HAL
open science

Trace Management and Analysis for Embedded Systems

Generoso Pagano, Damien Dosimont, Guillaume Huard, Vania Marangozova-Martin, Jean-Marc Vincent

To cite this version:

Generoso Pagano, Damien Dosimont, Guillaume Huard, Vania Marangozova-Martin, Jean-Marc Vincent. Trace Management and Analysis for Embedded Systems. [Research Report] RR-8304, INRIA. 2013, pp.21. hal-00821907

HAL Id: hal-00821907

https://inria.hal.science/hal-00821907

Submitted on 13 May 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.

Trace Management and Analysis for Embedded Systems

Generoso Pagano , Damien Dosimont , Guillaume Huard , Vania Marangozova-Martin , Jean-Marc Vincent

RESEARCH REPORT
N° 8304
May 2013
Project-Teams MESCAL, MOAIS

ISSN 0249-6399 ISRN INRIA/RR--8304--FR+ENG

Trace Management and Analysis for Embedded Systems

Generoso Pagano *, Damien Dosimont *, Guillaume Huard †,
Vania Marangozova-Martin †, Jean-Marc Vincent †

Équipes-Projets MESCAL, MOAIS

Rapport de recherche n° 8304 — May 2013 — 21 pages

Résumé : La complexité croissante de l'architecture matérielle et logicielle des systèmes embarqués rend l'analyse de leur comportement difficile. Face à cela, l'utilisation de traces apparaît comme une solution incontournable afin de fournir des informations pertinentes sur l'exécution de programmes. Cependant, la gestion de ces traces et leur analyse posent un certain nombre de contraintes, liées notamment à la diversité des formats de traces, l'incompatibilité des méthodes d'analyse de traces, la taille volumineuse et le stockage de ces dernières, mais aussi au niveau du passage à l'échelle des techniques de visualisation employées pour les représenter. Dans ce papier, nous présentons FrameSoC, une nouvelle infrastructure de gestion de traces qui répond aux problèmes susmentionnés. FrameSoC fournit des solutions génériques pour le stockage des traces, propose des interfaces et permet l'intégration de différents outils d'analyse sous forme de plugin. Un module de visualisation, fournissant une représentation passant à l'échelle basée sur un algorithme d'agrégation, sera décrit pour illustrer les fonctionnalités de FrameSoC.

Mots-clés : Traces d'exécution, débogage, profilage, systèmes embarqués, multicœur, gestion de traces, infrastructure, modèle de données, formats de traces, SoC, visualisation, passage à l'échelle, agrégation, modularité, outils d'analyse

This research is supported by FUI [1]

* INRIA, first.last@inria.fr
† UJF, first.last@imag.fr

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

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

Trace Management and Analysis for Embedded Systems

Abstract: The growing complexity of embedded 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 management and analysis are hindered by several issues like the diversity of trace formats, the incompatibility of trace analysis methods, the problem of trace size and its storage as well as by the lack of visualization scalability. In this paper we present FrameSoC, a new trace management infrastructure that solves all the above issues together. It provides generic solutions for trace storage and defines interfaces and plugin mechanisms for integrating diverse analysis tools. We illustrate the benefit of FrameSoC with a case study of a visualization module that enables representation scalability of large traces by using an aggregation algorithm.

Key-words: Execution traces, debugging, profiling, embedded systems, multicore, trace management, infrastructure, data-model, database, trace formats, SoC, visualization, scalability, aggregation, modularity, analysis tools

Table of contents

1 Introduction **4**

2 Data-Model for Trace Management **5**
2.1 Different Trace Formats 5
2.2 Proposal of an Innovative Generic Data-Model 6

3 Trace Storage **7**
3.1 Existing Solutions for Storing Traces 7
3.2 Our Database Solution for Trace Storage and Management 8
3.3 Performance Measurements of Our Database Solution 8
3.3.1 Importing traces of various sizes into the system 8
3.3.2 Querying a given trace over different dimensions 9
3.3.3 Impact of trace size on request time 9

4 Analysis tools management **10**
4.1 Existing Solutions for Tool and Flow Management 10
4.2 Our Framework for Tool Management 10
4.3 FrameSoC Capabilities for the Analysis Workflow 11

5 Time slice aggregation visualization **12**
5.1 Visualization Scalability Issues 12
5.2 Time Slicing and Best Cut Partition Algorithm 13
5.3 Implementation in FrameSoC 14
5.4 Experimental conditions 14
5.5 Experiment analysis 15
5.5.1 Global trace representation 15
5.5.2 Analysis flow 15
5.5.3 Performances 16

6 Conclusion **17**

1 Introduction

Nowadays, embedded 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 [2]. In this context, application debugging and performance optimization become tremendously difficult tasks.

In this paper we focus on tracing and trace management, which address the above issues by gathering information about an embedded system execution and then reasoning about it. However, we do not tackle trace collection mechanisms, which should be ideally designed to minimize intrusivity, possibly by using hardware support [3]. In our work we consider the issues that need to be managed after the trace collection. Namely, we focus on four of them : the heterogeneity of trace formats, the storage of large traces, the management of the trace analysis flow and the visualization.

Heterogeneity of Trace Formats : There are multiple trace formats [4, 5, 6, 7]. In most cases, a trace format is closely related to a specific type of application or platform and they are designed together to fulfill specific needs. This approach tends to associate a format with a specific debugging framework. This prevents analysts from using external tools and does not help the diffusion of the techniques they implement. Furthermore, conversion between formats is not straightforward as different trace formats might use different semantics, the main risk being information loss during conversion.

Storage of Big Traces : Execution traces of embedded systems need to include low-level events such as CPU activity, interruptions, context switches, memory accesses, etc. A trace collected even for a very short execution may contain a large quantity of information, which translates into a large data volume (from MB to GB) [8]. As trace analysis may consider random parts of a trace, an efficient management of trace storage is mandatory.

Trace Analysis Flow : Different treatments are often needed to understand traces. Statistics provide general information about application behavior, while pattern recognition [9] or data mining algorithms [10] extract information and synthesize the trace representation. Besides, filters or noise elimination processes help to reduce the amount of information. Trace analysis can be performed by applying such treatments on raw data, or within a flow where the result of one computation is reused as an input of another (for instance, one to filter the trace, another one to process it, and a last one to vizualize the result). 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 have provided a straightforward compatibility.

Visualization Scalability : An embedded system execution trace usually contains too much information to be entirely represented on the screen. This could result in cluttered drawings, non-exact proportions or uncontrolled visual aggregation [11]. Furthermore, different information needs to be represented differently. For example, time views are dependent of information granularity and execution duration, while structural representations depend on the number of entities. To solve these issues, analysts need synthetic representation of traces, where information loss is controlled and quantified, but still enables the detection of hot spots [12].

Our main contribution is a new trace management and analysis framework, FrameSoC (Figure 1), designed in the context of SoC-Trace Project [13] to answer to the above four issues. Regarding trace management, we tackle the problem of trace format heterogeneity with a generic data-model and an associated data access interface (Section 2), while we manage large traces by providing a database storage solution (Section 3). Regarding the analysis complexity issue, which represents our main challenge, we propose facilities to enable analysis flows, by expressing

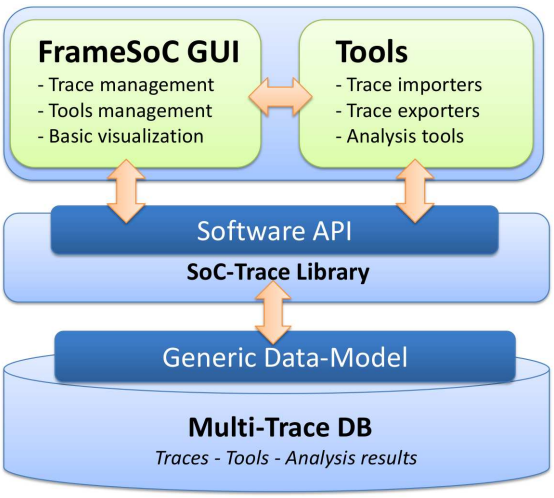

FIG. 1 – The FrameSoC infrastructure

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 (Section 4), using a generic interface. We corroborate this claim by presenting a scalable visualization module that takes advantage of FrameSoC (Section 5). This module is based on an aggregation algorithm [14] and answers to screen-limitation and context-loss issues by proposing a synthetic visualization of the whole trace, that outlines its global behavior. Moreover, we even can reuse other tool results to improve on this aggregation. Each section presents the relevant related works, our proposal, our implementation and some results. We will conclude and present future works in Section 6.

2 Data-Model for Trace Management

2.1 Different Trace Formats

Trace formats heterogeneity is mainly caused by the need to tailor the stored information to specific application domain requirements. For instance, in the context of MPI/OpenMP applications [15], the considered events cover MPI communications or OpenMP fork and join operations. Instead, in the context of embedded system [16] different event semantics, like context switches or interruptions, are more relevant. Furthermore, even in presence of the same needs, distinct communities tend to develop custom formats ([17, 18]) to leverage specific analysis tools, which is another cause of formats diversity.

Looking at various formats from a data-modeling point of view, we can separate them in two main categories : on one hand, static formats (e.g. Alog [19] and KPTrace [4]), which have predefined records and associated semantics, on the other hand, self-defining formats (e.g. SDDF [20] and Pajé [7]), which contain metadata describing the format of the trace itself, being therefore more flexible. To our knowledge, none of the existing trace formats enables to store analysis results in addition to raw trace data.

2.2 Proposal of an Innovative Generic Data-Model

To tackle the problem of format heterogeneity, we propose the use of an innovative generic data-model for traces (Figure 2), presented in details in [13]. The model is intrinsically self-defining and includes the representation of trace metadata, trace raw data and analysis results, with related tools metadata.

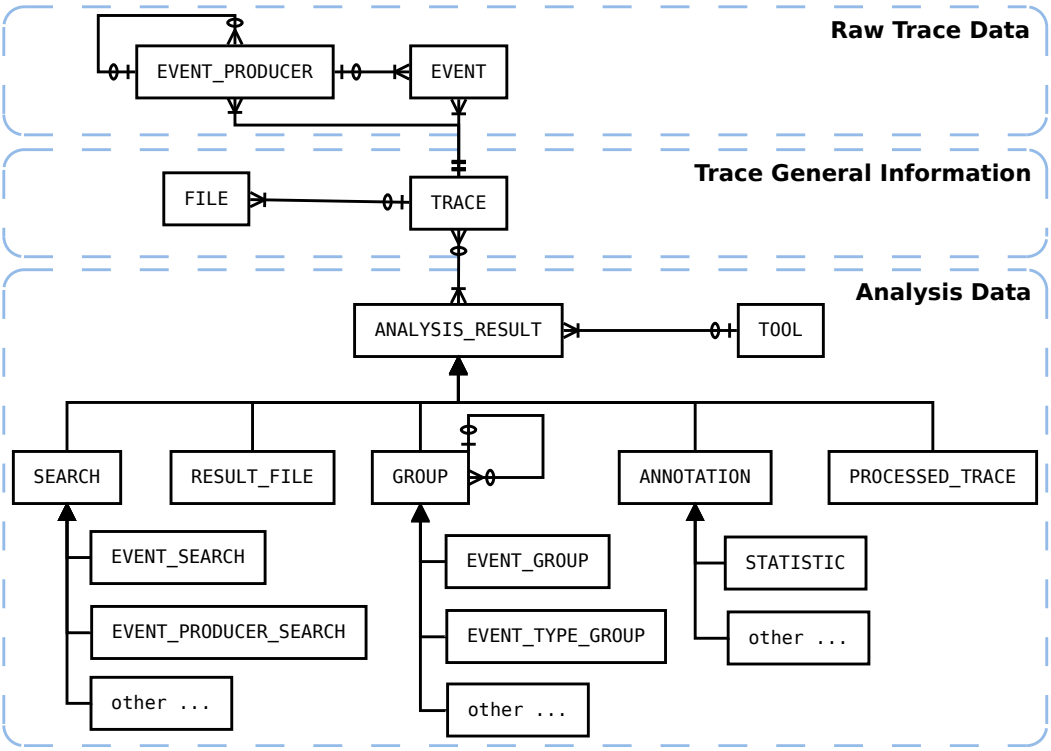

FIG. 2 – Generic data-model for trace management (Crow's Foot notation)

The central entity of the model is the trace, which has metadata and can be related to files (e.g. configuration files, platform description). 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). The model provides some predefined but extensible types of analysis results : 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. The interest of storing analysis results is double. First, it saves time consuming recomputations during future consultations and reuses. Second, defining a standard model for analysis results makes the collaboration among different analysis tools possible, thus offering the possibility to build a rich analysis flow.

The *EVENT* and *TRACE* entities are actually modeled using a self-defining pattern, detailed in Figure 3 for events. An event (`Event` class) has a type (`EventType` class) which is described by a given set of parameter types (`EventParamType` class). The values of these parameters for a given event are stored in the `EventParam` class. So the right part of this representation describes the *type* of an event (type and related parameter types), while the left part contains the *values*

(the event predefined attributes and parameter values). The same pattern applies to traces. Using this self-defining approach we obtain a generic trace representation, which has minimal associated semantic and is therefore suitable for representing any kind of trace format without information loss. At this time, we managed to represent with our model KPTrace [4], Pajé [7] and Tima [21] formats, by simply plugging specific importers to the infrastructure (Section 4).

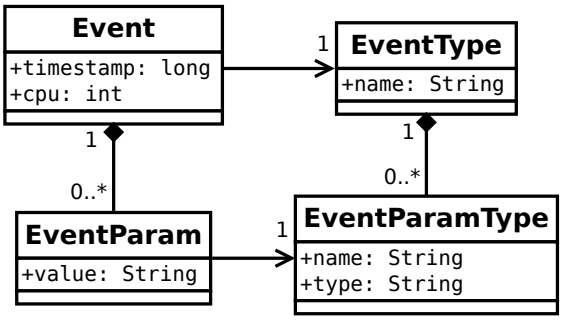

FIG. 3 – UML diagram of the self-defining pattern (*EVENT* entity)

To interact with this data-model, which is physically stored in a database (Section 3), we provide a database-independent Java API to write and retrieve the elements of the model. The software library providing this API (SoC-Trace Library) is composed by three major parts : an object-oriented representation of the data-model, a write interface to store the objects of the model and an SQL-agnostic query interface. This software library, facilitating the access to the data-model, answers to the need of factorization of functionalities for analysis tools, while taking care of proper data management.

3 Trace Storage

3.1 Existing Solutions for Storing Traces

Traditionally, raw trace data are stored in plain files (event logs) : this approach has the advantage of simplicity and can achieve good storage performance with binary formats, but it has not special support for optimized random trace accesses or basic processing, like filtering. Thus, the analysis requires to load the whole file into main memory [22].

Other approaches propose the use of a structured trace file, more suitable for specific kinds of access, like a frame-based file format [23] for fast time-guided navigation, or even more structured file formats [6] where the access is optimized in both time and space (processes) dimensions. These approaches facilitate 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 faces scalability issues while keeping the greatest flexibility for data-access. In [24] authors use a relational database to store execution traces, providing a system configurable in terms of trace formats and of metrics defined : for each trace there is a separate database, with a possibly different schema, while a central database stores trace metadata. In [25] authors store execution traces having a fixed format in a graph database, providing a database-independent storing and querying interface. In [26] a database is used to store relevant data during the execution in order to enable an offline debugging, able to go forwards and backwards in time. To face scalability issues this database architecture is distributed. The trace management framework proposed in [17] uses a database

to store embedded system execution traces. For each trace a different database is used and different trace formats can be supported. In [27] we find another solution for trace analysis using a database, but only for storing trace profiles, while the raw trace data are still kept on files.

3.2 Our Database Solution for Trace Storage and Management

Given the richness of our data-model, the role of the database is central in our solution. In fact, 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 interesting traces for a multi-trace analysis) or the storage of generic results, and analysis tools are not taken into account at all.

There are several pragmatic motivations that lead us to the use of a database. First of all, thanks to accurate modeling and normalization it is possible to store information with minimal redundancy. Furthermore, a database let us separate the logical data-model from the physical representation of data. Then, we can easily access to parts of the trace or perform noise filtering using trivial querying. Search operations can be optimized by defining related indexes : this mechanism is flexible and not limited to time or space dimensions. Finally, complex computations over trace data can be performed in the database, instead of loading the whole trace in memory and perform such computations at the application level.

In order to be independent from a given DBMS technology, our infrastructure is designed to be able to work with different DBMS, 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; all trace databases are coordinated using a central system database, with a resulting distributed architecture. The relational schema of both system and trace databases is given in [13]. When considering storage scalability issues, none of these two supported DBMS limits the number of databases managed. Considering 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.

3.3 Performance Measurements of Our Database Solution

To show that the proposed database solution is effective when analyzing data over several dimensions, we present in this section some performance results. The experiments are implemented using our SoC-Trace Library in combination with 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.

3.3.1 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 time needed to perform the import operation, with and without indexing. Import time (Figure 4) 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%.

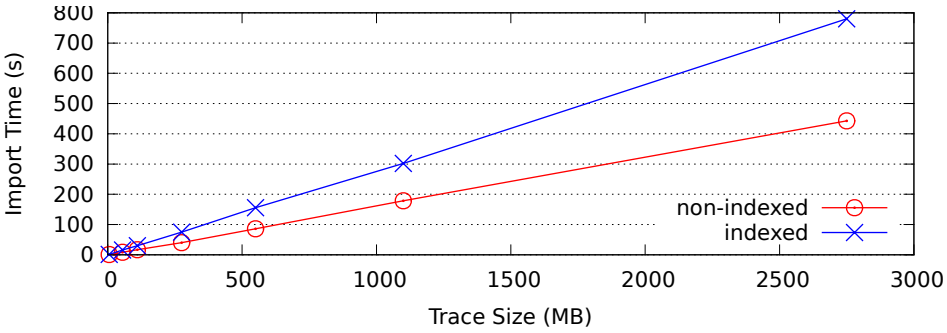

FIG. 4 – Import time for traces of various sizes, with and without indexing.

3.3.2 Querying a given trace over different dimensions

A great advantage brought by the use of 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 as 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 5) 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.

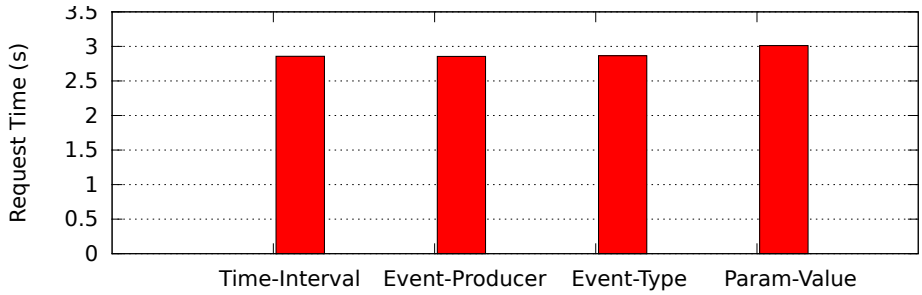

FIG. 5 – Time to retrieve 20000 events from a trace of 2 millions events, using various dimensions for filtering.

3.3.3 Impact of trace size on request time

One of the interests of putting huge trace data in the database is information retrieval, limiting the impact 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 6). 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.

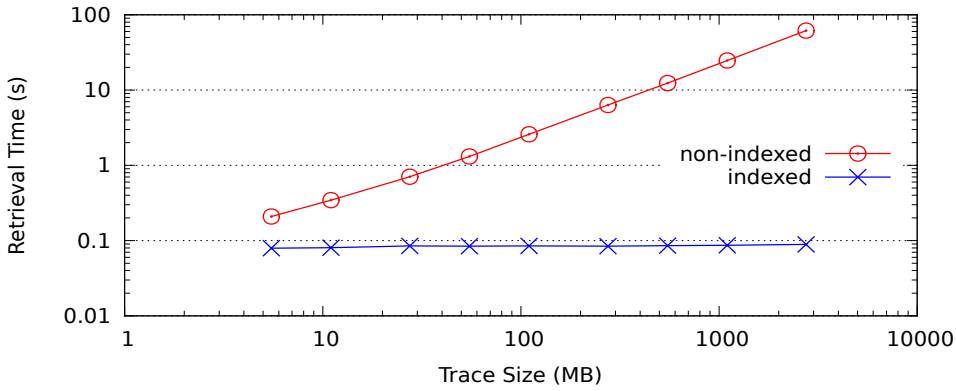

FIG. 6 – Time to retrieve 10000 events from traces of various sizes, using indexed or non-indexed databases.

4 Analysis tools management

4.1 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 [28], [16].

In the field of parallel-systems, different solutions have been proposed to address this problem. The visualization tool Pajé [22] 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 [29] 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 phylosophy, Tau [30] 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, such as [31], offer a closed set of functionalities tailored to specific hardware. Even open source solutions like [32] do not easily enable the plugging of new tools and do not support tool interaction through a shared data-model for analysis results.

4.2 Our Framework for Tool Management

With FrameSoC, we propose a framework for trace analysis in which a strong attention is given to tool management and support for tool cooperation. Our trace management infrastructure, based on the Eclipse platform[1], has been designed to be easily extensible and suitable for building rich analysis flows.

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

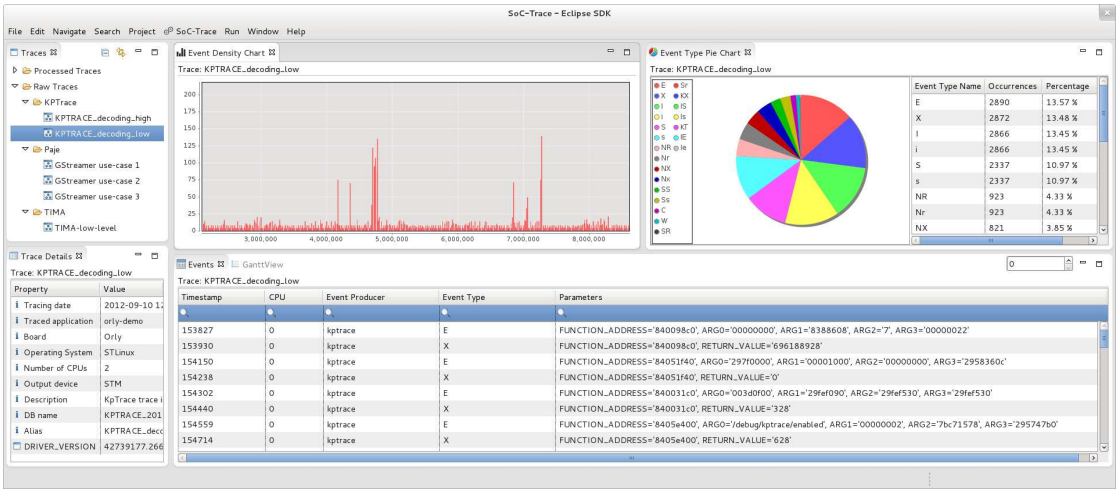

FIG. 7 – FrameSoC GUI based on Eclipse. On the left : a trace explorer, along with a related detailed view. On the right : an event density chart, a statistics pie-chart and a searchable table of events.

The integration of tools is made possible by three elements of our infrastructure. First, the generic data-model for trace data (Section 2), lets different tools work on the same trace. Second, the well defined but flexible data-model for trace analysis result (Section 2), enables the creation of analysis flows where the output of a tool is taken as input by the following tool. The support provided by the infrastructure is semantically agnostic and all the analysis logic remains within the tools. Third, a specific and explicit support for pluggability of new modules is provided. A preferred way to add a tool to FrameSoC is to provide an Eclipse plugin that uses the interface we defined through the extension-point mechanism. However our infrastructure also supports the possibility to integrate external back-box tools. In both solutions, tools deal with the same data-model for trace and results storage and are launched using the same interface.

The prototype implementation of FrameSoC itself provides some *framework* tools, to enable basic trace analysis (Figure 7) : 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 and Tima formats) and to export into Pajé format. As for analysis tools, we integrated a tool able to perform simple sequence-search with result saving (see the following subsection), 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 able to perform aggregation (Section 5).

4.3 FrameSoC Capabilities for the Analysis Workflow

Figure 8 shows an example of trace analysis workflow involving the simple sequence-search tool we developed. The raw trace is imported in the database according to the data-model entities. The analysis tool retrieve trace events, looking for sequences identified by a pattern of event types (A and B in the example) configurable by the user. All pattern instances and exceptions (events of above types outside the pattern) are saved in the database as a hierarchy of groups, using the

GROUP entity defined by the generic data-model. The result, saved with a defined format, can be easily retrieved by other tools for further analysis : for example it could be useful to visualize the time distribution of the pattern or deeply investigate the trace sections where exceptions to the pattern occur.

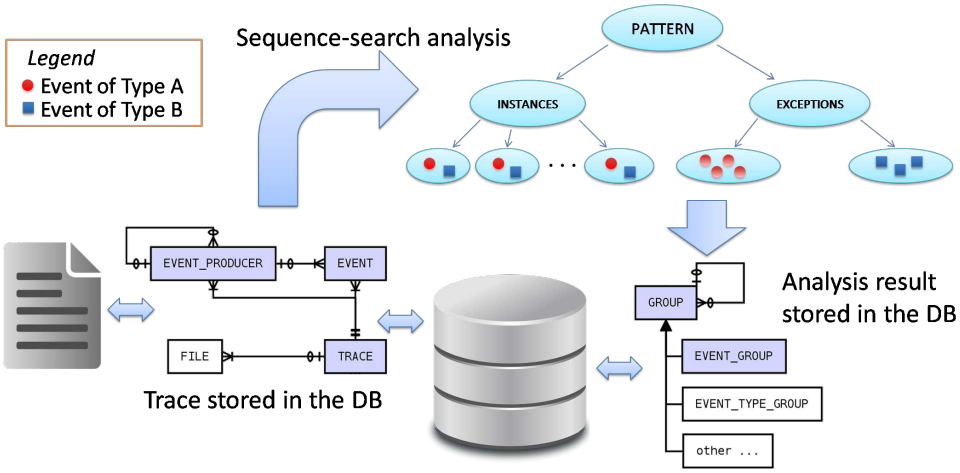

FIG. 8 – Example of analysis flow involving a sequence-search analysis tool

5 Time slice aggregation visualization

We propose a FrameSoC visualization module which aims at answering both time and space scalability issues. It provides a synthetic view of a trace, cut into time slices, and represented as a sequence of representative elements. This sequence is constructed using an aggregation algorithms which identifies consecutive parts of the trace that have a similar behavior.

5.1 Visualization Scalability Issues

Scalability issues are common to parallel and embedded system visualization tools. The Gantt chart representation [33], for example, is classically used to visualize application behavior over time thanks to its ability to represent causality relations. However, because of the quantity of information visualized (due to events granularity, platform heterogeneity or execution duration), an analyst may be forced to zoom out or to pane and, thus, lose either the execution context or representation fidelity.

A partial solution to this problem is proposed by Pajé [22], a parallel analysis tool, and LTTng Eclipse Viewer [32], a visualization tool associated with Linux LTTng tracer [18]. Both tools highlight events which 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. A different approach is proposed in KPtrace [16] with its time compression within the T-Charts view. The idea is to compress trace parts containing the same events. The compression depends on the scale and the display size. Designed with the goal of avoiding context loss, this technique can possibly introduce misinterpretation, as the width of the compressed time parts is not proportional to the duration of the original parts. ARM

DS-5 Streamline [31] represents state density by cutting the trace in time slices and using color gradients whose intensity is related to the state value amplitude. This technique shows hot spots while managing efficiently vertical space. However, it induces a loss of amplitude perception compared to bar charts.

A major issue in visualization is 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 [16], the hierarchy associated with a given core can be collapsed and represented as part of the root of the hierarchy. In the Vampir [28] task profile view, event producers are clustered using a proximity metrics like functions duration. This representation, however, is mainly used for profiling as it fails in showing causality relations. Triva [34] treemap views use multiple axes for hierarchies representation and show the evolution of the execution over time using animations. This visualization highlights network bottlenecks and unbalanced workload but is not suited to identify problems related to synchronization (deadlock) or scheduling.

5.2 Time Slicing and Best Cut Partition Algorithm

In this section, we adapt an existing Best Cut Partition algorithm [14] by using time slicing. To use the algorithm, the whole trace or a part of the trace is represented as a one-dimensional array, where each element is associated to a trace time slice. The idea then is to aggregate consecutive array elements that contain close values. The major aggregation issue is the trade-off between simplification (group in order to ignore small differences) and information loss (still keep track of the relevant parts). The goal is to reveal important behavioral changes in the aggregated representation and thus help the identification of hot spots.

In order to apply the Best-Cut Partition algorithm, we perform a time slicing in the following way. 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 ratio of associated event producers.

The Best-Cut Partition algorithm computes a quality measure 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. The quality measures characterize the aggregation gain (or complexity reduction) and the information loss and are inherited from information theory. Aggregation gain (1) is calculated using Shannon entropy [35], while information loss (2) is computed with Kullback-Leibler divergence [36].

$$gain(A) = (v(A)\log_2 v(A)) - \sum_{e \in A}(v(e)\log_2 v(e)) \tag{1}$$

$$loss(A) = \sum_{e \in A} v(e) \times \log_2\left(\frac{v(e)}{v(A)} \times |A|\right) \tag{2}$$

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 $quality(A)$ corresponds to $gain(A)$ or $loss(A)$:

$$quality(A) = \sum_{i \in n} quality(A[i]) \tag{3}$$

TAB. 1 – 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 p	**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

In order to compute the Best Cut Partition, we require a parameter p. It is used to compute a *parametrized Information Criterion* for each aggregation $\mathcal{A}$.

$$pic(\mathcal{A}) = p \times gain(\mathcal{A}) - (1 - p) \times loss(\mathcal{A}) \tag{4}$$

Aggregated parts are chosen by selecting those with the highest pIC value. 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. List of relevant values of p are computed using a search by bissection, that finds successive parameters that give a different configuration. An example is shown in Table 1.

5.3 Implementation in FrameSoC

We implement the Best Cut Algorithm in C++. Our vector 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 acquiring parameters and best cuts from the shared library. The user interface part provides interaction mechanisms to set or select the different parameters for queries and computation. The result is visualized in a frame representing the trace as a one-dimensional array. The parts are labeled by numbers and emphasized by colors, which are identical for aggregated parts.

5.4 Experimental conditions

In our experiment, use cases are based on a basic open-source G-Streamer video application[2], displaying a mpeg video. We generate traces associated to several executions by using the `GST_DEBUG` option. For each execution, we introduce an anomaly by using the *stress* tool[3] in order to perturb the video streaming. The traces are converted into Pajé trace format with the help of python scripts and Poti library[4], and 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.

[2] `https://code.google.com/p/gst-player/`
[3] `http://weather.ou.edu/~apw/projects/stress/`
[4] `https://github.com/schnorr/poti`

TAB. 2 – G-Streamer application execution contexts

Use Case	Perturbation : *stress* settings	Streaming behavior	Tracing duration	Trace size	Event producers number	Events number
1	Not activated	Normal	10 s	112.5 MB	1523	1420535
2	After 2 s, 4 i/o, 4 mem, during 2 s	Freeze at 2 s, during 2 s	10 s	115.5 MB	1499	1404817
3	After 3 s, 4 i/o, 4 mem, during 2 s	Freeze at 3 s, during 2 s	11 s	119.5 MB	1496	1517717
4	After 3 s, 4 i/o, 1 mem, during 1 s	Light disruption at 3 s	11 s	121.8 MB	1496	1545959

The table 2 summarizes *stress* configuration, generated trace values and effects of *stress* on the video quality.

5.5 Experiment analysis

5.5.1 Global trace representation

Here, we represent the traces by using our module, with the aim of showing that our visualization is coherent with our experiment, in other words matches perturbation timestamps. We perform aggregation on the whole trace cut in 20 time slices. We start our analysis with a fully aggregated trace of the non perturbed case. We progressively discover different parts by disaggregating our representation, leading to a representation that highlights a transitory state at 2 s, followed by a steady state of 7.5 s (Figure 10).

By taking into account the application behavior, we deduce that its execution is made of an initialization followed by a constant behavior. Performing the same treatment on case 3 results in a representation with an initialization phase of 2.8 s (5 first parts), followed by 2 s during which all states are disaggregated and a final steady state (Figure 11). These timestamps match the application perturbation settings and visible behavior (3 s after the start we have 2 s of stress). The case 2 exhibits a similar behavior and is not reported here.

In case 4, we get a slight disruption at 3 s followed by a 2 s long steady state, to finish with another 5 s long steady state (Figure 12). We can guess that the perturbation is followed by a transitory recovery state before returning to a normal behavior. All these results confirm that our aggregation tool is able to highlight behavioral changes in the trace.

5.5.2 Analysis flow

In this section, we perform a focused analysis by filtering relevant event producers in the trace, after having found which of them are not related to application behavior. We are helped by statistics, filtering, and results management facilities provided by FrameSoC. In Section 5.5.1, the aggregation is performed on the whole trace. In each case, an initialization phase of about 2 s is outlined by the visualization. The pie-chart statistics view indicates that event producers have an event occurrence number ranging from 1 to 100000, which suggests that they do not have an equivalent importance in application behavior. Using the filter described in Section 4.2, we are able to distinguish event producers being active only during initialization phase and those

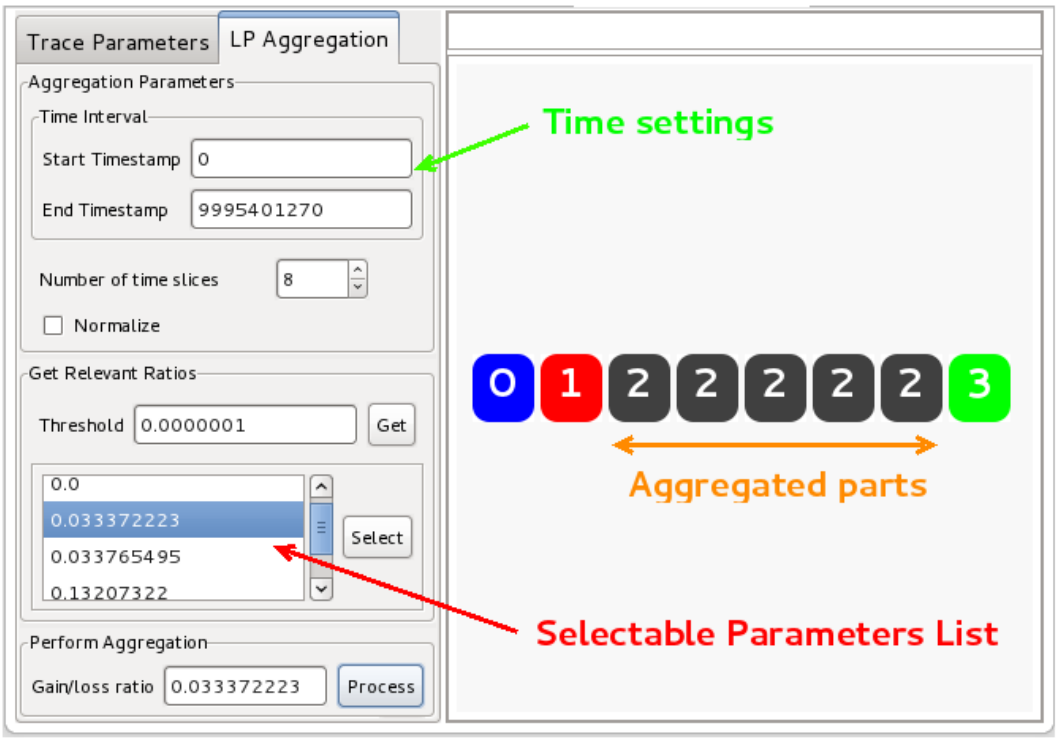

FIG. 9 – User interface provided by the visualization module

FIG. 10 – Use case 1 : initialization sequence at slices 0-3 followed by a steady state part (4)

active in the whole trace. These subsets of producers are saved in the trace database, by using FrameSoC results management capabilities, and retrieved by our aggregation tool. In case 3, for instance, only 20% of event producers are active after the first 1.5 s of execution. Performing the aggregation on them gives a quite similar result than the one in Figure 11, excepted that the two first initialization parts are aggregated. Indeed, in this experiment, the point of interest is not located in the initialization phase. In this case, using filtering at the beginning of the analysis flow helps separating unrelated hot spots and decreases visualization complexity.

5.5.3 Performances

Getting all the events from the database takes between 15 s and 20 s depending on the specific case, while vector computation takes 50 s for about 1.5 million events. In our implementation, these values are only computed once for a given time slices number. Computing quality metrics takes 20 ms while getting the relevant parameters p and print parts related to a parameter takes less than 2 ms. These values are acceptable for an analysis, especially on such a large trace.

FIG. 11 – Use case 3 : disruption at slices 3-6

FIG. 12 – Use case 4 : disruption at slices 3-4, the aggregation highlights the following transitory state (5) and steady state (6)

6 Conclusion

This paper presents FrameSoC, a trace management infrastructure for embedded systems. It addresses the problems of trace formats heterogeneity, large traces management, trace analysis flow setup and scalable visualization.

To tackle the problem of format heterogeneity we proposed a generic data-model able to represent not only raw data but also meta information about the trace and analysis results. The expressiveness of our model has been validated by experiments with several real trace formats. For the future, we foresee refinements to our data-model induced by new formats conversions. We are also interested in the introduction and the efficient implementation of new types of analysis results, supporting, for example, multi-trace analysis.

FrameSoC manages large traces by storing them in a relational database. This choice enables filtering and searching in various dimensions, while keeping reasonable read and write performance. The framework supports several DBMS in order to be independent from a specific technology. Access to data being crucial for analysis tools, our future research will consider specific use case requests optimization and possible alternative storage solutions, such as temporal or non-relational databases.

FrameSoC puts a strong emphasis on tools management and interoperability. Of course, our shared data-model is a basic block for the creation of analysis flows in which several tools can take part. But an explicit support has also been given to tools pluggability and it 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 thanks to the efforts invested in the SoC-Trace project. An other interesting perspective is to provide to the final user a convenient interface in order to define analysis chains.

We have illustrated the features of FrameSoC with a trace aggregation visualization module. From the framework point of view, the experience has been successful as the implementation of the aggregation algorithm has been seamless. From the visualization point of view, the module succeeds in representing an application global behavior. It has been applied on simple software streaming applications, in which we have introduced some perturbations. Compared to traditional Gantt charts, this view has the ability to coarsely describe the whole trace behavior over space and time and highlight disruptions, while managing large amount of events, keeping context and representation fidelity. It could be interesting to see how a behavior disruption is propagated across different groups of the producers hierarchy by simultaneously showing several arrays associated to these subsets. Our future works will also focus on the enhancement of the information displayed along with the aggregated view such as used metrics and cuts. We will also apply this aggregation algorithm to other kinds of application such as parallel applications.

Indeed, an interesting perspective for FrameSoC is to take even more advantage of the tracing algorithms and tools developed in the high performance computing domain. Each day, MPSoC architectures become closer to HPC one and give rise to the same problems.

References

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

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

[3] Aleksandar Milenkovic, Vladimir Uzelac, Milena Milenkovic, and Martin Burtscher. Caches and Predictors for Real-Time, Unobtrusive, and Cost-Effective Program Tracing in Embedded Systems. *IEEE Transactions on Computers*, 60(7) :992–1005, 2011.

[4] KPTrace Trace Format. `http://www.stlinux.com/`.

[5] M. Desnoyers. *Low-Impact Operating System Tracing.* PhD computer science, Université de Montreal, 2009.

[6] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Introducing the Open Trace Format (OTF). In *Computational Science – ICCS 2006*, volume 3992, pages 526–533. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[7] Lucas Mello Schnorr, Benhur de Oliveira Stein, and Jacques Chassin de Kergommeaux. Paje Trace File Format, Version 1.2.5. Technical report, Laboratoire d'Informatique de Grenoble, France, February 2013.

[8] Felix Wolf, Felix Freitag, Bernd Mohr, Shirley Moore, and Brian JN Wylie. Large Event Traces in Parallel Performance Analysis. *Architecture of Computing Systems, ARCS 2006 (19a. Frankfurt-Main, Alemanya)*, 2006.

[9] Patricia López Cueva, Aurélie Bertaux, Alexandre Termier, Jean François Méhaut, and Miguel Santana. Debugging Embedded Multimedia Application Traces Through Periodic Pattern Mining. page 13. ACM Press, 2012.

[10] C. Kamdem Kengne, L.C. Fopa, N. Ibrahim, A. Termier, M.C. Rousset, and T. Washio. Enhancing the Analysis of Large Multimedia Applications Execution Traces with FrameMiner. pages 595–602. IEEE, December 2012.

[11] Lucas Mello Schnorr, Arnaud Legrand, and Jean-Marc Vincent. Detection and Analysis of Resource Usage Anomalies in Large Distributed Systems Through Multi-Scale Visualization. *Concurrency and Computation : Practice and Experience*, 24(15) :1792–1816, 2012.

[12] B. Shneiderman. The Eyes Have It : A Task by Data Type Taxonomy for Information Visualizations. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, page 336–343, 1996.

[13] Generoso Pagano and Vania Marangonzova-Martin. SoC-Trace Infrastructure. Technical Report RT-0427, INRIA, November 2012.

[14] Robin Lamarche-Perrin, Lucas Mello Schnorr, Jean-Marc Vincent, and Yves Demazeau. Evaluating Trace Aggregation Through Entropy Measures for Optimal Performance Visualization of Large Distributed Systems. Research Report RR-LIG-037, Laboratoire d'Informatique de Grenoble, France, 2012.

[15] Felix Wolf and Bernd Mohr. Automatic Performance Analysis of Hybrid MPI/OpenMP Applications. *Journal of Systems Architecture*, 49(10-11) :421–439, November 2003.

[16] Carlos Prada-Rojas, Frederic Riss, Xavier Raynaud, Serge De Paoli, and Miguel 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, September 2009.

[17] 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.

[18] Pierre-Marc Fournier, Mathieu Desnoyers, and Michel R. Dagenais. Combined Tracing of the Kernel and Applications with LTTng. In *Proceedings of the 2009 Linux Symposium*, 2009.

[19] Ed Karrels and Ewing Lusk. Performance Analysis of MPI Programs. In *Workshop on Environments and Tools for Parallel Scientific Computing*, 1994.

[20] Ruth A. Aydt. The Pablo Self-Defining Data Format. Technical report, Departement of Computer, University of Illinois, Urbana, Illinois, 1992.

[21] Damien Hedde and Frederic Petrot. A Non Intrusive Simulation-Based Trace System to Analyse Multiprocessor Systems-on-Chip Software. pages 106–112. IEEE, May 2011.

[22] J Chassin de Kergommeaux. Pajé, an Interactive Visualization Tool for Tuning Multi-Threaded Parallel Applications. *Parallel Computing*, 26(10) :1253–1274, August 2000.

[23] C. Eric Wu, Anthony Bolmarcich, Marc Snir, David Wootton, Farid Parpia, Anthony Chan, Ewing Lusk, and William Gropp. From Trace Generation to Visualization : A Performance Framework for Distributed Parallel Systems. In *Supercomputing, ACM/IEEE 2000 Conference*, page 50–50, 2000.

[24] 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.

[25] Igor Andjelkovic and Cyrille Artho. Trace Server : A Tool for Storing, Querying and Analyzing Execution Traces. In *JPF Workshop, Lawrence, USA*, 2011.

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

[27] K.A. Huck, A.D. Malony, R. Bell, and A. Morris. Design and Implementation of a Parallel Performance Data Management Framework. pages 473–482. IEEE.

[28] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. The Vampir Performance Analysis Tool-Set. In *Tools for High Performance Computing*, pages 139–155. Springer Berlin Heidelberg, Berlin, Heidelberg.

[29] Score-P – HPC Profiling and Event Tracing Infrastructure. `http://www.vi-hps.org/projects/score-p`.

[30] S. S. Shende. The Tau Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2) :287–311, May 2006.

[31] ARM development studio 5 (DS-5). `http://www.arm.com/products/tools/software-tools/ds-5/index.php`.

[32] Linux Tools Project/LTTng2/User Guide - Eclipsepedia. `http://wiki.eclipse.org/index.php/Linux_Tools_Project/LTTng2/User_Guide`.

[33] J.M. Wilson. Gantt Charts : A Centenary Appreciation. *European Journal of Operational Research*, 149(2) :430–437, 2003.

[34] Lucas Mello Schnorr, Guillaume Huard, and Philippe Olivier Alexandre Navaux. A Hierarchical Aggregation Model to Achieve Visualization Scalability in the Analysis of Parallel Applications. *Parallel Computing*, 38(3) :91–110, March 2012.

[35] Claude Elwood Shannon. A Mathematical Theory of Communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1) :3–55, 2001.

[36] S. Kullback and R. A. Leibler. On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1) :79–86, March 1951.

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 0249-6399