Efficient Analysis Methodology for Huge Application Traces

Damien Dosimont[*][†][‡] [§], Generoso Pagano[*][†][‡] [§], Guillaume Huard[†][‡][*] [§], Vania Marangozova-Martin [†][‡][*] [§]
and Jean-Marc Vincent[†][‡][*] [§]
[*]*Inria*
[†]*Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France*
[‡]*CNRS, LIG, F-38000 Grenoble, France*
[§]*firstname.lastname@imag.fr*

***Abstract*—The growing complexity of computer system hardware and software makes their behavior analysis a challenging task. In this context, tracing appears to be a promising solution as it provides relevant information about the system execution. However, trace analysis techniques and tools lack in providing the analyst the way to perform an efficient analysis flow because of several issues. First, traces contain a huge volume of data difficult to store, load in memory and work with. Then, the analysis flow is hindered by various result formats, provided by different analysis techniques, often incompatible. Last, analysis frameworks lack an entry point to understand the traced application general behavior. Indeed, traditional visualization techniques suffer from time and space scalability issues due to screen size, and are not able to represent the full trace. In this article, we present how to do an efficient analysis by using the Shneiderman's mantra: "Overview first, zoom and filter, then details on demand". Our methodology is based on FrameSoC, a trace management infrastructure that provides solutions for trace storage, data access, and analysis flow, managing analysis results and tool. Ocelotl, a visualization tool, takes advantage of FrameSoC and shows a synthetic representation of a trace by using a time aggregation. This visualization solves scalability issues and provides an entry point for the analysis by showing phases and behavior disruptions, with the objective of getting more details by focusing on the interesting trace parts.**

Keywords*—*Application analysis, trace management, analysis tools, visualization tools, debugging, performance analysis

I. Introduction

Nowadays, computer systems are made of increasingly complex hardware and software components. Their hardware architectures are possibly multicore, heterogeneous and distributed. Their software stack is composed of numerous layers including, for example, middlewares to abstract the platform [1]. In this context, application debugging and performance optimization become tremendously difficult tasks.

By tracing the application, the analyst gathers low-level information on its execution (function calls, thread or process execution states, interruptions, CPU load, memory usage, hardware counters). In debugging case, the objective is finding the cause of a perturbation or an undesirable behavior. In performance optimization, the analyst looks for bottlenecks and less efficient algorithms and code parts. Following Shneiderman's principle [2], an analysis starts by an overview of the trace, showing general information. Then, the analyst focuses on the interesting parts (visible perturbation, particular phase), and filters noise. Finally, he get details on demand, e.g., access to source code. This process can be iterative if necessary. However, several issues hinder this analysis flow:

Big Trace Management: Computer program traces may contain a large quantity of events (for example, we get several million events for a dozen of seconds of G-Streamer video decoding). High quantity of information in the trace translates into a large data volume to store and load in program memory for analysis. In particular, access to trace data is slow because it is often sequential or mono dimensional. In worst cases, the analyst cannot even access to the trace because of performance and memory. Efficient trace management storage is thus a mandatory first step to do an efficient analysis.

Analysis Flow Support: An effective analysis typically involves several treatments on traces, either on raw data, or within a flow where the result of one computation is reused as an input of another (for instance, filter the trace, process filtered data, then visualize the result of this processing). Usually, because of the variety of analysis techniques and tools, output data is not standardized. Thus, the analysis flow requires an adaptation to enable data sharing between tools. This leads to a strong software complexity, whereas output data standardization would provide a straightforward compatibility.

Trace Overview: Analysis first step requires to show a synthetic view of the trace. Traditional techniques, like Gantt Chart, are used to represent trace events over time and space. However, these techniques suffer from scalability issues, because the level of detail is too high. Using a finite screen, representing one million events leaves only one pixel for an event. This leads to cluttered drawings, non-exact proportions or uncontrolled visual aggregation. Zooming or panning, to counter these issues, provoke context loss. Aggregating the events is an other tentative to represent the full trace, but existing solutions cause an important information loss.

We solve these three main issues with two contributions. The first one is FrameSoC [3], a new trace management and analysis framework. With FrameSoC, we manage large traces by providing a database storage solution, where trace information is represented with a generic data-model. FrameSoC features an interface to get and filter trace information, which optimizes access time to data and avoid memory saturation. Regarding the analysis complexity issue, which represents our main challenge, we propose facilities to enable analysis flows, by expressing and storing analysis results using a common format. Moreover, we can plug to the infrastructure various analysis tools, like statistics modules, filters, data mining engines and visualizations, using a generic interface. Our second contribution is Ocelotl, a visualization tool employing time aggregation techniques to represent a trace synthesis. Its objective is to provide the analyst an entry point to the analysis. By interacting with the visualization, the user gets information such as phases (initialization, steady states) or behavior disruptions. Moreover, compared to other aggregation techniques, Ocelotl gives the user the control over information loss. The tool is plugged into FrameSoC and takes advantages of its features, such as data queries, event filtering and result management. In this article, we present successively these two contributions (Section II, III, IV), by evoking for each one related works, theoretical aspects and implementations. In Section V, we detail a complete analysis flow, from the overview provided by Ocelotl to more detailed information, by using case studies. This part has the objective of validating our analysis methodology across a real example. In particular, it will highlight the synergy between both contributions and their respective features. We will conclude in Section VI by proposing new features and improvement for the analysis.

II. FrameSoC: Trace Management Framework

A. Existing Solutions for Storing Traces

Traditionally, raw trace data are stored in plain files (event logs), with no specific support for optimized random accesses or filtering. As a consequence, the analysis requires to load the whole file into main memory [4]. Other approaches propose the use of a structured trace file, more suitable for specific kinds of access. A frame-based file format [5], for example, enables fast time-guided navigation. Another structured format [6] optimizes accesses in both time and space (processes) dimensions. These approaches help the access to trace information only in a fixed and limited number of dimensions and are not flexible for arbitrary selections. A different approach for storing traces is the use of a database, which ensures scalability while keeping flexibility for data-access. Some of the database solutions proposed in literature only provide the support for a single trace format (e.g., [7], [8]), while other solutions are more open to different trace formats (e.g., [9], [10]).

B. Our Database Solution for Trace Storage and Management

FrameSoC addresses the issue of huge trace storage by using a relational database. Several pragmatic motivations led us to this choice. First, a database separates the logical data-model from the physical representation of data. Furthermore, thanks to accurate modeling and normalization, information is stored with minimal redundancy. Then, we can easily access parts of the trace or filter noise by using trivial querying. Search operations can be optimized by defining indexes: this mechanism is flexible and not limited to time or space dimensions. Finally, complex computations on trace data can be performed in the database, instead of loading the whole trace in memory and do such computations at the application level.

The core of our database solution is the generic data-model. It represents trace metadata, trace raw data and analysis results, with related tools metadata (Figure 1). The central entity of the model is the trace, which has metadata and can be related to files (e.g., configuration files, platform description).

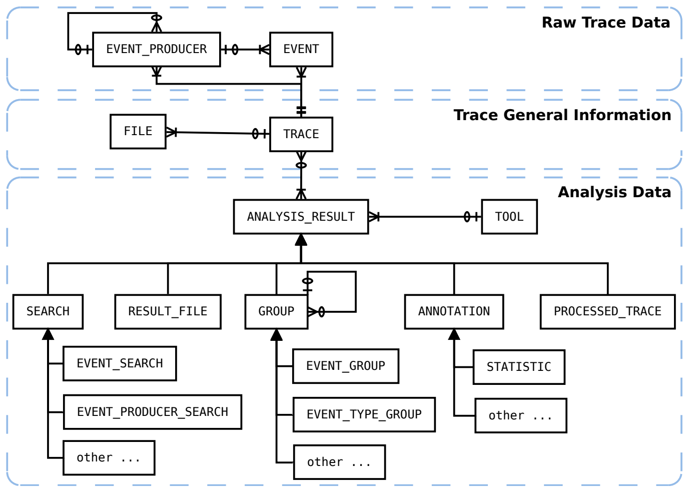


Fig. 1. Generic data-model for trace management (Crow's Foot notation)

A trace is composed by several events, each of them having an active entity producing it. Event producers can be organized in a hierarchy, reflecting, for instance, the execution hierarchy in the traced application (processes/threads). This model is innovative since, beside trace data, it provides some predefined but extensible types of analysis results, with the link to the corresponding analysis tools (Section III). Our data-model is actually a new self-defining trace format (like SDDF [11] or Pajé [12])), since the description of trace types and event types is part of the stored information. Using this approach we obtain a generic trace representation, with minimal semantics and suitable for representing any kind of trace format without information loss. At present, we have managed to represent with our model KPTrace [13], Pajé [12]. A Java API (FrameSoC library) is provided to easily interact with this data-model. Given the richness of our data-model, the role of the database is central in our solution. Indeed, we use the database to manage several traces, store analysis results produced on such traces, and also organize the tools producing such results. None of the aforementioned existing database solutions consider multi-trace requests (e.g., to identify a subset of traces for a multi-trace analysis) or the generic storage of results, and analysis tools are not taken into account.

To be independent from a given DBMS technology, our infrastructure is designed to be able to work with different DBMS (DataBase Management System), provided that a simple adaptation module is implemented: at this time, support for MySQL and SQLite is provided. With the aim of providing a simple and scalable solution, we store each trace in a different database and all trace databases are coordinated using a central system database. When considering storage scalability issues, none of the supported DBMS limits the number of databases managed. Considering the database size, in the case of MySQL a table can grow up to the maximum file size (4 TB on ext3 file systems) and there are partitioning techniques to manage tables exceeding this limit. For SQLite the actual database size limit is fixed by the file system maximum file size.

C. Performance Measurements

To show that the proposed database solution is effective when analyzing data over several dimensions, we present in this section some performance results. The DBMS used is SQLite. We use synthetic traces, where different event producers and event types are uniformly distributed over time. The workstation used has a 3.30 GHz x 12 CPU, a 256 GB SSD and 16 GB of DDR3 RAM.

1) Importing Traces of Various Sizes Into the System: We imported traces of different sizes, ranging from 5.5 MB (100 thousands of events) to 2.75 GB (50 millions of events), measuring the import time with and without indexing. Import time (Figure 2) grows linearly with trace size in both cases, as proved with a linear regression showing a coefficient of determination R^2 of $1 - 10^{-4}$. Import times keep reasonable values even for huge traces (without indexes about 7.5 minutes for a 2.75 GB trace). Using indexes, the import times grow by about 75%.

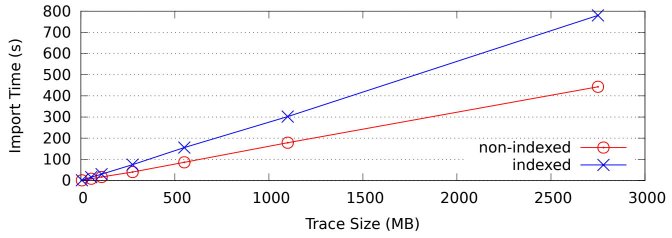


Fig. 2. Import time for traces of various sizes, with and without indexing.

2) Querying a Given Trace over Different Dimensions: A great advantage of using a database for storing traces is the flexibility it offers when performing requests in various dimensions. Using a synthetic trace of 2 million events, we performed requests to retrieve events respectively in a given time interval (a), from a given producer (b), of a given type (c), or having a given value for a parameter (d). For each request, the result set has the same size (20000 events). No indexing has been used in databases. The time needed to filter trace events using each of the four different dimensions (Figure 3) remains in the same order of magnitude. This confirms that the joint use of a well designed data-model and database technology lets trace analysts explore a given trace from different perspectives at a comparable cost. On the contrary, a structured-file trace format as OTF [6] optimizes only producers and time dimensions.

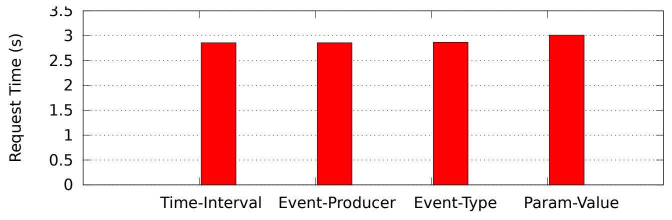


Fig. 3. Time to retrieve 20000 events from a 2 million events trace, using various dimensions for filtering.

3) Evaluation of Trace Size on Request Time: One of the interests of putting huge trace data in the database is information retrieval, limiting the effects of trace size. For this reason we retrieved a fixed number of events (10000) contained in a time interval from traces of different size (from 5.5 MB to 2.75 GB), measuring the request time (Figure 4). Ideally, we would like the retrieval time to be constant, given that the result set size is fixed; however, without any indexing, the retrieval time grows linearly with trace size (from less than 1 s to 60 s), as confirmed with a linear regression showing an R^2 of $1-10^{-6}$. Performing careful indexing at the database level, we actually managed to get near constant retrieval time (less than 0.1 s), paying only at import time the indexing price.

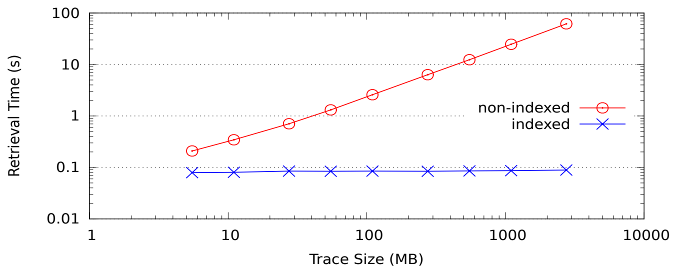


Fig. 4. Time to retrieve 10000 events from traces of various sizes, using indexed or non-indexed databases.

4) Dealing with Gigantic Traces: To test the limit of our system, we imported a gigantic trace of 110 GB, containing almost 2 billions of events. For reasons of disk space, we used a standard hard disk drive of 1 TB for our experiments. Table I shows the whole trace import time and the request time to retrieve 10000 events contained in a given time interval. We use both indexed and non-indexed databases. For comparison, we tried to do the same kind of filtering also directly on the raw trace file (event log) using the *awk* program. Database import time is significant given the trace volume, especially with indexing. The results are however still linear with the ones obtained in the first experiment (the small increment in time is due to magnetic disk performance). The interesting point is that, even for this gigantic trace, we manage to filter the events: without indexing the time is huge, but using indexes, filtering time is extremely small and similar to the results obtained for traces of smaller size (third experiment). On the contrary, the manual filtering on the raw trace file has a duration not suitable for interactive analysis.

	Non-indexed DB	Indexed DB	Raw trace file
Import the trace (hours)	5.9	9.9	-
Filter 10000 events (seconds)	42232	0.12	875

TABLE I. GIGANTIC TRACE RESULTS

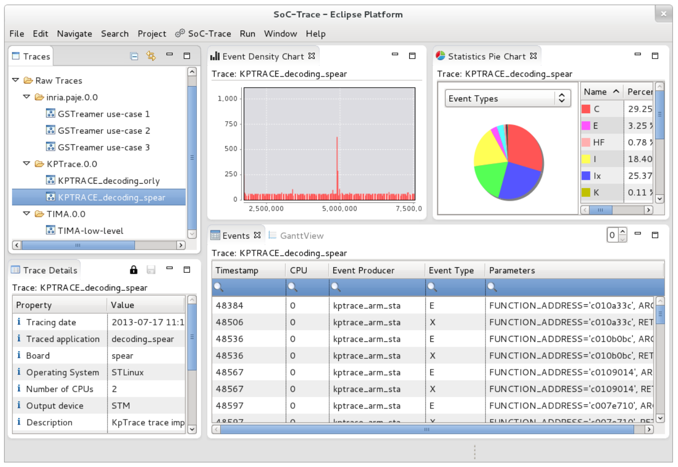
This test shows that our framework, taking advantage of database features, enables fast access to trace data for analysis purposes even when dealing with gigantic traces.

III. ANALYSIS FLOW MANAGEMENT WITH FRAMESOC

A. Existing Solutions for Tool and Flow Management

The need for differentiated analysis of traces forces the analyst to face a situation of extreme tool heterogeneity, with consequent compatibility issues, since specific tools tend to work with specific formats [14], [15]. In the field of parallel-systems, different solutions have been proposed to address this problem. The visualization tool Pajé [4] adopts a modular structure, where different modules can be plugged to the analysis flow by using semantic-agnostic interfaces. However the creation of a new analysis flow is static and requires reassembling the different modules in a new program. Score-P [16] measurement infrastructure tackles tool heterogeneity by multiplexing/demultiplexing different instrumentation types to different output formats, without the notion of shared data-model, neither for trace data nor for analysis results. With the same philosophy, Tau [17] provides a trace analysis environment where the interaction among different tools is obtained via trace translators. A shared data-model exists only for trace profiles. In the domain of embedded systems, existing frameworks for trace analysis are even more specific to given formats or hardware platforms, so that no actual support for generic tool interaction exists. Proprietary solutions (e.g., [18]) typically offer a closed set of functionalities tailored to specific hardware. Even open source solutions (e.g., [19]) do not easily enable the plugging of new tools and do not support tool interaction through a shared data-model for analysis results.

B. *FrameSoC Tool Management and Workflow Support*



Name	Percen
🟥 C	29.25
🟪 E	3.25
🟥 HF	0.78
🟨 I	18.40
🟦 Ix	25.37
🟨 K	0.11

Trace: KPTRACE_decoding_spear

Property	Value
Tracing date	2013-07-17 11:1
Traced application	decoding_spear
Board	spear
Operating System	STLinux
Number of CPUs	2
Output device	STM
Description	KpTrace trace imp

Trace: KPTRACE_decoding_spear

Timestamp	CPU	Event Producer	Event Type	Parameters
48384	0	kptrace_arm_sta	E	FUNCTION_ADDRESS='c010a33c', ARC
48506	0	kptrace_arm_sta	X	FUNCTION_ADDRESS='c010a33c', RET
48536	0	kptrace_arm_sta	E	FUNCTION_ADDRESS='c010b0bc', ARC
48536	0	kptrace_arm_sta	X	FUNCTION_ADDRESS='c010b0bc', RET
48567	0	kptrace_arm_sta	E	FUNCTION_ADDRESS='c0109014', AR
48567	0	kptrace_arm_sta	X	FUNCTION_ADDRESS='c0109014', RET
48597	0	kptrace_arm_sta	E	FUNCTION_ADDRESS='c007e710', ARC

Fig. 5. FrameSoC GUI based on Eclipse. On the left: a trace explorer, along with a related detailed view. On the right: an event density chart, a statistics pie-chart and a searchable table of events.

With FrameSoC, we propose a framework for trace analysis, where tool management and tool cooperation are central points. FrameSoC is based on the Eclipse platform[1] and has been designed to be easily extensible and suitable for building rich analysis flows. The integration and cooperation of tools within FrameSoC is made possible by two elements of our framework: the expressivity of our data-model and the plugin mechanism we propose. As anticipated in Section II, the generic data-model enables to represent both trace data and some predefined (but strongly customizable) analysis results. More in details, the model provides the following analysis result types: searching/filtering results, custom files with tool-dependent semantics, grouping results to model patterns of events or event types, generic trace annotations and processed traces obtained by enriching or adding levels of abstraction to raw traces. With these types of results, the idea is to provide a well-defined data format to store the results of different analysis, in order to avoid time-consuming recomputations and enable tool cooperation. Indeed, a tool can easily retrieve the result of another tool and use it to perform its analysis, since the results are stored with a well-known format. The support provided by FrameSoC to tool enchainment is semantically agnostic and all the analysis logic remains within the tools. FrameSoC helps the contribution of new tools to the framework with a clean plugin mechanism based on the Eclipse one. Indeed, a preferred way to add a tool to FrameSoC is to provide an Eclipse plugin that implements the interface we defined through an extension-point. This extension point defines the metadata and the class the tool plugin should provide in order to be integrated in FrameSoC. However, our infrastructure also supports the possibility to integrate external back-box tools. In both cases, tools deal with the same data-model for trace and result storage, and are launched using the same interface.

[1]http://www.eclipse.org/whitepapers/eclipse-overview.pdf

The prototype implementation of FrameSoC itself provides some *framework* tools, to enable basic trace analysis (Figure 5): a structured trace explorer with details on trace metadata, an event-density chart to easily identify trace hot spots, a pie-chart gathering some statistics about the trace and a form for event querying using regular expressions. The infrastructure explicitly supports the plugging of trace importers, trace exporters and more general analysis tools. At this time, we plugged tools able to import real traces (KPTrace, Pajé formats) and to export into Pajé format. As for analysis tools, we integrated a tool able to perform simple sequence-search with result saving and a filter for event producers, able to find and save the subset of producers being active (or idle) during a given time interval. Finally, we also propose an innovative visualization tool, Ocelotl, able to perform aggregation (Section IV).

IV. Ocelotl: Trace Overview Module

This section describes Ocelotl, an innovative visualization tool plugged into FrameSoC. This tool is used to highlight FrameSoC ability of helping the analysis flow. Ocelot aims at showing a trace overview, answering to both time and space scalability issues. The trace is cut into time slices and represented as a sequence of representative elements. This sequence is constructed using an aggregation algorithm that identifies consecutive parts of the trace showing a similar behavior, and aggregates them.

A. *Trace Overview Existing Approaches*

Existing analysis tools use different approaches to provide a trace overview. Statistics representation, such as graph or bar charts, may represent metrics over time. These kinds of representations are proposed by KPTrace [15] with its Outline View, and are convenient to distinguish CPU activity, for instance. However, the notion of software and hardware hierarchy is totally missing, so the space dimension cannot be studied with this technique. On the contrary, other KPTrace statistic techniques [15] or those provided by LTTng Eclipse Viewer [19], show activity time proportion for each event producer. But here, the drawback is the lack of time dimension representation (aggregation is done on the full trace), and the analyst cannot observe the process behavior over time.

Another approach is based on time views, like Gantt chart representation [20]. It is classically used to visualize application behavior over time, thanks to its ability to represent causality relations. However, because of the amount of information to visualize (due to the events granularity, the platform heterogeneity or the execution duration), an analyst may be forced to zoom out or to pane, thus losing either the execution context or the representation fidelity. A partial solution to this problem is proposed by Pajé [4] and LTTng Eclipse Viewer [19]. Both tools highlight the events that are too small to be correctly represented using pixels. They use a specific shape/color to represent an aggregation of these groups of events. However, even if such technique shows the possible information loss, it lacks associated semantics that would help the analyst to understand the trace.

Another major issue in providing trace overviews is the hierarchy representation. The space axis in Gantt charts, for example, may be used for this purpose, but the user may scroll and lose the context. In KPTrace Gantt chart [15], the hierarchy associated with a given core can be collapsed and represented as part of the root of the hierarchy. Unfortunately, it is not possible to distinguish which child an event belongs to, which may be confusing. In the Vampir [14] task profile view, event producers are clustered using a proximity metric, like the function duration. This representation, however, fails in showing causality relations. Triva [21] treemap view uses multiple axes for hierarchy representation and show the evolution of the execution over time by using animations. This visualization highlights network bottlenecks and unbalanced workloads, but is not suited to identify problems related to synchronization (deadlock) or scheduling.

B. Build a Macroscopic Description of a Trace

The contribution we propose is a temporal view, where trace areas having a "close" behavior are aggregated. This aggregation is materialized by a rectangle area of a given color. Theoretical background comes from Lamarche-Perrin's works [22], dedicated to the Multi-Agent System macroscopic analysis. From a microscopic view, the analyst gets a macroscopic representation that has its own semantic and enables to analyze the system with a different point of view. The way to generate this system macroscopic description is the data aggregation. This process involves three concepts: information loss, complexity reduction and macroscopic semantic. Information loss is useful to determine element proximity. It is calculated from Kullback-Leibler divergence [23] (Eq. 1), which is a metric that represents logical information lost by using an aggregated description instead of the microscopic description. Entropy reduction (Eq. 2), calculated from Shannon entropy [24], represents logical information saved by encoding the aggregated description instead of the microscopic description.

$$\mathrm{loss}(A) = \sum_{e \in A} v(e) \times \log_2 \left(\frac{v(e)}{v(A)} \times |A| \right) \tag{1}$$

$$\mathrm{gain}(A) = (v(A) \log_2 v(A)) - \sum_{e \in A} (v(e) \log_2 v(e)) \tag{2}$$

The knowledge of these two metrics enables to compute a data aggregation, controlling information loss and complexity reduction. More the aggregation is strong (i.e., more elements are aggregated), more the information loss grows and the complexity reduces. On the contrary, a weak aggregation keeps the amount of information but also increases the complexity. What is interesting is to find a compromise between information loss and complexity reduction to build a meaningful macroscopic description. This compromise can be explicitly defined by using *parametrized Information Criterion* (Eq. 3) to find the desired aggregation (the one having the higher pIC).

$$\mathrm{pIC}(\mathcal{A}) = p \times \mathrm{gain}(\mathcal{A}) - (1-p) \times \mathrm{loss}(\mathcal{A}) \tag{3}$$

To adapt these concepts to trace analysis, we need do define a microscopic description. We chose to perform a time slicing of the trace. We generate an array whose index is associated to the temporal position. Each element of the array is a vector, whose elements correspond to the event producers of the trace. The vector values are computed using a particular metric, for instance, the activity time ratio of the associated event producers. However the analyst may be interested by metrics with a richer semantic. For this case, we provide a cubic matrix to perform time-slicing. One dimension is related to the time slice number, the second one to the event producers, and the last one is associated to a chosen metric, as for example the activity time ratio of each state type (e.g., read, write, idle).

The macroscopic description is then generated by applying the Best Cut Partition algorithm [22] on the array. The principle is to aggregate only the temporally contiguous parts, by taking the values of each dimension into account. The first step consists in computing the quality measures (information loss and complexity reduction) for each combination of consecutive cuts. As an example, assume that, at the beginning, there are 4 slices (0, 1, 2 and 3). The algorithm computes a quality measure between 0 and 1 (i.e. aggregate 01), between 1 and 2 (12), between 2 and 3 (23) but also between 01 and 2, between 0 and 12, etc.

As the original algorithm works with scalar arrays, we need to adapt it to vector arrays. The gain and loss metrics associated to an aggregation in n dimensions are respectively the sum of aggregation gains and losses in each dimension. Hence, the new formula, where $\mathrm{quality}(A)$ corresponds to $\mathrm{gain}(A)$ or $\mathrm{loss}(A)$:

$$\mathrm{quality}(A) = \sum_{i \in n} \mathrm{quality}(A[i]) \tag{4}$$

The principle is the same for matrix arrays:

$$\mathrm{quality}(A) = \sum_{i \in n} \sum_{j \in m} \mathrm{quality}(A[i][j]) \tag{5}$$

The second step requires to provide the gain/loss parameter p to compute the *parametrized Information Criterion*, and then, get the corresponding aggregation. For $p = 0$, maximizing the pIC is equivalent to minimizing the loss: a null loss will result in no aggregation, except for strictly identical contiguous vectors. For $p = 1$, the output array will be fully aggregated, resulting in a total loss of information. When p is between these extrema, different aggregation configurations will emerge according to the input vectors values. A list of relevant values of p is computed using a search by bisection, that finds successive parameters that give a different configuration. The objective is then to find the right aggregation parameter corresponding to a meaningful macroscopic description. An example of aggregation applied on random vector data is shown in Table II. Vectors that are aggregated for a given p are represented with a similar number.

TABLE II. EXAMPLE OF AGGREGATION APPLIED TO A VECTOR ARRAY DEPENDING ON THE GAIN-LOSS PARAMETER P

	Vector array with randomly generated values				
	(3, 6, 7)	**(5, 3, 5)**	**(6, 2, 9)**	**(1, 2, 7)**	**(0, 9, 3)**
Gain-loss parameter	**Corresponding parts** (aggregated if same number)				
0 : no aggregation	0	1	2	3	4
0.035 : 4 aggregates	0	1	1	2	3
0.052 : 3 aggregates	0	0	0	1	2
0.078 : 2 aggregates	0	0	0	0	1
0.223 : 1 aggregate	0	0	0	0	0

C. *Interaction to Find the Best Aggregation*

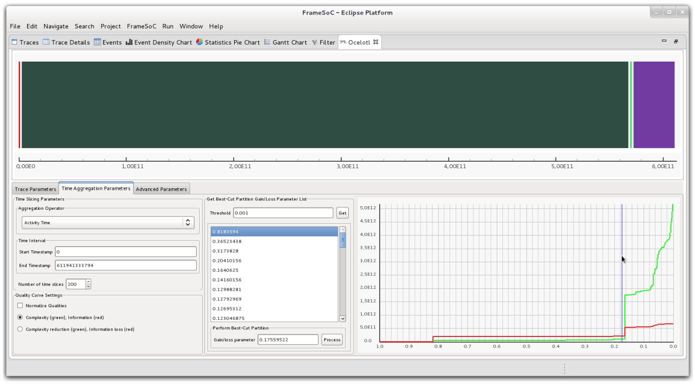
The methodology we propose with Ocelotl implies to find the aggregation whose semantic is meaningful in regard to the analyst objectives. To do that, we propose several interaction mechanisms. First, the user selects the number of time-slices. This number should be chosen according to the screen resolution, but also adapted to the Best Cut Partition algorithm complexity. In fact, the original algorithm complexity is $o(n^2)$. By taking account of the vector and matrix adaptation, it becomes $o((n \times m \times l)^2)$ where m and l are the new added dimensions. Empiric measures show that $n \times m \times l$ should not be superior to 10000 to avoid memory saturation (6 GB required for 10000 elements). After determining the number of time-slices and getting a list of relevant values for the parameter p, the user starts by progressively disaggregating the representation, from the most aggregated to the least one. We provide the two quality curves in function of the parameter value, which the user can interact with. By clicking, he gets the corresponding parameter and thus the related aggregated representation. The aim is to determine the information quantity a new representation brings, compared to the previous one. This feature is interesting to spot disruptions apparition during the disaggregation process. Indeed, a disruption is often related to a jump in the complexity and an information quantity curves. After spotting an interesting trace part, the user can zoom and generate a new aggregation, until the provided representation is quite precise to determine the exact area to focus on with another tool.

D. *Implementation in FrameSoC*

We implement the Best Cut Algorithm in C++ for performance and memory management reasons. Our vector and matrix array management is generic as it has no associated semantics. The code is compiled as a shared library and is accessed through JNI. The Eclipse Java module integrated in FrameSoC is divided in two parts. The core part is in charge of performing queries to the database, using the FrameSoC dedicated interface, and also acquiring the parameters provided by the user and the best cut algorithm output from the shared library. The user interface part provides interaction mechanisms to set or select the different parameters for the queries and the computation. The result is visualized in a frame representing the trace as a one-dimensional array. The parts are emphasized by colors, which are identical for aggregated parts.

V. EXAMPLE OF AN ANALYSIS FLOW

In this section, we present a use case (Table III) based on a basic open-source G-Streamer video application[2], displaying



Time Slicing Parameters
Aggregation Operator: Activity Time
Time Interval
Start Timestamp: 0
End Timestamp: 611941333794
Number of time slices: 200
Quality Curve Settings
☐ Normalize Qualities
◉ Complexity (green), Information (red)
○ Complexity reduction (green), Information loss (red)
Get Best-Cut Partition Gain/Loss Parameter List
Threshold: 0.001
0.8183594
0.36523438
0.3173828
0.20410156
0.1640625
0.14160156
0.12988281
0.12792969
0.12695312
0.123046875
Perform Best-Cut Partition
Gain/loss parameter: 0.17559522

Fig. 6. Ocelotl visualization showing initialization, execution, and application termination. Complexity and information related to this aggregation are weak (curves are shown respectively in green and in red in the bottom right, while current parameter p position is shown by the blue line).

a mpeg video. We introduce an anomaly by using the *stress* tool[3] in order to perturb the video streaming. The trace is then imported into the FrameSoC database. The workstation used for the test has a 2.40 GHz x 8 CPU, a 256 GB SSD and 8 GB of DDR3 RAM.

The first objective is to validate Ocelotl synthetic visualization by relating the trace representation to the application perturbation timestamps. Moreover, we will compare the complexity and information curve behavior with a reference case that is not perturbed. The second aim is to find a way to reduce the trace to the areas involved in the behavior disruption. More precisely, we want to remove the event producers that are not active during this moment (space dimension reduction), and focus on the perturbation timestamps (time dimension reduction). The goal of this step is to minimize further computations, by saving the analysis result into the trace database. This result can then be reused by the overview tool, decreasing initial processing time, or by another analysis tool, like a Gantt chart.

A. *Overview of the Trace with Ocelotl*

We start our analysis by an overview with Ocelotl. By applying the method we evoked above (subsection IV-C), we progressively disaggregate the trace. We first discover a representation showing different phases: a slight initialization phase, at the beginning (0–10 s), and also a termination phase, at the end (550–610 s), corresponding to the period where the application is still active but the video is over (Figure 6). By continuing the process, we get an aggregation step that corresponds to a complexity and an information jump, as shown in the Figure 7. Curve behavior means that the representation semantics changes: we can indeed distinguish several big aggregates (10–550 s). With more disaggregation (Figure 8), we highlight a completely disaggregated area (around 300 s), while other trace parts are still represented by big aggregates. Actually, this area matches with perturbation timestamps, which validates our claim to represent problematic behavior with Ocelotl visualization. We deduce also that during the perturbation, trace behavior becomes unstable, which leads to a heterogeneous area.

[2]https://code.google.com/p/gst-player/

[3]http://weather.ou.edu/~apw/projects/stress/

TABLE III. G-STREAMER APPLICATION EXECUTION CONTEXTS

Use Case	Perturbation: *stress* settings	Streaming behavior	Tracing duration	Trace size	Event producers number	Events number
Reference	Not activated	Normal	10 min	8.7 GB	1507	30000000
Perturbed	After 5 min, 8 CPU workers, 8 Memory workers, during 12 s	Freeze at 5 min, during 12 s	10 min	8.7 GB	1535	29413091

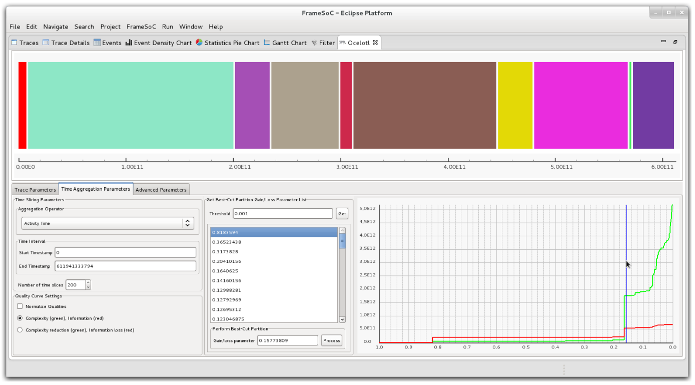


Fig. 7. After passing complexity and information jump, we get several aggregates. Our representation becomes more precise.

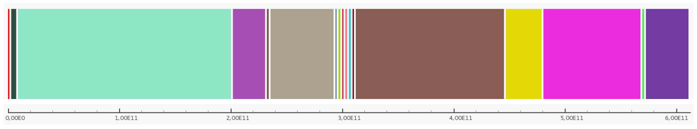


Fig. 8. More disaggregation shows an heterogeneous area around 300 s, which matches to our perturbation timestamps

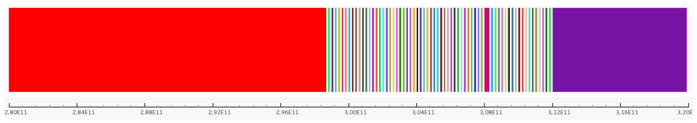


Fig. 9. Zooming on the perturbation (280–320 s). Perturbed area is the heterogeneous part composed by multiple aggregates.

B. Comparison with the Reference Case

We compare the perturbed case with the reference case by using the same methodology. Here, we get the same initialization and termination timestamps. We also obtain a complexity and an information jump. However, we go from a coarsely full aggregated trace to an heterogeneous representation, without intermediary steps where the trace is progressively cut. This phenomenon is related to application stable behavior: the complexity suddenly grows but the new information brought by new aggregates is weak, so the threshold to disaggregate becomes very sensitive. By using both overview and quality measure curve, we are thus able to distinguish a perturbed behavior than a more stable execution.

C. Filtering the Space Dimension

The second analysis step is to reduce event and event producer sets (i.e., the space dimension), to improve further analysis computation. Ocelotl view shows us that there are initialization and termination phases. FrameSoC provides statistic views such as pie chart, which gives the event distribution according to their event producers. In our perturbed case,

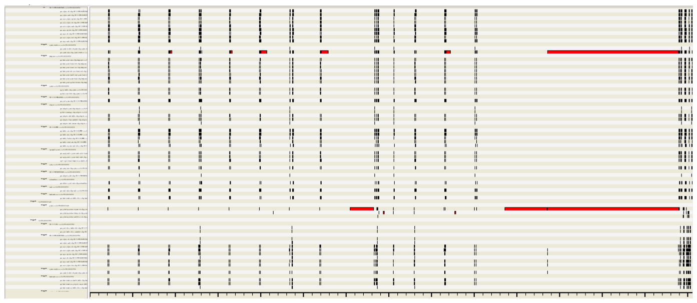


Fig. 10. Here, we focus on the perturbation start timestamps (between 298.4 s and 299.6 s) with the help of a Gantt chart. Regular patterns (end of steady state) are followed by long events that correspond to the beginning of the perturbation.

pie chart coarsely shows that only 20% of event producers generate 80% of trace events. We hypothesize that the 80% less active event producers are only active during boot and end steps, and can be removed without changing the aggregated representation behavior. We design a filter that returns as analysis result a set of event producers that are active during a given period. The objective is to select the event producers being active during the behavior disruption, and remove those being active only during the initialization and the termination phases. So we hope to keep the 20% most active event producers. We filter event producers between 20 and 560 s. Result set contains now only 18% of event producers, which confirms our hypothesis. By viewing the application behavior with Ocelotl again, query time does not decrease, because it is mainly dependent of retrieved event number (which is almost the same here). However, our microscopic description size (event producer dimension is now 18% of the full event producers size) is reduced and this enables to work with 5 times more parts than before (memory complexity of aggregation is $o((n * m)^2)$), which leads to more precision. Finally, the tool produces the same aggregation behavior as for the full trace.

D. Zooming and Filtering Time Dimension

We now focus on the perturbation part by zooming with Ocelotl. The aim is now time dimension reduction, by determining the perturbation timestamps with the best possible precision. By doing several zoom and aggregation, we finally chose 280 and 320 s as bounds (Figure 9). Then, we use a second filter, which saves a set of events that are present during a time period. We now get only 93358 events that are actually relevant to understand trace behavior, i.e., almost 300 times less than at the beginning of the analysis.

E. More Details with Gantt Chart

The final analysis step is a detailed representation of the trace selected area. We visualize the application behavior between 298.4 and 299.6 s with a Gantt Chart, by providing filtering results (Figure 10). Because the event amount and the event producer number are now reduced, the Gantt chart does not suffer from time and space scalability issues as much as before.

F. Analysis Conclusion

Our analysis flow provides an overview of the trace, and then focuses on a precise trace area, with the help of statistics views, filtering tools and result management provided by FrameSoC. The external perturbation we introduced is precisely detectable. The next step will be to introduce a perturbation directly inside the program, to go further in the analysis and, for instance, rely trace behavior to the source code.

VI. Conclusion

FrameSoC manages large traces by storing them in a relational database. Traces are represented according to a generic data-model. The database choice enables filtering and searching in various dimensions, while keeping reasonable read and write performance. Experiments with huge and gigantic traces support this claim. Access to the data being crucial for analysis tools, our future research will consider specific use case requests optimization or request partitioning. The use of alternative storage solutions, such as temporal or non-relational databases, is also a perspective. FrameSoC puts a strong emphasis on analysis tool management and interoperability. Our shared data-model is a basic block for the creation of analysis flows, in which several tools can take part, possibly reusing other tool results. An explicit support is given to tool pluggability: this has been validated by the various tools we have already added to the framework. Regarding the evolution of our framework, we expect to enlarge the family of tools working with FrameSoC. An other interesting perspective is to provide to the final user a convenient interface to define analysis chains.

The visualization module Ocelotl is used as an entry point to the analysis, thanks to its ability to coarsely describe the whole trace behavior over time. With the help of user interaction and a filtering tool, we can reduce space and time dimension elements to focus on those related to a particular behavior, like a perturbation. The use of these different tools, combined with statistic views and result management provided by FrameSoC, corresponds to a coherent and complete analysis flow. Our current work is about the extension of the aggregation technique, to manage also the space representation. Indeed, the space dimension is considered to compute the time aggregation, but it is not represented. An other interesting point would be the improvement of the result management: to avoid useless and time-expensive recomputation, like retrieving the events and generating a microscopic definition each time we open a trace, we could save these results in the database.

References

[1] W. Wolf, “Middleware Architectures for Distributed Embedded Systems,” in *2008 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, 2008, pp. 377–380.

[2] B. Shneiderman, “The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations,” in *Visual Languages, 1996. Proceedings., IEEE Symposium on*, 1996, p. 336–343.

[3] G. Pagano, D. Dosimont, G. Huard, V. Marangozova-Martin, and J.-M. Vincent, “Trace Management and Analysis for Embedded Systems,” in *Proceedings of the IEEEth International Symposium on Embedded Multicore SoCs (MCSoC-13)*, Tokyo, Japan, sep 2013.

[4] J. Chassin de Kergommeaux, “Pajé, an Interactive Visualization Tool for Tuning Multi-Threaded Parallel Applications,” *Parallel Computing*, vol. 26, no. 10, pp. 1253–1274, Aug. 2000.

[5] C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp, “From Trace Generation to Visualization: A Performance Framework for Distributed Parallel Systems,” in *Supercomputing, ACM/IEEE 2000 Conference*, 2000, p. 50–50.

[6] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel, “Introducing the Open Trace Format (OTF),” in *Computational Science – ICCS 2006*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, vol. 3992, pp. 526–533.

[7] I. Andjelkovic and C. Artho, “Trace Server: A Tool for Storing, Querying and Analyzing Execution Traces,” in *JPF Workshop, Lawrence, USA*, 2011.

[8] G. Pothier and É. Tanter, “Back to the Future: Omniscient Debugging,” *Software, IEEE*, vol. 26, no. 6, p. 78–85, 2009.

[9] R. Borgeest and C. Rodel, “Trace Analysis with a Relational Database System,” in *Parallel and Distributed Processing, 1996. PDP’96. Proceedings of the Fourth Euromicro Workshop on*, 1996, p. 243–250.

[10] C. Prada-Rojas, M. Santana, S. De-Paoli, and X. Raynaud, “Summarizing Embedded Execution Traces through a Compact View,” in *Conference on System Software, SoC and Silicon Debug S4D*, 2010.

[11] R. A. Aydt, “The Pablo Self-Defining Data Format,” Departement of Computer, University of Illinois, Urbana, Illinois, Technical Report, 1992.

[12] L. M. Schnorr, B. de Oliveira Stein, and J. Chassin de Kergommeaux, “Paje Trace File Format, Version 1.2.5,” Laboratoire d’Informatique de Grenoble, France, Technical Report, Feb. 2013.

[13] “KPTrace Trace Format,” http://www.stlinux.com/.

[14] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, “The Vampir Performance Analysis Tool-Set,” in *Tools for High Performance Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 139–155.

[15] C. Prada-Rojas, F. Riss, X. Raynaud, S. D. Paoli, and M. Santana, “Observation Tools for Debugging and Performance Analysis of Embedded Linux Applications,” in *Conference on System Software, SoC and Silicon Debug - S4D 2009*, Sophia de Antipolis, France, Sep. 2009.

[16] “Score-P – HPC Profiling and Event Tracing Infrastructure,” http://www.vi-hps.org/projects/score-p.

[17] S. S. Shende, “The Tau Parallel Performance System,” *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, May 2006.

[18] “ARM development studio 5 (DS-5),” http://www.arm.com/products/tools/software-tools/ds-5/index.php.

[19] “Linux Tools Project/LTTng2/User Guide - Eclipsepedia,” http://wiki.eclipse.org/index.php/Linux_Tools_Project/LTTng2/User_Guide.

[20] J. Wilson, “Gantt Charts: A Centenary Appreciation,” *European Journal of Operational Research*, vol. 149, no. 2, p. 430–437, 2003.

[21] L. M. Schnorr, G. Huard, and P. O. A. Navaux, “A Hierarchical Aggregation Model to Achieve Visualization Scalability in the Analysis of Parallel Applications,” *Parallel Computing*, vol. 38, no. 3, pp. 91–110, Mar. 2012.

[22] R. Lamarche-Perrin, L. M. Schnorr, J.-M. Vincent, and Y. Demazeau, “Evaluating Trace Aggregation Through Entropy Measures for Optimal Performance Visualization of Large Distributed Systems,” Laboratoire d’Informatique de Grenoble, France, Research Report RR-LIG-037, 2012.

[23] S. Kullback and R. A. Leibler, “On Information and Sufficiency,” *The Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, Mar. 1951.

[24] C. E. Shannon, “A Mathematical Theory of Communication,” *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, no. 1, p. 3–55, 2001.