is 7.3% higher and the memory used is 5.3% higher because of the overhead of managing more format data. Coming to the CDB multi-connection configurations (third and fourth bars), we can see that from the time point of view we gain a lot (about 43%) since we removed the sequentialization, but the memory used is more (about 5%) since we have to manage different connections with the DB. Anyway, CDB results are always worse than DDB ones, regarding both memory and time : even in the multi-connection configuration, CDB is slower (7.6% and 13% slower for single and multiple format respectively) and it takes more memory (5.7% and 10.6% more memory for single and multiple formats respectively).

These considerations are confirmed by the study on large traces (Figure 14). In the CDB single-connection configurations the application is up to 158% slower than in DDB cases, even if

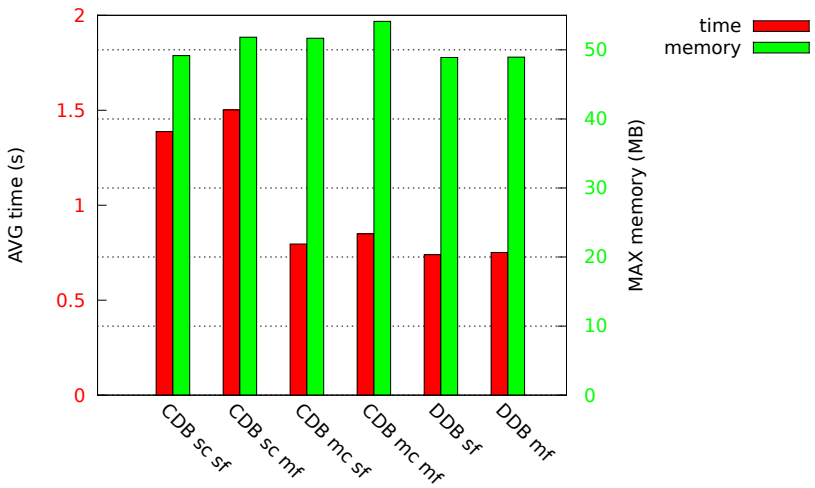


FIG. 13 – Average time and maximum amount of memory used to run the *ad hoc* application for small traces (lower is better)

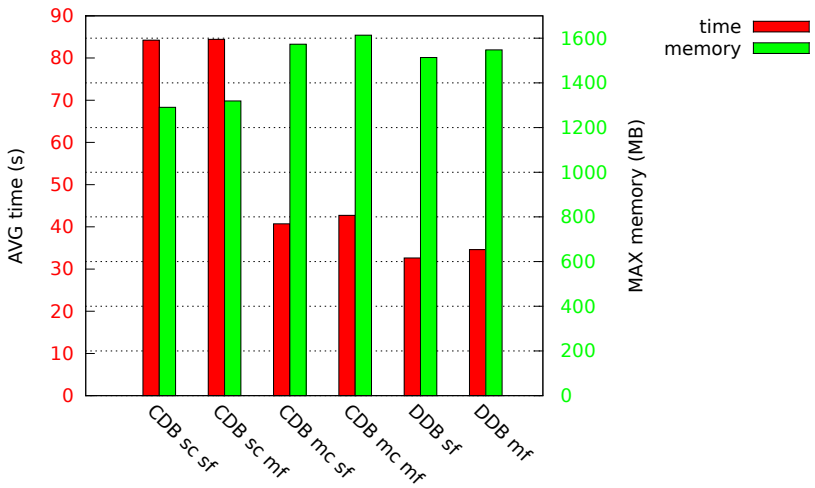


FIG. 14 – Average time and maximum amount of memory used to run the *ad hoc* application for large traces (lower is better)

it takes less memory (about 15% less). Considering then the CDB multi-connection configurations both time and memory are still larger than DDB results : for the single format configuration time is 25% longer and memory is 4% higher, for the multi format configuration time is 23.5% longer and memory is 4.2% higher.

We conclude from the analysis of this experiment that that DDB outperforms CDB in all configurations on almost all metrics. The CDB single-connection configuration is always a lot slower than DDB. Considering the CDB multi-connection configuration, it is less slower, but it takes more memory. Considering the average time metric, this kind of behavior is significantly more visible when dealing with larger traces.

# 5 Conclusions

In this report we have compared two alternative implementations of the trace management infrastructure developed for the SoC-Trace project. The first one, referred as DDB, features distributed databases, having a distinct database for each trace and a central system database. The second one, referred as CDB, has a centralized implementation of the database, where all traces are stored in the same database.

The goal of our benchmark is to see which implementation is the best, considering as metric execution time, disk usage and memory usage for three different use-cases. The first experiment is focused on disk usage, the second on trace import time and the third on time and memory usage with multi-trace analysis result production and saving.

We conclude this study saying that the overall performance of the two implementations is not so different in the first two experiments, while in the last one DDB outperforms CDB. For trace storage size and trace import CDB is actually slightly better than DDB, because it does not replicate the format and does not create a DB for each trace, but these kind of optimization produce a gain which is basically constant : for large traces and long import times it becomes absolutely negligible. Considering the large traces results, in the first experiment CDB is less then 0.1% better and in the second experiment is about 2% better. Coming to the third experiment, we notice here that DDB performs better than CDB and the gain is not independent from trace size : on the contrary, dealing with larger traces makes the difference between the two implementation bigger. Summarizing, with single-connection, CDB memory usage is comparable but time values are up to 160% worse; with multi-connection, time results are still worse (about 25% slower) and memory usage is worse too (about 5% more memory).

With these results, we conclude that DDB is globally a better solution than CDB from a performance point of view. The added complexity in the DB schema and the software library introduced to support the catalog solution is therefore not justified. In addition to this, we can do some non-functional considerations about CDB solution. First of all, given that all the possible applications working on top of SoC-Trace Infrastructure deal with the same database instance, preserving the DB consistency becomes more difficult. Furthermore, direct DB interaction is less straightforward because of the catalog solution itself. Finally, with the separation of trace databases, DDB leaves the place for the usage of embedded DB solutions (like SQLite) that have the limitation of one file for one DB, as an alternative to MySQL. DDB is therefore the solution of choice for SoC-Trace Infrastructure.

# A Database performance

The benchmark described in this document is mainly focused on comparing two different implementations of SoC-Trace Infrastructure. Anyway, from the results obtained we can do some observations also regarding absolute performance with whatever implementation of the system (CDB and DDB). Basically we observe that the time spent to perform an import (parsing a trace and putting it in the DB) is always quite long : almost 6 minutes to import 100 MB of raw trace data. In the technical report [2] we observed that most of import time (about 98%) was spent in writing into the DB. As observed in Figure 8, both implementations work on the same software environment where, at the bottom, we have the MySQL DBMS and the connection with the Java code through the network and the JDBC driver. In this appendix we point out some preliminary analysis on this DB write performance issues.

In the first place, we consider again the system software layers (Figure 8) and we try to isolate the problem. The writing issue in fact can be due to a problem in the SoC-Trace Library, in the JDBC driver, in the latency of the network connection or at the DBMS level.

To start, we exclude all the Java part of the system (SoC-Trace Library and JDBC driver) to see if there are issues at the DBMS level. For this purpose we simply dump a trace database to an SQL file using the `mysqldump` tool and then we re-execute this SQL script to re-import the database, measuring the time to re-import it.

The commands used are :

- make the dump : `mysqldump -u root database > database_dump.sql`
- re-import the database : `mysql -u root database < database_dump.sql`

The trace considered is the 100 MB one, for which the import time was around 350 s using the KPTrace parser. We observe that restoring the DB from the dump takes a lot less time : only 43 seconds on the average (computed over 30 repetitions). Figure 15 clearly makes the point. The KPTrace parser is more than 8 times slower than the dump restore, so apparently the writing issues is in the Java part of the system. Furthermore, from now on, we can consider the time taken by a direct dump restore as a reference (a minimum) for our following tests.

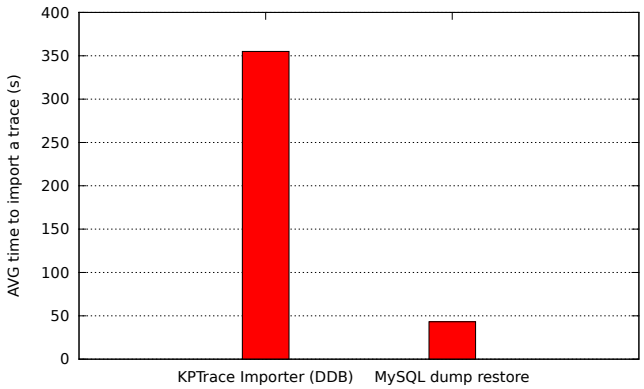


FIG. 15 – Comparison between KPTrace importer and MySQL dump restore

Considering only the Java layers of the system, we are interested in understanding if the problem is in the SoC-Trace Library or below it, in the JDBC part. To exclude that the Library (or even the application - the parser - above it) has issues, we wrote a very simple test program, that simply stores event values in EVENT and EVENT_PARAM tables, using directly JDBC prepared statements with batch execution (the storage part of the Library is not used at all). For

this test we consider a synthetic trace composed by 4000000 of events, each having 2 parameters. The events (all identical) are generated dynamically and no actual trace file is considered.

For all the following experiments, we performed 30 repetitions.

To initialize the system we run a first time the test program to have a trace database and dump it, in order to be able to re-import it using MySQL command line client. Then we run both the dump restore and the program. The results are shown in Figure 16.

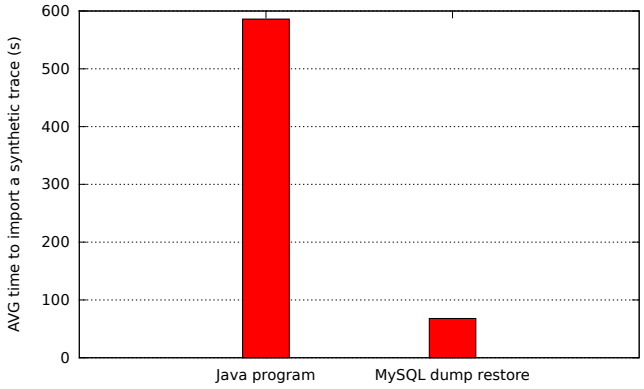


FIG. 16 – Average import time for a synthetic trace of 4000000 of events (lower is better)

Even the simple Java program directly using JDBC prepared statements without any (possible) SoC-Trace Library overhead is more than 8 times slower than the MySQL dump restore. Being the ratio between the Java program and dump restore basically unchanged, we may conclude that the write issue, though being in the Java part, is not in the SoC-Trace Library.

We investigated further in order to understand whether the problem is at the JDBC driver level or at the network connection level.

Considering only the network connection between a MySQL client and the server, on a Unix system we can have two different situations :

- TCP/IP connection to local or remote server.
- Unix socket file connection to local server (more efficient).

Being the MySQL JDBC driver a JDBC Type 4 driver, it does not use any native code to access the database. Therefore, given that Java has no method to access Unix sockets, the driver can not use them either, thus not being able to take advantage of local server optimization. To fix this issue, we used the `junixsocket` library [5] together with the provided socket factory for MySQL driver. This way we can compare the difference in the execution time of our simple test program with or without using the Unix socket optimization. Results are in Figure 17.

We notice that using Unix sockets performance gets better of about 10%, but the result is still far from the command line dump restore. The write speed issue is therefore not mainly due to the connection latency, but is to be found in MySQL JDBC driver implementation.

To further analyze the MySQL JDBC driver implementation behavior we perform the import of the synthetic trace using the following approaches :

- JDBC prepared statements with batch execution (`PS batch`)
- JDBC prepared statements without batch execution (`PS no batch`)
- JDBC simple statements with batch execution (`SS batch`)
- JDBC simple statements without batch execution (`SS no batch`)
- JDBC simple statements with *manual* batch execution (`Manual`)

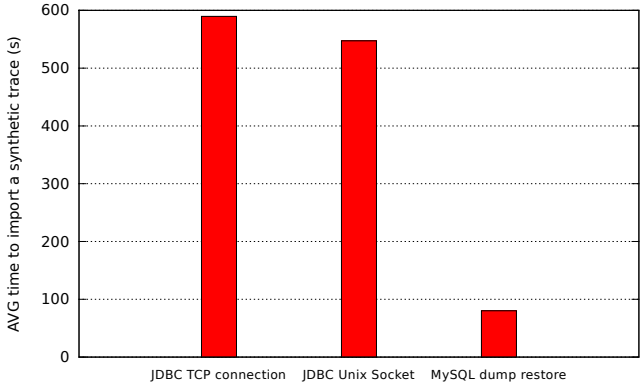


FIG. 17 – Average import time for a synthetic trace with and without using Unix sockets (lower is better)

In all MySQL configurations we use the Unix socket connection analyzed above to remove this cause of speed loss. By *manual* batch execution we simply mean that we manually implemented in Java the batched execution of several inserts, without passing through JDBC batch functionalities, but simply using simple statement with several rows passed to a single INSERT command. For the first four configurations (PS batch, PS no batch, SS batch and SS no batch) we run the test program also using another DBMS (SQLite) and therefore another JDBC driver implementation for comparison purposes. The results are shown in Figure 18, where the last red bar is, as always, the average time obtained restoring the dump, which is our reference.

Considering only MySQL (red bars) we notice that all *standard* JDBC modes are pretty much similar and way slower than our reference (the dump restore). On the contrary the manual implementation of batch execution is extremely similar to the reference (only 5% slower, because of the overhead introduced by Java). This is a strong argument to conclude that the MySQL JDBC driver implementation has a critical issue in managing buffered inserts. Another argument is that, considering only the first four configurations, for both prepared statement (PS) and simple statements (SS), the buffered execution is slower than the non buffered one, which is very couterintuitive and unexpected. Furthermore, using prepared statements or not does not affect a lot the performance. On the other side, considering SQLite implementation of JDBC driver, we observe a more intuitive behavior :

- batched execution is faster than non batched one (from 5% to 2% using or not using prepared statements respectively)
- using prepared statements is faster than non using them (more than 200% faster)

Finally, comparing the best MySQL result (the manual one) with the best SQLite result (prepared statements with batched execution) we observe that the SQLite solution is almost 3 times faster than MySQL. This is most likely due to the fact that SQLite is an embedded DB solution, while MySQL introduces some overhead because of its client-server architecture.

This conclusion, in addition to the fact that the DDB implementation of SoC-Trace Library proved to be preferable, is a strong argument in choosing SQLite as DBMS for SoC-Trace Infrastructure : the distributed architecture, in fact, having a single trace in each Trace DB, is not really affected by the single-file constraint imposed by SQLite (DB size is upper-bounded by the file system limits on the file size). On the contrary, with CDB, putting all the information managed by the system in a single file (single SQLite DB) won't scale. This database performance appendix finally enforces the choice of DDB.

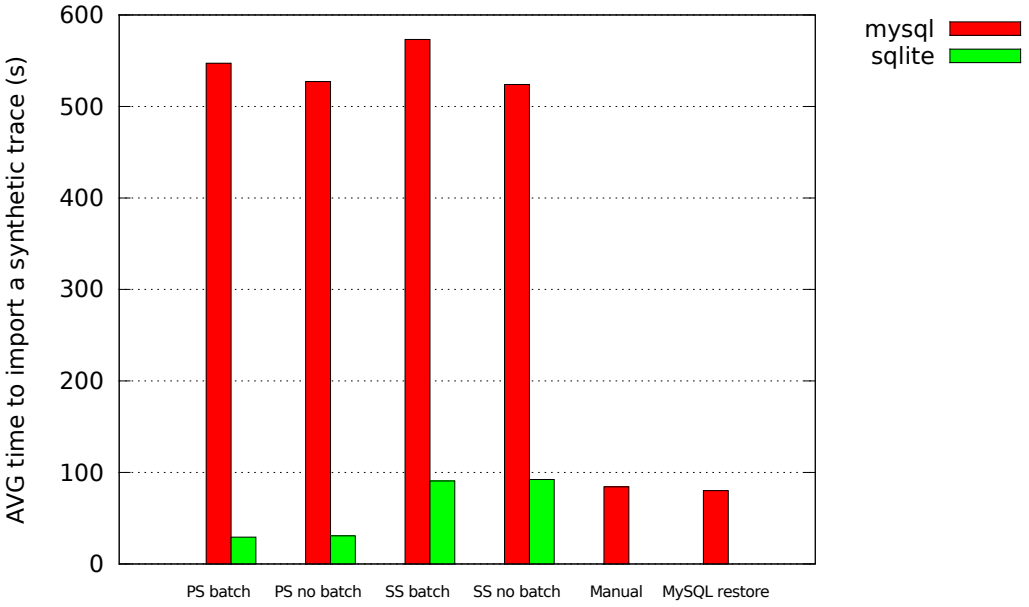


FIG. 18 – Average import time for a synthetic trace with different JDBC configurations (lower is better)

# B SoC-Trace Infrastructure Current State

After the analyses reported in this document, SoC-Trace Infrastructure has evolved keeping the DDB solution. In this appendix we briefly summarize the principal improvements to our infrastructure, which now is simply referred to as FrameSoC[4]. More details about FrameSoC and some concrete use-cases showing its current possibilities are given in the research report RR-8304 [6].

In the bibliographic study conducted in [6] about database solutions for trace management, we found that most of such solutions ( [7], [8] and [9]) use a distributed approach, basically for scalability or simplicity of implementation. This corroborate our choice of DDB as the reference implementation of our infrastructure.

The principal improvements to FrameSoC can be grouped in three categories : database management, tool management, graphical user interface. Each category is detailed below.

## B.1 Database Management

The performance analysis conducted in Appendix A shows the interest of experimenting different DBMS with the infrastructure. For this reason, we designed and implemented the possibility to work with different DBMS, provided that a simple adaptation module is provided for a specific DBMS technology.

More in details, inside the `DBObject` there is a `DBManager` that encapsulates all the DBMS-specific operations. The `DBManager` is actually an abstract class : concrete subclasses are provided for real DBMS. At the moment being concrete DB managers have been implemented for MySQL

[4]By FrameSoC we mean our Framework for System-on-Chip execution-trace management

and SQLite.

This multi-DB feature on one hand allows easy experimentation with different DBMS and on the other hand makes FrameSoC independent from a specific DBMS technology.

Still at the database interaction level, new `Visitor` classes have been implemented in order to support also *update* and *delete* operations on the object of the data-model.

## B.2 Tool Management

A great improvement in tool management is the support of the Eclipse [10] plugin mechanism to add tools to FrameSoC. In fact, we defined an extension point for FrameSoC tools, defining the interface for FrameSoC compliant tools developed as Eclipse plugin. The extension point has the following fields :

**class** a class extending the abstract `FramesocTool`, containing a the launching method.

**type** the tool type, which is one of *ANALYSIS*, *IMPORT* and *EXPORT*[5].

**name** the tool name, as presented to the user.

**doc** the tool launching documentation, to be presented to the user.

A user wanting to develop a FrameSoC tool in the form of an Eclipse plugin, has simply to extend our extension point, providing the necessary fields. FrameSoC is then able to interrogate the Eclipse plugin registry in order to be aware of the FrameSoC tools actually present, thus facilitating the installation and the launching of such tools. Note that, even with the plugin mechanism, the user keep the control on the tools actually registered into the FrameSoC DB, since such registration is done explicitly using the GUI described below. Of course, the possibility to manage black-box (non-plugin) tools is still present, and all kind of tools (plugin or not) are registered or launched with the same user interface.

## B.3 Graphical User Interface

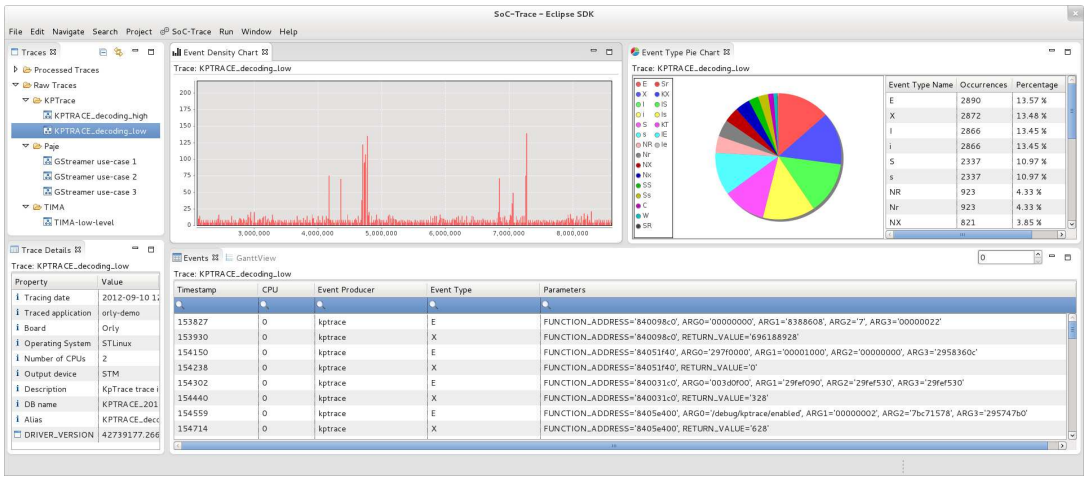


FIG. 19 – FrameSoC GUI

[5]Note that now, in addition to *ANALYSIS* and *IMPORT* tools, also *EXPORT* tools are explicitly supported by the framework.

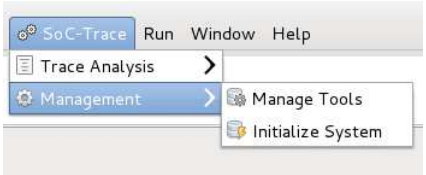


FIG. 20 – FrameSoC Management Menu

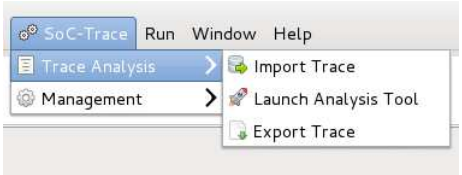


FIG. 21 – FrameSoC Trace Analysis Menu

A significant improvement to our infrastructure is the development of a powerful GUI based on Eclipse, facilitating both management and analysis tasks for the framework user. Figure 19 shows an overview of FrameSoC GUI, with some *framework* tools, enabling basic analyses : a structured trace explorer with details on trace metadata, an event-density chart to easily identify trace hot spots, a pie-chart showing some statistics about the trace and a form for event querying using regular expressions.

Using the SoC-Trace menu on the top of the GUI, the user may access to Management (Figure 20) and Trace Analysis (Figure 21) menus.

The management functionalities include :

- Initialize the system, choosing the DBMS and specifying its parameters.
- Manage tools : add, remove, edit tools. The possibility to look for FrameSoC compliant plugin-tools is also provided.

The analysis functionalities include :

- Import a trace to the framework, using a registered importer tool.
- Perform an analysis, launching a registered analysis tool.
- Export trace data, using a registered exported tool.

# References

[1] FUI (Fonds Unique Interministériel). `http://competitivite.gouv.fr`.

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

[3] MySQL Table Partitioning. `http://dev.mysql.com/doc/refman/5.6/en/partitioning.html`.

[4] R. K. Jain. *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley, 1 edition, April 1991.

[5] junixsocket. `http://code.google.com/p/junixsocket/`.

[6] Generoso Pagano, Damien Dosimont, Guillaume Huard, Vania Marangozova-Martin, and Jean-Marc Vincent. Trace Management and Analysis for Embedded Systems. Rapport de recherche RR-8304, INRIA, May 2013.

[7] Rolf Borgeest and Christian Rodel. Trace Analysis with a Relational Database System. In *Parallel and Distributed Processing, 1996. PDP'96. Proceedings of the Fourth Euromicro Workshop on*, page 243–250, 1996.

[8] Guillaume Pothier and Éric Tanter. Back to the Future : Omniscient Debugging. *Software, IEEE*, 26(6) :78–85, 2009.

[9] Carlos Prada-Rojas, Miguel Santana, Serge De-Paoli, and Xavier Raynaud. Summarizing Embedded Execution Traces through a Compact View. In *Conference on System Software, SoC and Silicon Debug S4D*, 2010.

[10] Eclipse. `http://www.eclipse.org`.

**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