2013 IEEE 7th International Symposium on Embedded Multicore/Manycore System-on-Chip

Trace Management and Analysis for Embedded Systems

Generoso Pagano, Damien Dosimont
INRIA, first.last@inria.fr

Guillaume Huard, Vania Marangozova-Martin, Jean-Marc Vincent
UJF, first.last@imag.fr

***Abstract*—The growing complexity of embedded systems 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 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 article we present FrameSoC, a new trace management infrastructure that solves these issues. It proposes 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 provides representation scalability for large traces by using an aggregation algorithm.**

I. Introduction

In our work we focus on the following main issues concerning embedded system trace management and analysis.

Heterogeneity of Trace Formats: As a trace format is typically designed for the needs of a specific type of application or platform, trace formats are numerous ([1], [2], [3]). However, different formats work with different debugging frameworks. This prevents analysts from using external tools and does not help the diffusion of the techniques they implement.

Storage of Big Traces: Execution traces of embedded systems include low-level events such as interruptions, context switches, memory accesses and so on. Even a trace collected during a very short execution time may result in a large data trace (from MB to GB) [4]. As trace analysis may consider random parts of a trace, an efficient trace storage is mandatory.

Trace Analysis Flow: Trace analysis can be performed using different treatments (e.g., pattern recognition [5] or data mining [6]), which may be applied to raw data, or to the result of a trace analysis. However, chaining such trace analysis treatments in a flow is a challenge because of the variety of analysis techniques and tools and their respective data outputs.

Visualization Scalability: An execution trace usually contains too much information to be entirely represented on a single screen. As a result, some representations use non-exact proportions or uncontrolled visual aggregation [7]. To detect hot spots, analysts need synthetic representation of traces in which information loss is controlled and quantified [8].

Our main contribution is a new trace management and analysis framework, FrameSoC, designed in the SoC-Trace project [9] context to address the four issues above. For managing heterogeneous trace formats, we define a generic data-model and an associated data access interface (Section II). We manage large traces by providing a database storage solution (Section III). We help analysis flows by expressing and storing analysis results using a common format. We define a generic interface for plugging to the infrastructure various analysis tools including statistics modules, filters, data mining engines and visualizations (Section IV). We illustrate this feature with a scalable visualization module which tackles the screen-limitation and context-loss issues by providing a synthetic visualization of the whole trace behavior (Section V). Each section presents the relevant related works and our proposals. We conclude and present perspectives in Section VI.

II. Generic Data-Model for Trace Management

A. Different Trace Formats

Trace format heterogeneity is mainly caused by the need to tailor the stored information to specific application domain requirements and concepts. Furthermore, even if there are the same needs, distinct communities tend to develop custom formats ([10], [11]) to promote specific analysis tools. Looking at various formats from a data-modeling point of view, we can separate them in two main categories: static formats (e.g., KPTrace [1]), which have predefined records and associated semantics, and self-defining formats (e.g., Pajé [3]), which contain metadata describing the format of the trace itself. To our knowledge, no existing trace format enables the storage of analysis results in addition to raw trace data.

B. Proposal of an Innovative Generic Data-Model

To solve the problem of format heterogeneity, we propose the use of an innovative generic data-model for traces (Figure 1), presented in details in our technical report [9]. The model is intrinsically self-defining and includes the representation of trace metadata, trace raw data, trace analysis results and related tool metadata.

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 produced by an active entity. Event producers can be organized in a hierarchy, reflecting, for instance, the execution hierarchy in the traced application (e.g., processes/threads). The innovative point of the model is the fact that it embeds 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 benefit of storing analysis

978-0-7695-5086-2/13 $26.00 © 2013 IEEE
DOI 10.1109/MCSoC.2013.28

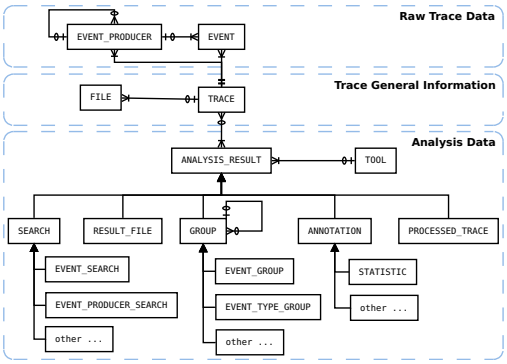

Fig. 1. Generic data-model for trace management (Crow's Foot notation)

results is twofold. First, results can be obtained and reused without time consuming recomputations. Second, defining a standard model for analysis results facilitates the collaboration among different analysis tools. It is thus possible to build a rich analysis flow. The *event* and *trace* entities are modeled using a self-defining pattern [9]. Using this approach we obtain a generic trace representation: with minimal semantics and suitable for representing any kind of trace format without information loss. We have managed to represent with our model KPTrace [1], Pajé [3] and Tima [12] formats.

To interact with the physical representation of the data-model, stored in a database (Section III), we provide a database-independent Java API. The software library providing this API factorizes the functionalities needed by analysis tools, while taking care of proper data management.

III. Efficient Trace Storage

A. Existing Solutions for Storing Traces

Traditionally, raw trace data are stored in plain files, with no special support for optimized random accesses or filtering. As a consequence, the analysis requires to load the whole file into the main memory [13]. Other approaches propose the use of a structured trace file. A frame-based file format [14], for example, allows fast time-guided navigation. Another structured format [2] optimizes accesses in both time and space (processes) dimensions. These approaches optimize trace accesses in a fixed and limited number of dimensions. An alternative approach for storing traces is the use of a database (e.g., [10], [15], [16]). Databases ensure scalability while keeping great flexibility for data-access.

B. 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. Indeed, we use the database to manage multiple traces, organize the trace analysis tools and also store the produced results. None of the existing database solutions considers multi-trace requests or the generic storage of results, and analysis tools are not taken into account at all.

Our infrastructure is independent from a given database management system (DBMS). It supports different DBMS via simple adaptation modules. With the aim of providing a simple and scalable solution, we store each trace in a different database (in the sense of the unit of organization in a DBMS). All *trace* databases are coordinated using a central *system* database. The relational schema of both *system* and *trace* databases is given in our technical report [9].

To show that our database solution is effective in analyzing data along several dimensions, we carried out a performance analysis, described in details in our related research report [17]. We showed that the time to import a trace into our database grows linearly with the trace size, keeping reasonable values even for large traces (about 7.5 minutes for a 2.75 GB trace). As for data access, we showed that database indexing enables for near constant retrieval time for result sets of the same size, even for traces of different size. Finally, we observed that the time needed to filter trace events is not deeply influenced by the filtering dimension (the producer, the type, the value of a parameter or the time interval). 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. Further experiments and comparisons with existing trace storage systems are among our future works.

IV. Analysis Tool Management

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. There are important compatibility issues, since specific tools tend to work with specific formats [18], [19]. In both parallel and embedded system domains, different solutions have been proposed to address this issue [17]. However, these solutions do not provide for easy tool plugging, nor tool cooperation.

B. Our Framework for Tool Management

We propose FrameSoC, a framework for trace analysis, in which tool management and tool cooperation is a major issue. 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 of tools is made possible by three components of our framework. First, the generic data-model for trace data (Section II) makes possible for different tools to work on the same trace. Second, the well defined but flexible data-model for trace analysis results (Section II) enables the creation of analysis flows where the output of a tool is taken as an input to another tool. The framework support is independent from the tools semantics and all the analysis logic remains within the tools. Third, the framework defines an explicit plug-in mechanism for new analysis tools. We use the Eclipse plugin management features and define a specific interface for tool integration. However, our infrastructure also supports the possibility to integrate external black-box tools.

FrameSoC comes with several *framework* tools for basic trace analysis (Figure IV-B). Namely it provides a structured trace explorer with details on trace metadata, an event-density chart to easily identify hot spots, a pie-chart gathering some statistics about the trace and a form for event querying

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

using regular expressions. Using the plugin mechanism, we plugged into FrameSoC several trace importers and exporters (importers of KPTrace, Pajé and Tima formats and an exporter to Pajé format). As for analysis tools, we integrated a tool able to do simple sequence-search with result recording and a filter for event producers, able to find and store the subset of producers being active (or idle) during a given time interval. Finally, we also propose an innovative visualization tool able to do aggregation, reusing the results of the filter above, thus implementing a real workflow (Section V). Another real example of trace analysis workflow involving the simple sequence-search tool is described in our research report [17].

V. Time Slice Aggregation Visualization

A. Visualization Scalability Issues

Gantt chart representation [20] is classically used to visualize application behavior over time thanks to its ability to represent causality relations. However, this technique suffers from time and space scalability issues. Because of the quantity of information visualized (due to event granularity, platform heterogeneity or execution duration), an analyst may be forced to zoom out or to pan and, thus, lose either the execution context or representation fidelity. A partial solution to this problem is proposed by Pajé [13], a tool for parallel application trace analysis, and LTTng Eclipse Viewer [21]. Both tools highlight events that are too small to be correctly represented using pixels. However, even if such technique shows the possible information loss, it lacks associated semantics that would help the analyst to understand the trace.

B. Time Slicing and Best Cut Partition Algorithm

We propose a solution to these scalability issues by adapting an existing Best-Cut Partition algorithm [22]. To use this 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, according to a threshold defined by the user. The major issue is to choose the value for this parameter with the objective of finding the best trade-off between simplification (group to ignore small differences) and information loss (still keep track of the relevant parts). The goal is to reveal important behavior changes in the aggregated representation and thus help the identification of trace interesting parts. The time-slicing and best-cut partition algorithm uses are more precisely described in the related research report [17]. We implement this algorithm in a FrameSoC tool that performs database queries and computations, provides a user interface and represents the aggregation results as a one-dimensional array. In the Figure IV-B the trace parts being aggregated have the same color.

C. Experimental Conditions

In our experiments, we trace a basic open-source G-Streamer video application[2], displaying a mpeg video. The first use case is a normal execution (10 s duration), taken as reference. The second case is the same video perturbed 3 s after its start by some CPU loads and memory allocations, thanks to the *stress* tool[3]. Both resulting traces have almost 1,5 millions of events produced by 1500 functions. Their size is about 110 MB.

D. Experiment Analysis

1) Global Trace Representation: Figure 3 and Figure 4 represent the traces and show that our visualization is coherent with our experiment, i.e., it matches perturbation timestamps. Consecutive parts with same colors and numbers are aggregated. We do aggregation on the whole trace cut into 20 time slices. We start our analysis with a fully aggregated trace of the non perturbed case. Here, time slice duration is 0.5 s. We progressively discover different parts by disaggregating our representation, leading to a representation that highlights a transitory state at 2 s (fifth slice), followed by a steady state of 7.5 s (Figure 3). By taking into account this application behavior, we deduce that its execution is made of an initialization followed by a constant behavior. Performing the same treatment on case 2 (where time slice duration is 0.56 s) 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 4). These timestamps match the application perturbation settings and visible behavior (3 s after the start we have 2 s of stress). These results confirm that our aggregation tool is able to highlight behavior changes in the trace. Moreover, computation duration is reasonable (a 1 m initialization time and less than 1 s to retrieve each aggregation configuration) and enables easy interactivity.

2) Analysis Flow: In the previous section, 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 helps us to see that some event producers are associated with few events (less than 10). Using the filter tool cited in Section IV-B, we isolate the event producers active only during the initialization phase from the others. These subsets of producers are saved in the FrameSoC database and retrieved by our aggregation tool. In the perturbed use case, we find that only 20% of the event producers are active after the first 1.5 s of execution. Performing the aggregation algorithm on them gives a quite similar result than the one in Figure 4. In this case, filtering at the beginning of the analysis flow helps the analyst to focus only on the event producers of interest.

VI. Conclusion

This article presents FrameSoC, a trace management framework for embedded systems. We solve the problem of trace format heterogeneity with a generic data-model able to represent not only raw data but also meta information about the trace and analysis results. For the future we are interested in the introduction 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 considering various dimensions, while keeping reasonable trace access performance. Access to data being crucial for analysis tools, our future research will consider specific use case optimizations and possible alternative storage solutions, such as temporal or non-relational databases.

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

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

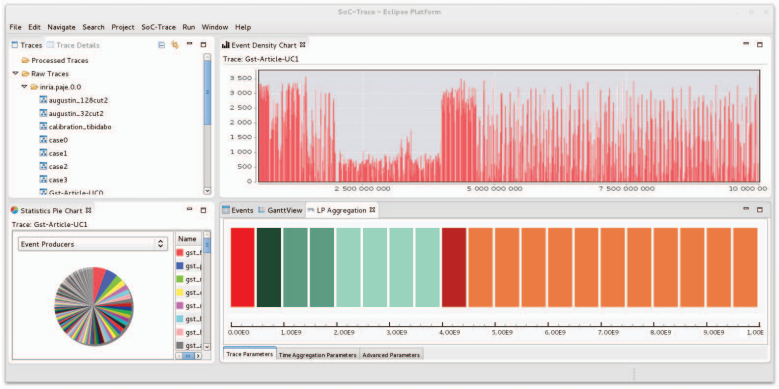

Fig. 2. FrameSoC GUI based on Eclipse, with different analysis views. On the top left, the trace explorer enables to select a trace, to get and to modify its metadata. The three other views are related to the selected trace. The statistics pie-chart (on the bottom left) represents the event producers event count. In this case, the sum of the events produced by 20% of the event producers represents 80% of trace events. Therefore, these producers are the main responsible of trace behavior. The event density chart (on the top right) gives information about the event distribution over time. It shows a disruption in trace behavior between 2 s and 4 s, where there are less events. The trace aggregated view (on the bottom right) show the same behavior, with similar timestamps. However, by disaggregating more the trace, we would show that the perturbed period is not stable, as we could think by taking into account the event density chart amplitude only.

Fig. 3. Use case 1: initialization (slices 0–3) followed by a steady state (4)

Fig. 4. Use case 2: behavior perturbation at slices 3–6

FrameSoC puts a strong emphasis on tool management and interoperability. Regarding the evolution of our framework, we expect to enlarge the family of tools working with FrameSoC thanks to the efforts of the members of the SoC-Trace project.

We have illustrated the features of FrameSoC with a trace aggregation visualization tool. This tool succeeds in representing the global behavior of an application. It has been applied on simple multimedia streaming applications, in which we have introduced some perturbations. Compared to traditional Gantt charts, the proposed view has the ability to highlight disruptions while representing the whole trace behavior over space and time. Managing a large amount of events, it succeeds in keeping the context and the representation fidelity. It could be interesting to see how a behavior disruption is propagated across different subsets of event producers, by simultaneously showing several arrays associated to these subsets.

References

[1] "KPTrace Trace Format," http://www.stlinux.com/.

[2] 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.

[3] 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.

[4] F. Wolf, F. Freitag, B. Mohr, S. Moore, and B. J. Wylie, "Large Event Traces in Parallel Performance Analysis," *Architecture of Computing Systems, ARCS 2006 (19a. Frankfurt-Main, Alemanya)*, 2006.

[5] P. López Cueva, A. Bertaux, A. Termier, J. F. Méhaut, and M. Santana, "Debugging Embedded Multimedia Application Traces Through Periodic Pattern Mining." ACM Press, 2012, p. 13.

[6] C. K. Kengne, L. Fopa, N. Ibrahim, A. Termier, M. Rousset, and T. Washio, "Enhancing the Analysis of Large Multimedia Applications Execution Traces with FrameMiner." IEEE, Dec. 2012, pp. 595–602.

[7] L. M. Schnorr, A. Legrand, and J.-M. Vincent, "Detection and Analysis of Resource Usage Anomalies in Large Distributed Systems Through Multi-Scale Visualization," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 15, pp. 1792–1816, 2012.

[8] 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.

[9] G. Pagano and V. Marangonzova-Martin, "SoC-Trace Infrastructure," INRIA, Technical Report RT-0427, Nov. 2012.

[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] P.-M. Fournier, M. Desnoyers, and M. R. Dagenais, "Combined Tracing of the Kernel and Applications with LTTng," in *Proceedings of the 2009 Linux Symposium*, 2009.

[12] D. Hedde and F. Petrot, "A Non Intrusive Simulation-Based Trace System to Analyse Multiprocessor Systems-on-Chip Software." IEEE, May 2011, pp. 106–112.

[13] 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.

[14] 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.

[15] 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.

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

[17] G. Pagano, D. Dosimont, G. Huard, V. Marangozova-Martin, and J.-M. Vincent, "Trace Management and Analysis for Embedded Systems," Research Report RR-8304, May 2013.

[18] 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.

[19] 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.

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

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

[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.