TOWARDS GENERIC SYSTEM OBSERVATION MANAGEMENT

Vania Marangozova-Martin

UNIVERSITÉ DE GRENOBLE

THÈSE D'HABILITATION À DIRIGER LES RECHERCHES

en vue de l'obtention du diplôme
d'HABILITATION À DIRIGER DES RECHERCHES
de l'UNIVERSITÉ DE GRENOBLE-ALPES, Spécialité INFORMATIQUE

présentée par

VANIA MARANGOZOVA-MARTIN

Le 12 Juin 2015

TOWARDS GENERIC SYSTEM OBSERVATION MANAGEMENT

Jury

President	Prof. Noël De Palma
Reviewers	Prof. Christine Morin
	Prof. Pierre Sens
	Prof. Felix Wolf
Examiners	Prof. Yves Denneulin

There's always a story.
It's all stories, really.
The sun coming up every day is a story.
Everything's got a story in it.
Change the story, change the world.
— Terry Pratchett

Acknowledgements

It is an honor for me to present my habilitation thesis before the excellent scientists that have agreed to participate in my jury: Prof. Dr. Christine Morin, Prof. Dr. Pierre Sens, Prof. Dr. Felix Wolf, Prof. Dr Yves Denneulin and Prof. Dr. Noël De Palma. Thank you for your consideration and appreciation.

This habilitation is a result of numerous collaborations, exchanges and joint studies with colleagues and students. My gratitude goes to Prof. Jean-François Méhaut with whom I have worked during several years, who was the initiator of the embedded adventure and helped me mature through it. I also thank Miguel Santana, CPD Software Development Tools Manager at ST Microelectronics, the fruitful collaboration with whom was a major driver for this work. In more general terms, I thank my colleagues from the MESCAL and MOAIS teams for the rich environment in which I have evolved since my beginnings as associate professor. Passionate scientists, exuberant characters, ingenious thinkers or human treasures: they are numerous to have helped me with their example, ideas and encouraging smiles. Thanks to Brigitte Plateau, Jean-Louis Roch, Jean-Marc Vincent, Vincent Danjean, Florence Perronnin, Denis Trystram, Guillaume Huard, Gregory Mounié, Arnaud Legrand, Brice Videau and all the others.

A big thank you to all the "youngsters" I have supervised, who have helped me understand and have grown their way in the process. A special thought for the PhD students and the engineers whose work is reflected in this thesis: Carlos Hernan Prada Rojas, now Doctor and working at ST, Augustin Degomme, Kiril Georgiev, now Doctor and working in Denmark, Generoso Pagano, principal developer of the Framesoc benchmark, Alexis Martin who will undoubtedly have his startup one day... But also thanks to the masters: Thibault Jurado Leduc, Pedro Penna, Margaux Clerc, Johan Cronsioe, Mickaël Vanneufville, Dimitar Slavchev...

The Springboard community and my female colleagues were there to push me through the times of doubt. A special thank you to Françoise Le Mouël, Catherine Berrut, Gaëlle Calvary and Sara Bouchenak.

For the fun and the welcome in the change, I thank Noël De Palma, Vivien Quéma, Renaud Lachaize, Didier Donsez and Nicola Palix from the ERODS team.

On the personal side, my deep gratitude goes to my family whose love and belief in me have always been an invaluable support.

V. M-M.

Abstract

One of the biggest challenges in computer science is to produce correct computer systems. One way of ensuring system correction is to use formal techniques to validate the system during its design. This approach is compulsory for critical systems but difficult and expensive for most computer systems. The alternative consists in observing and analyzing systems' behavior during execution. In this thesis, I present my research on system observation. I describe my contributions on generic observation mechanisms, on the use of observations for debugging nondeterministic systems and on the definition of an open, flexible and reproducible management of observations.

Résumé

Un des plus grands défis de l'informatique est de produire des systèmes corrects. Une manière d'assurer la correction des systèmes est d'utiliser des méthodes formelles de modélisation et de validation. Obligatoire dans le domaine des systèmes critiques, cette approche est difficile et coûteuse à mettre en place dans la plupart des systèmes informatiques. L'alternative est de vérifier le comportement des systèmes déjà développés en observant et analysant leur comportement à l'exécution. Ce mémoire présente mes contributions autour de l'observation des systèmes. Il discute de la définition de mécanismes génériques d'observation, de l'exploitation des observations pour le débogage de systèmes non déterministes et de la gestion ouverte, flexible et reproductible d'observations .

Резюме

Едно от най-големите предизвикателства на информатиката е да създава правилно работещи компютърни системи. За да се гарантира коректността на една система, по време на дизайн могат де се прилагат формални методи за моделиране и валидация. Този подход е за съжаление труден и скъп за приложение при мнозинството компютърни системи. Алтернативният подход е да се наблюдава и анализира поведението на системата по време на изпълнение след нейното създаване. В този доклад представям научната си работа по въпроса за наблюдение на копютърните системи. Предлагам един общ поглед на три основни страни на проблема: как трябва да се наблюдават компютърните системи, как се използват наблюденията при недетерминистични системи и как се работи по отворен, гъвкав и възпроизводим начин с наблюдения.

Contents

1 Introduction

1.1 Computer Systems and Embedded Systems

A computer system is composed of hardware and software working together to provide a specified functionality. The hardware includes processing units (general purpose processors, accelerators), storage elements (memory, disks) and communication components (network, peripherals). The software defines the way hardware resources are to be exploited to ensure the system functionality.

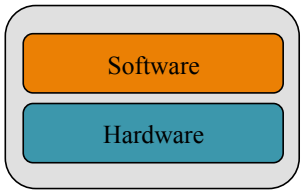


Figure 1.1 – A Computer System

The hardware and software architecture of a system differ according to its required functionality i.e. to its application domain. A *desktop system*, for example, mostly targets single users running interactive applications. A standard hardware configuration would include a processor, main memory, a storage disk and various peripherals. The software will typically be composed of an operating system (OS) and a number of user applications. The OS provides hardware abstractions, manages hardware resource sharing and guarantees the execution of the applications in isolation.

Distributed systems allow users to benefit from applications running on distant machines. At the hardware level, a distributed system unites the resources of multiple machines interconnected through a network. The software of a distributed system typically includes a distributed *middleware* whose role is to hide the distribution complexity from applications. The middleware tackles the issues of failures and network latencies, provides an abstract vision of the network (e.g with a network overlay) and includes common services such as naming, communication, group management, security, fault tolerance, etc. Distributed applications go from

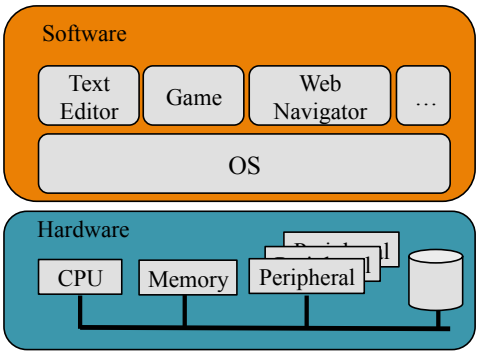


Figure 1.2 – A Desktop System

classic client-server, through peer-to-peer architectures to large-scale hybrid organizations brought to existence by the development of cloud computing.

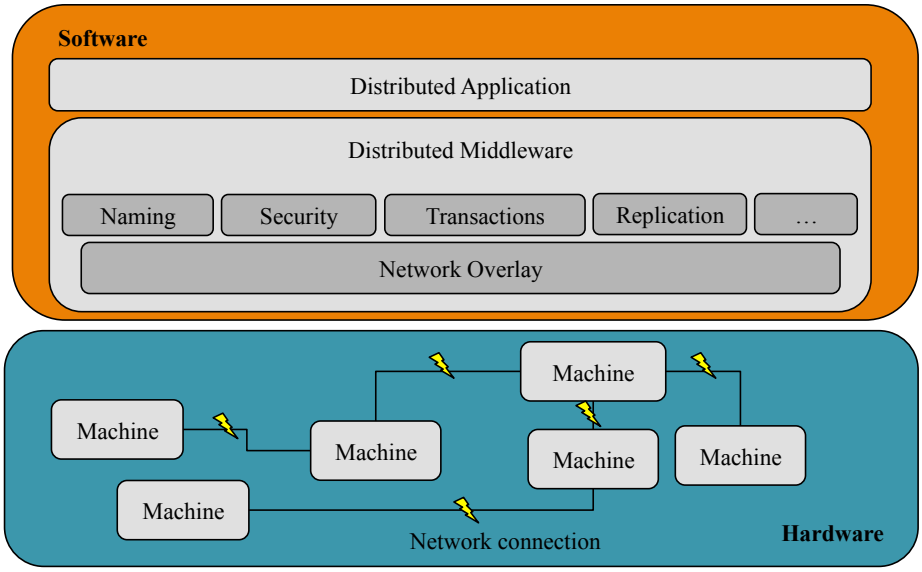


Figure 1.3 – A Distributed System

High performance computing (HPC) systems address data-intensive and computationally intensive applications such as scientific simulations. These applications need to execute many computations on large sets of data while optimizing the global execution time or the computation throughput. The hardware, as a consequence, is designed for parallel computations and typically includes hundreds, even thousands, processing cores. The architecture is hierarchical, with shared memory multi-core nodes interconnected by a fast network. These interconnections form clusters which can further be connected in a grid. At the software level, we usually find a resource management system monitoring resource availability and providing

for resource allocation. On top of it, there is a runtime environment providing a specific programming model (e.g MPI, OpenMP, Charm++,...) for the high-end parallel applications.

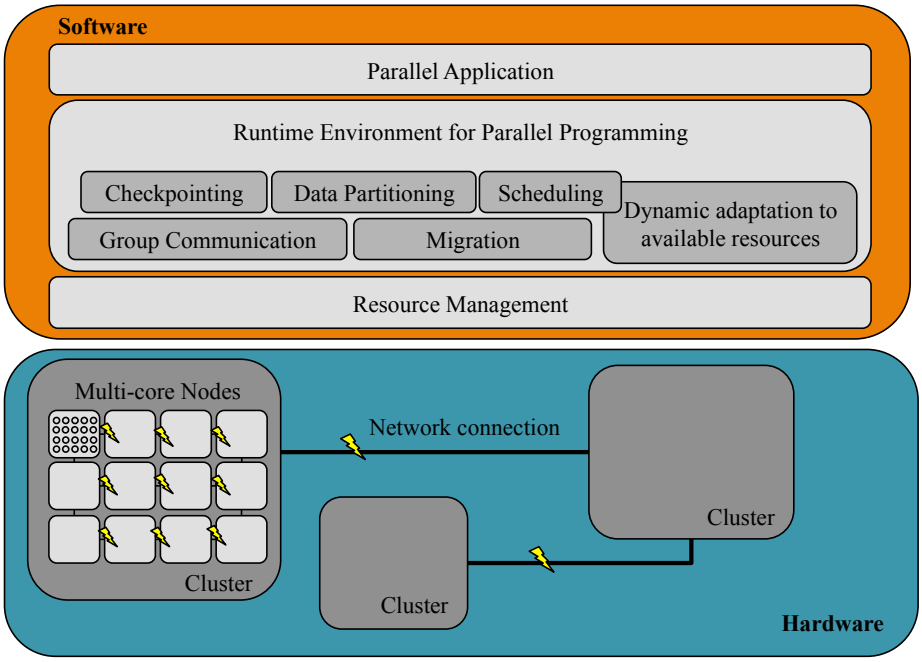


Figure 1.4 – A High-Performance Computing (HPC) System

What about *embedded systems*? Initially, embedded systems were built as simple computer systems with specific hardware and software designed to control a specific task [62, 13, 61]. Such systems are used in consumer electronics, telecommunications, medical systems, transportation, military applications, etc. They operate in constrained environments imposing real time requirements, size constraints, limited data storage, energy efficiency, etc. Their design follows a top-down approach which consists in specifying their functional requirements first and providing the supporting hardware accordingly. Existing systems are closely integrated black boxes specific to the vendor.

Today, embedded systems have entered multiple spheres of everyday life and undergo a tremendous technological evolution. Indeed, the classic design approach cannot meet anymore the requirements for shorter time-to-market, lower production cost and smarter devices. The design process has evolved to consider separately the hardware and the software which both grow in complexity.

At the *hardware level*, the trend is to develop more general-purpose architectures that could serve multiple purposes. The term *SoC* (System-on-Chip) reflects the fact that today, a single chip integrates the hardware components and the performance characteristics of a full computer. As in standard computer architectures, SoC systems have memory, processors and I/O devices. Contrary to standard computer architectures, there is no standard SoC architecture but an exceptional variety of hardware designs. Moreover, a typical SoC includes

not only general-purpose processors but also accelerators (Digital Signal Processors, DSP) for fast specialized treatments. SoC systems have also increased the number of integrated computing cores and have evolved towards Multiple-Processor Systems-on-Chip (*MPSoC*) and Multiple-Core/Many-Core Systems-on-Chip (*MCSoC*). The increased number of hardware components has brought the use of *Networks-on-Chip (NoC)*. With architectures such as the STHORM design [19] or the Kalray's MPPA [73], SoC become integrated parallel systems with both HPC and distributed issues to tackle.

At the *software level*, embedded systems slowly evolve towards a layered approach including operating systems, middleware and applications. Currently, however, most of the embedded software provided by vendors concerns operating systems. Even if they implement similar concepts (interruption management, memory management, multitasking...), there is a rich variety of systems having their own interfaces [42]. Embedded middleware is in its beginnings and emerges mostly in specific areas, such as multimedia, whose common services start to be well-known. The generalized use of middleware is hindered by the fact that introducing additional software layers between the hardware and applications slows down the execution which is critical en embedded systems.

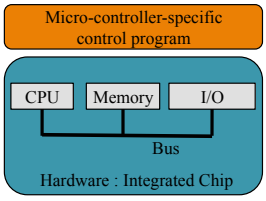


Figure 1.5 – A Microcontroller System

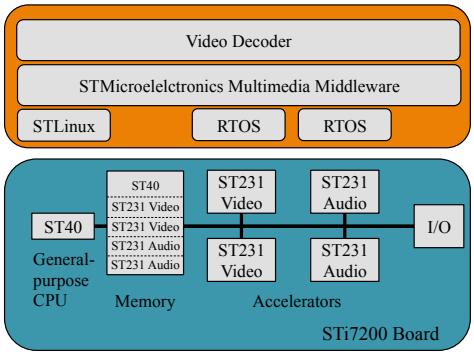


Figure 1.6 – A Multimedia System

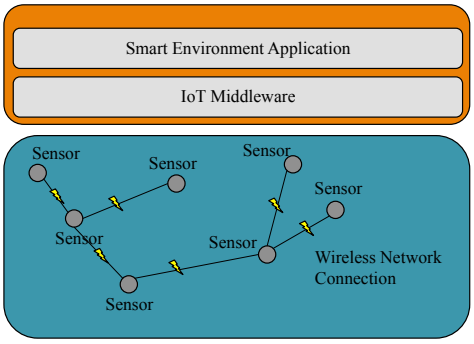


Figure 1.7 – An IoT System

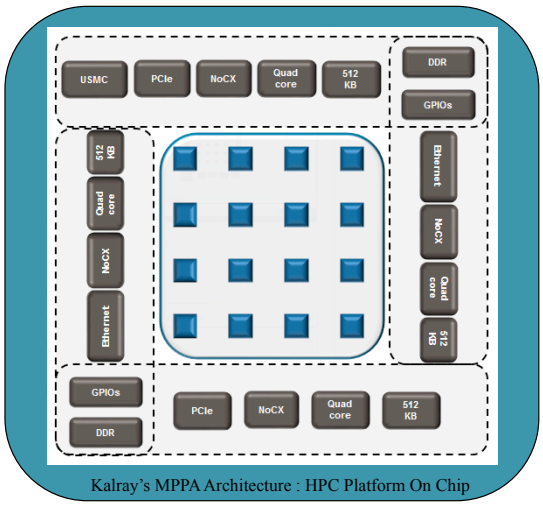


Figure 1.8 – An HPC-on-Chip System

One major change in the embedded system domain is the fact that embedded systems have evolved from closed, black-box, ready-to-use devices to open experimental platforms. Users can write and deploy their own applications [7, 6, 60] and even build their own hardware platforms [8]. The new technological possibilities, user imagination and the development of the *Internet-of-Things (IoT)* brings a new exciting era for embedded systems.

To conclude, the major aspects characterizing today embedded systems are:

- *Size and Performance Constraints.* Even if embedded systems undergo important technological evolutions, they remain constrained systems. They are constantly diminishing in size and have limited autonomy, as well as computational and storage capacity.
- *Heterogeneity.* There is an explosion in the variety of embedded hardware designs [61], embedded software and embedded uses. Vendors provide multiple types of embedded processors, as well as interconnection boards. Embedded software covers proprietary solutions, real-time operating systems, releases of open-source operating systems like Ubuntu or Android, higher-level (middleware) services, etc.
- *Ubiquity.* Embedded systems are everywhere. Embedded systems include critical systems as controllers in nuclear stations or airspace shuttles. Embedded systems bring TV to people homes with set-top boxes. Embedded systems are an integral part of all kinds of transports. Embedded systems include smartphones and tablets. Embedded systems include cameras, sensors and all kinds of connected "gadgets" for smart environments.
- *Lack of standards.* The embedded system domain experiments today with the creation of various devices and corresponding software. The competition for providing the most optimized or original device results in the lack of standards in terms of both hardware and software architectures [153, 101].

1.2 On the Importance of System Observation

System observation consists in gathering information about a system's execution. It reflects what happens both at the hardware and the software levels. The gathered information may concern interruptions, hardware counters' values, CPU load over time, function calls, memory management, etc.

System observation is used in the process of validating *correct* system functionality in cases where formal validation methods are too complex or too costly to apply. Indeed, formal methods [17, 3] are widely used in critical systems, such as nuclear stations or space shuttles, where failures have dramatic consequences. Such systems follow a model-driven design in which the system behavior is modeled and proven correct before the actual development of the system [17, 3]. In most cases, however, the scale, the complexity or the variability of a system cannot be properly represented with a formal model. This is why, the behavior of distributed, HPC and emerging embedded systems is mostly explored using system observation.

System observation is structured in two main phases: 1) information collection and 2) information exploitation. The first phase is responsible of accessing and retrieving relevant information about the system. It happens during the execution of the system but may require an initial phase that prepares the system to be observed (instrumentation) (cf. Figure 1.9). The second phase consists in analyzing the captured information in order to understand and/or enhance the system. It may be done during the execution of the system (*online*) or after (*off-line*). The design and implementation of the two observation phases depend on what goals are pursued through system observation. In most works, the accent is put on the first phase i.e on the collection of execution-related information. How the information is to be used during the second phase is usually left up to the developer's experience.

In the following, we start by identifying the major goals that are pursued through system observation before discussing in turn the major issues in information collection and exploitation.

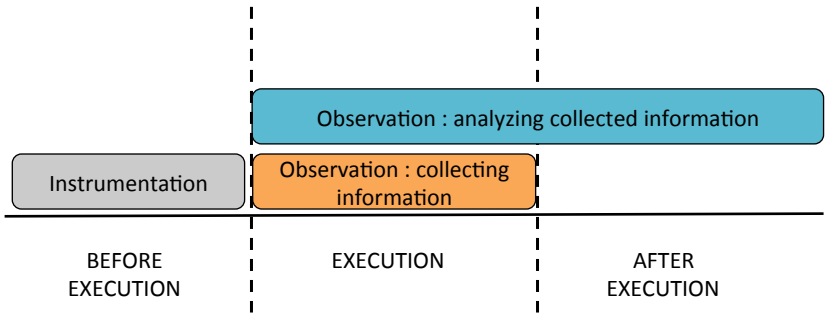


Figure 1.9 – Observation

1.2.1 Observation Goals

System observation has three major goals:

- *Check upon system correction*
  In this context, the system is observed in order to decide whether its execution is compliant with some well-defined use cases. This may be done in two forms: *testing* [74] or *monitoring* [43]. Testing compares execution results to the ones defined in specifications. The collection of information, as well as their exploitation is done at the end of an execution. Monitoring, on the other hand, considers long-running systems and collects information throughout the whole execution. The exploitation of these information is usually done online and consists in verifying that there is not an offending operation.

- *Find the cause for an incorrect execution*
  When an incorrect behavior is detected, *debugging* is used to execute the system step by step to identify the error's source. The collection of information is done during execution, at specific execution points (e.g. *breakpoints*). The information typically reflects the call stack, the state of the memory, etc. The analysis is done by the developer who guides the debugging process.

- *Characterize the execution*
  In multiple contexts, the information collected during system execution is basically used to provide an insight about the system's behavior and performances. *Profiling* [126, 53, 36], for example, provides statistical measures about the system execution. Typically, it counts the occurrences of a given event (function calls, context switches), quantifies resource usage (CPU load, disk activity, etc) and reflects the system activity in different parts of the code. *Logging* [18] captures the execution status of a set of predefined system operations and provides a historical record of error messages. Finally, *tracing*, targets fine-grained execution events and provides a detailed execution record. All three types of observation are done during the execution of the system. The collected information is usually exploited *a posteriori* to check the system correction or to debug an incorrect execution.

1.2.2 Collecting Observations

When collecting information about a system's execution, the major design issues are the following.

- *Coverage (What should be observed?*
  In most cases, the system complexity makes impossible the observation of all aspects of a system's execution. As a consequence, it is necessary to decide what is to be observed. The choice is difficult as there are various entities and operations. For example, observation may focus on hardware (CPU, network links, sensors...), on the system layer (system calls, context switches, memory accesses,...), on middleware (replication operations, communication operations,...) or on the application (function calls, object instanciation,...). It may follow dynamic entities (tasks, processes...) or the static software structure (modules, packages,...). Finally, observation may need to consider all the occurrences of a given phenomenon or only the occurrences in a given context. In most cases, as it is impossible to predict what could cause an execution problem, the general trend is to observe as much as possible and make out what happened later.

- *Cost (What is the overhead?)*
  Observation perturbs the behavior of a system. This perturbation represents the observation cost and is called *intrusion*. It may affect the system in different ways: slow it down, modify the execution path or change the final result. It does not imperatively lead to an error. For example, in operating systems, an intrusion of maximum 5% is considered acceptable.

  The intrusion is proportional to the quantity of information collected during system observation. Indeed, the more fine-grained the observation, the greater the number of execution events to be intercepted and recorded. The level of detail of an observation (its precision) is thus directly related to its cost.

- *Result (What data should be collected?)*
  The information gathered during a system's execution may be a detailed execution his-

tory or, on the contrary, a synthetic representation. In the first case, observation reveals system *events* and produces a chronological trace of its execution. This information is useful for a detailed analysis of the system execution. The second case concerns the production of *profiles* which give a macroscopic vision of the system execution. This information is usually used to characterize the performances of the system.

An important aspect of system observation is the transport and the storage of the result. Not only it puts additional requirements in terms of system resources but the corresponding operations increase the slow down of the system.

- *Quality (Did this really happen?)*
  As observations are used to reason about system behavior, a major question is to what extent they reflect the execution[50]. Indeed, gathered data may vary in precision and recording the exact ordering of system operations, especially in a large-scale system, is a challenge.

- *Configurability (How to collect what the developer really needs?)*
  A tool for observation is configurable if it gives its user the ability to specify what is to be observed and what data is to be collected. Such features help minimizing the observation overhead and simplify the further analysis of the data. However, most observation solutions provide a predefined set of observables. If the user needs a different type of information, it is usually required to use a different tool.

1.2.3 Exploiting Observations

Observation exploitation includes all processing treatments applied to collected observational data. From the architectural point of view, it relies on a storage support, an access interface and a set of computational treatments (cf. Figure 1.10). The storage contains the collected data. It may be based on persistent storage or volatile memory, benefit from SQL-like systems or be a collection of files. It may also be encoded or encrypted for space-optimization or security reasons. The way data is represented in the storage is usually specific to the tool responsible for the data collection. For this reason, it is made accessible through an explicit interface. Typically, the interface will provide operations for going through and consulting attributes of the observation records. Finally, the computational treatments are the ones analyzing the observational data and producing different or more synthetic system representations.

If there are various data collection techniques, data formats and data access interfaces [171], from the functional point of view, observation exploitation is still in its initial phase. Indeed, the extraction of synthetic human-understandable representation of a system execution is a major research topic. However, the following operations are common to multiple tools coming from different application domains and may be identified as an initial functional kernel for data exploitation.

- *Read/Write Access*
  Read/Write operations are the required basics for data manipulation. Depending on the

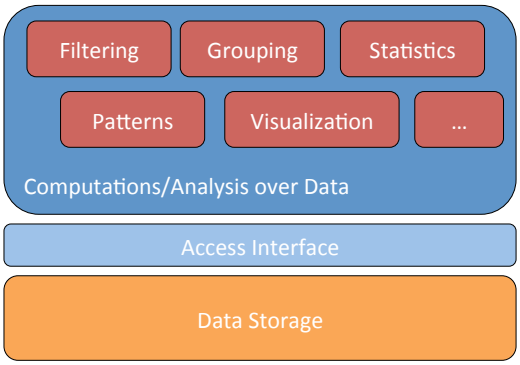


Figure 1.10 – Exploiting Observational Data

data structure, these operations may be more or less optimized, including aspects of encoding and parallelization.

- *Filtrering/Selection*
  Most observations contain different types of data for different types of analysis. As a consequence, it is typical to first filter the *useful* data before applying some treatment. The filtering criteria may typically concern the time window or the type of considered events. In some domains, such operations are used to *purify* the data i.e to dismiss wrong or unrepresentative data.

- *State Computation*
  Many observations reflect punctual events i.e phenomena happening at a given moment of time. State computation consists in revealing system states which are defined in relation with the system semantics. Examples of states are function durations and message communications.

- *Statistics*
  Statistics provide a synthetic view of the system execution. They usually emphasize the importance of a certain characteristic (e.g number of function calls, CPU load) during a given execution period.

- *Aggregation*
  Aggregation consists in applying a function on a set of data to obtain a synthetic result. It allows for diminution of data volume and is used to simplify the analysis. However, aggregation hides execution details and leads to information loss. Visualization aggregation, for example, may be nondeterministic, hide execution problems and lead to false conclusions [121, 122, 82].

- *Grouping*
  Groups, with possible hierarchical organization, are used to represent collected data following the application logic, the software architecture or the used hardware resources. In HPC, for example, it is typical to group observational data per process [78]. In embedded systems, it is typical to consider the execution events per processor [129].

- *Pattern Detection*
  Closely related to the semantics of a system, pattern detection targets the identification of abnormal situations indicating an execution problem. In HPC applications, for example, it is used to search for long-duration communications or blocking synchronizations [112, 120].

- *Visualization*
  Visualization aims at representing the results from observing and analyzing the system [143, 120, 107, 129]. It stays one of the major techniques for manipulating and understanding voluminous and complex data (*big data*) [45].

1.3 Observation Challenges in Embedded Systems

The growing complexity of embedded systems prevents the generalized application of formal methods to guarantee correct functionality and optimal performances at the design phase. The complexity of embedded systems architecture, as well as the distribution of the embedded systems design among different departments of the vendor companies and even among vendors and clients, make "the global picture" quite challenging to obtain and understand. In this context, execution observation becomes a major tool for enhancing system behavior. The definition of the JTAG standard [70] for testing, tracing and debugging is directly related to this phenomenon.

The observation process is faced with multiple challenges both during the collection and the exploitation phases.

Need for Genericity During the collection phase, the ultimate goal is to find the optimal trade-off between the observation intrusion and the quantity of useful data reflecting the system exploitation.To minimize intrusion, most existing tools propose platform-specific observation solutions [135, 147, 86][171]. These may reflect the underlying hardware or focus on the specific entities of a given software. Not only the implementation of such a solution demands a great technical expertise, but it is also confined to its initial application context and cannot be reused. This is particularly true in the domain of embedded systems where observation requires additional resources and directly affects the execution performances. Existing solutions for embedded systems minimize the execution overhead in an ad-hoc and even vendor-specific manner.

The specificity and non reusability of tools is also reflected in their data exploitation facilities. Indeed, most existing tools come with some functionalities for accessing and manipulating observation-related data, the most sophisticated treatment being data visualization. Considering different tools for the same domain or tools from different domains, one can notice that there is a common functional kernel concerning data organization, statistics treatments, machine learning methods, etc. However, even if the principles and the algorithms are close, their implementations remain context/platform-specific. The computational treatments over observational data should be separated from the domain specificity and define generic bricks

for data analysis. Such bricks would allow for reuse, optimization and refinement. Most of all, they would provide a starting point for data analysis and facilitate the definition of more complex and rich computations.

Need for High-Level Information As performances are a key issue in embedded systems, most observation solutions focus on low-level information concerning the hardware and the operating system. The collected data does reflect the resource usage of the system but presents two major flaws. First of all, low-level execution events have a high frequency and the resulting data has an important volume which is difficult to transport, store and manipulate. Second, it is quite challenging to establish the relation between low-level events and application-level phenomena. In most cases the gap is filled by the developer who may use ad-hoc tools to observe the application and rely on his/her experience to correlate low-level and application-level events. Such work is hindered by the important number of observation tools, their heterogeneity and the fact that they are meant to be used independently.

Extracting a macroscopic vision from microscopic execution events is a major issue in data analysis and thus is part of the data exploitation phase. The "big data" challenge consists in finding the right question to ask in order to obtain useful information. There is much research on the different methods to extract information from big data [5]. In the domain of embedded systems, however, the diversity of applications and therefore the lack of pivot domain semantics makes difficult the application of higher-level analysis methods. Investigations have started [14, 76] but remain limited and context-specific.

Need for Scalability With the miniaturization of embedded systems and the increasing number of hardware and software components, scalability is of prime importance. From the observation point of view, proposed solutions should not only be able to handle important data volumes but also target a subset of the system in order to zoom into interesting execution phenomena. Scalability support is to be thought of at all levels, starting from hardware support for observation, through the cost of software instrumentation, the infrastructure for data collection and up to the algorithms and frameworks to handle observation data.

The increasing scale of systems presents another challenge to observation which is nondeterminism. Indeed, as the execution may take different paths, the results of observation are also nondeterministic. How can we decide to what extent these observations reflect the behavior of the system? Does an observation reflect a normal behavior, a frequent behavior or an abnormal one? How can we investigate what happened during the execution if we are not sure to reproduce the execution path? In a large-scale system where, due to intrusion reasons, developers observe different parts of the system during consecutive runs, how could they be sure that the corresponding observations are consistent and do not reflect different execution cases? A tempting solution may be to prevent nondeterminism via adapted hardware, runtime or programming mechanisms [38, 20, 21, 24]. This solution does guarantee execution reproduction but is costly in terms of hardware or development efforts and cannot be applied in the general case.

1.4 Research Contributions

My work has contributed on the generic mechanisms for tracing of embedded systems, the exploitation of traces for debugging nondeterministic embedded systems and the definition of an open framework for trace exploitation. I have namely worked on:

EMBera: A Generic Framework for Embedded System Observation
The motivation behind this work lies in the existence of numerous observation tools for embedded systems which are technology-driven, platform-specific and non reusable. Usually they provide fine-grain low-level representation of the system execution and do not address the issues of multilevel tracing and of tracing configuration. In other terms, what is observed, what data is captured and what is done with the data is predefined in existing tools.

The work developed in the PhD thesis of Carlos Herman Prada Rojas proposes a generic framework for observation of embedded systems. Using a component-oriented approach, EMBera addresses the aspects of observation genericity, of partial observation, of multi-level observation and of configuration. The major contributions of this work consist in 1) identifying the major phases in a typical observation activity, 2) the proposal of generic basic building blocks for observing an embedded system and 3) the instantiation of the proposal on real embedded boards with use cases provided by STMicroelectronics.

After his PhD, Carlos Rojas Prada obtained a permanent position at the IDTEC team at STMicroelectronics. He continues his work on observation tools and has been a principal instigator of the Multi-Target Trace API (MTTA).

The publications of this work are [190, 189, 188, 186, 187].

EMBera is presented in the first chapter of this report.

RedSoC: Debugging Nondeterministic Errors in Embedded Systems
The increasing complexity of embedded architectures has a strong impact on their debugging. A major problem, intensified by the increasing number of computing cores and the level of system parallelization, are nondeterministic errors. Classic debugging techniques do not apply in this context as, on one hand, they are not scalable enough and, on the other hand, they cannot guarantee the reproduction of the errors.

The work developed during the PhD thesis of Kiril Georgiev proposes a record-replay solution allowing for post mortem debugging using execution traces. RedSoC defines a debugging cycle allowing for zooming on errors by applying temporal and spatial selection criteria. The idea behind spatial and temporal selection is to consider not the entire execution of the whole application but deterministically replay a part of the application during a specific execution interval. The proposed mechanisms are connected to GDB and allow for a useful visual representation of the trace.

After his PhD, Kiril Georgiev obtained a permanent position at the R&D department of Excitor A/S, a Danmark company developing solutions for securing the working environment of mobile users.

The publications of this work are [158, 155, 156, 159, 160, 161].

ReDSoC is presented in the second chapter of this report.

Framesoc: Trace Management Infrastructure for Embedded Systems
In the domain of embedded systems, the aspect of trace capture with minimum execution overhead is well managed by existing tracing solutions. However, trace exploitation is still at its early stage and most existing tools only provide time-chart visualizations and some basic statistics. Framesoc, developed in the context of the SoC-TRACE FUI project, targets the design of an open infrastructure for next generation trace management.

Publicly available since July 2014 (soc-trace.minalogic.net), the first version of the SoC-TRACE software product provides solutions for trace storage, trace access and trace analysis. For trace storage, Framesoc tackles the problem of trace format heterogeneity and provides a generic data model. Stored data includes not only captured trace data but also the results of trace processing (analysis) treatments. As for trace analysis, Framesoc provides a management framework that has successfully integrated data mining tools (LIG/UJF), sequence statistics (ProbaYes) and visualization (INRIA and STMicroelectronics).

Since september 2014, Framesoc has been published as an open-source project (http://soctrace-inria.github.io/framesoc/). Recent works of Generoso Pagano, engineer in SoC-TRACE, have integrated the LTTng viewer in Framesoc. This connects our work to a very large open community and enlarges the application domain of the framework. Another line of work is pursued by Alexis Martin, a PhD student, investigating the automatization of trace analysis workflows. The idea is to be able to define basic analysis blocks that could be connected to form complex analysis chains. The support for such features would allow for reusing the processing treatments in different contexts with different traces, the construction of a knowledge base of interesting trace analysis cases and the reproduction of trace analyses.

The publications of this work are [179, 170, 171, 178, 180, 177, 181, 173, 172].

The work around Framesoc and the definition of a trace management infrastructure are described in the third chapter of this report.

The three chapters dedicated to EMBera, ReDSoC and Framesoc are followed by Chapter 5 presenting the research perspectives of this work.

2 Generic Observation with EMBera

The performance requirements of embedded systems, as well as the hardware/software codesign have influenced existing observation solutions in an important way. Indeed, existing observation solutions are centered either on the SoC architecture [9, 92], or on the operating system software [135, 86]. At the hardware level, specialized components intercept, timestamp and trace processor, memory and bus events. At the processor level, there typically are registers containing information about the number of executed instructions, the number of memory accesses, the cache misses, etc. At the operating system level, the observed events are related to interruptions, context switches and system calls.

What can be said about these observation solutions is that they are platform(vendor)-specific and non reusable. They provide low-level information which is difficult to analyze and correlate with higher level software functionalities (cf. Figure 2.1). They provide a predefined set of observable data and the configuration facilities are limited to activating/deactivating observation points.

2.1 Component-Oriented Systems and Observation

Components have been introduced to simplify the development and management of complex applications. They emphasize the modular approach to development, promote reuse and, most of all, address application administration. Indeed, components may be used to separate the software configuration aspects related to a specific platform from the reusable functional kernel [139, 138]. Compared to the object-oriented approach, components are coarser grained, express the management aspects of applications (eg. security, fault tolerance, life cycle management) and explicit their architecture. A major advantage is the fact that components explicitly target application deployment and thus facilitate the process [163].

In our work we consider the definition of components given by Stal in [25]. A component is a reusable entity, a black box, with well-defined interfaces that characterize the services it requires from the external environment and the services it provides (cf. Figure 2.2(a)). Components are connected through their interfaces (cf. figure 2.2(b)). to create composite components and the final application (cf. figure 2.2(c)).

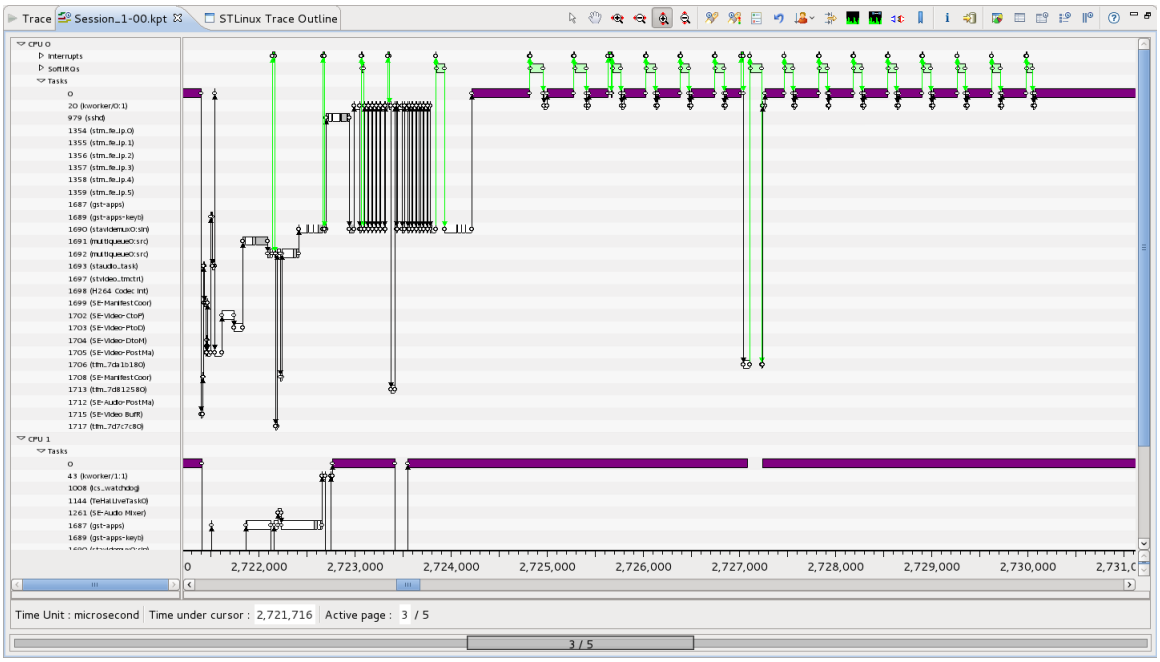


Figure 2.1 – STMicrolecetronics' KPTraceViewer is part of STWorkbench and visualizes KPTrace execution traces. This view shows the different processors (`CPU0`, `CPU1`), as well as the related `Inetrrupts`, software interrupts (`SoftIRQ`) and function executions (`Tasks`). The information displayed concerns a time window of only $10\mu s$.

There are multiple component models [137, 58, 104, 26, 2, 152, 35] and their usage continues to spread, especially with the development of cloud computing. If observation facilities for component-based systems are provided, they are mostly focused on end-user applications and middleware. Observation typically targets component architecture and component interactions. The Fractal component model [26], for example, can detail the set of executing components and the existing bindings between components. It can also trace component creations and communications. Similarly, OpenCCM [104], an open-source implementation of the CORBA Component Model [58], uses interceptors in order to capture method invocation, and thus, monitor component creations and communications. The same approach is applied to the implementations of the EJB model [137]. Component observation at the application level has an important advantage which is to be independent of the underlying system software and hardware. However, it is unfortunately unrelated to low-level performance metrics which are crucial for embedded system development.

Few projects employ components for the needs of embedded systems. The PURE project [23], for example, targets deeply-embedded systems but focuses mainly on the trade-off between efficiency and software engineering and not on observation. The PIN component model provides a simple component framework targeting embedded system design [65]. It does not consider execution observation but features frameworks reasoning about system performance and prediction [15]. The ROBOCOP [115, 87] ITEA project has combined the KOALA [144], COM and CORBA models to propose a suitable component-based solution for consumer

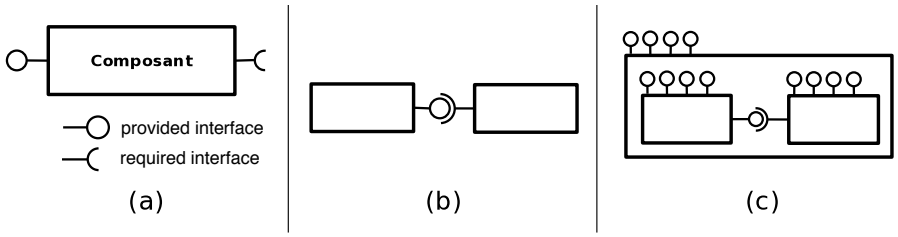


Figure 2.2 – Component interfaces, connexions and composition

electronics design. The Robocop model defines component properties related to memory consumption, timing and reliability. However, these properties are mostly used as resource requirements declared prior to the execution and managed through resource budgeting. Resource monitoring and control is mentioned but is left to future developments. As for the Nomadik Multiprocessing Framework project [44] of STMicroelectronics which defines the context of our work, observation concerns the application component level and stays disconnected from the resource perspective.

2.2 EMBera : Component-based Generic Observation

The EMBera project [186, 187, 188, 189] explores the use of components for designing a generic framework for observing embedded systems. The major contribution of EMBera is to provide an observation framework providing for partial observation, reuse, scalability and configuration.

Partial Observation As it is not realistic to observe everything that happens during the execution of an MPSoC system, the EMBera project allows for selection of the observable entities and actions in a system. To do so, EMBera encapsulates the chosen entities in observable components providing generic observation interfaces to control the data collection process. It may focus, for example, on a group of cores, on communications or on some application modules.

Genericity and Reuse EMBera abstracts the data collection framework from the underlying platform's complexity and heterogeneity. It defines a *probe* base layer which is responsible of collecting raw data from the system and relies on the implementation of specific components to encapsulate different data sources. The *probe* components provide unified observation interfaces upon which the data collection framework defines generic data processing components. These do not depend on the platform and thus may be reused in a different application contexts or on different MPSoC hardware.

Scalability Scalability is managed in EMBera through the possibility for partial observation, as well as using the natural capacity of components to form hierarchies. Indeed, in a large-scale system, it is possible to define coarse grain components providing observation information

about important parts of the system. When needed, these may be defined as composite components and zoom to a finer level of detail through the components they contain.

Configuration To take into account the specificity of embedded platforms and of the entities to observe, EMBera provides source code skeletons to be completed with suitable optimized treatments.

2.2.1 Approach

The EMBera model is inspired by the Fractal component model [26]. We have chosen Fractal since it is a general component model that is system and language independent. Indeed, it can be used at the system level, as well as middleware or application level and it can be implemented in Java, C or other programming languages. Another major advantage of Fractal is that it is already used at STMicroelectronics which defines our working context [89].

An EMBera application is composed of a number of interconnected components. Components are active entities and each component has its own execution flow. This choice follows the current practice for MPSoC applications in which multiple treatments are executed on different processor units.

Each component is characterized by a set of provided and required interfaces. A predefined interface for component control includes operations for component creation, interconnection and life-cycle management. To explicitly address system observation, each EMBera component also provides *observation interfaces*. These include an introspection interface that provides information about the observable events, a control interface to enable or disable observation-related treatments and a data interface to access collected observation data.

Targeting post-mortem data processing, EMBera focuses on the data collection aspects of observation (cf. Figure 2.3). The *selection* phase is responsible of defining the part of the system to target during observation. The *instrumentation* phase prepares the system to be observed. The *capture* phase collects execution data. The *pretreatment* phase typically relies on filtering and aggregation to eliminate wrong, redundant or unimportant data and decrease data volume. *Formatting* unifies data representation before the *Storage*.

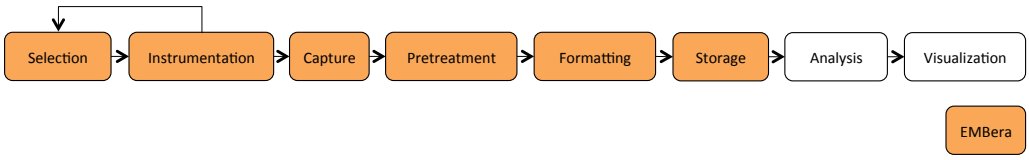


Figure 2.3 – Observation for post-mortem processing

To reflect the identified aspects of observation, EMBera defines three types of components: basic components, data treatment components and storage components.

Basic Components are the probe components that encapsulate platform-specific entities in order to make them observable in a generic way. A basic component may be used to

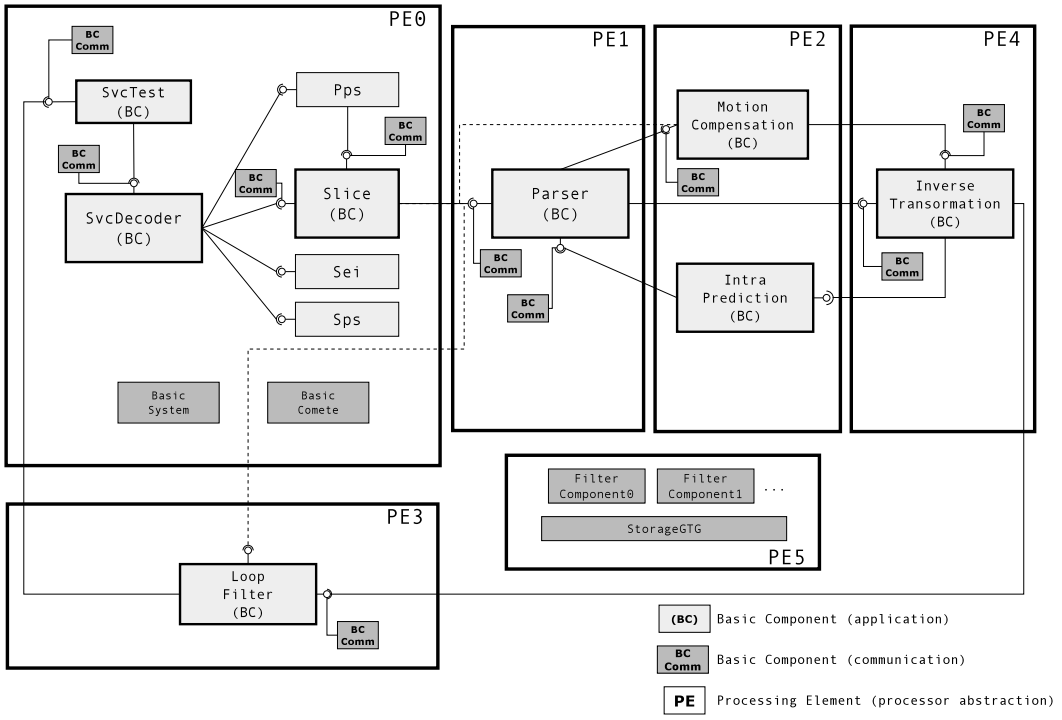


Figure 2.4 – EMBera Components in the SVC Application [190]

encapsulate, for example, a process or a hardware register. Basic components represent the sources of observation data in the EMBera model.

Data Treatment Components are responsible for pretreating the observation data before storage. Typically these encapsulates filter or aggregator operators and apply them to the data produced by one or more basic or data treatment components.

Storage Components address data formatting and the data storage on persistent supports. They encapsulate the specific mechanisms for accessing the target storage.

2.2.2 Validation

The EMBera framework has been implemented in C and has been instantiated in three different use cases, featuring different applications and execution platforms. It has been validated with multimedia applications which represent a major part of the software running on commodity embedded systems. The applications include both component-based and not component-based software with different size and complexity. The simplest application is a test decoder for the MJPEG format [93] which has been reengineered to become component-based. The second application is a non component-based video decoding middleware provided by STMicroelectronics. The software stack in the third use case is composed of the Linux operating system, a component-based multimedia middleware (Comete) and the SVC decoder application [123].

Figure 2.4 shows the integration of the EMBera components in the SVC application. The compilation process has been modified to enrich SVC components with the EMBera's observation facilities and to monitor inter-component communications.

Figure 2.5 shows a partial representation of the multi-level EMBera observation. It features the observation of the application components, of their management at the middleware level and of their resource aconsumption at the system level.

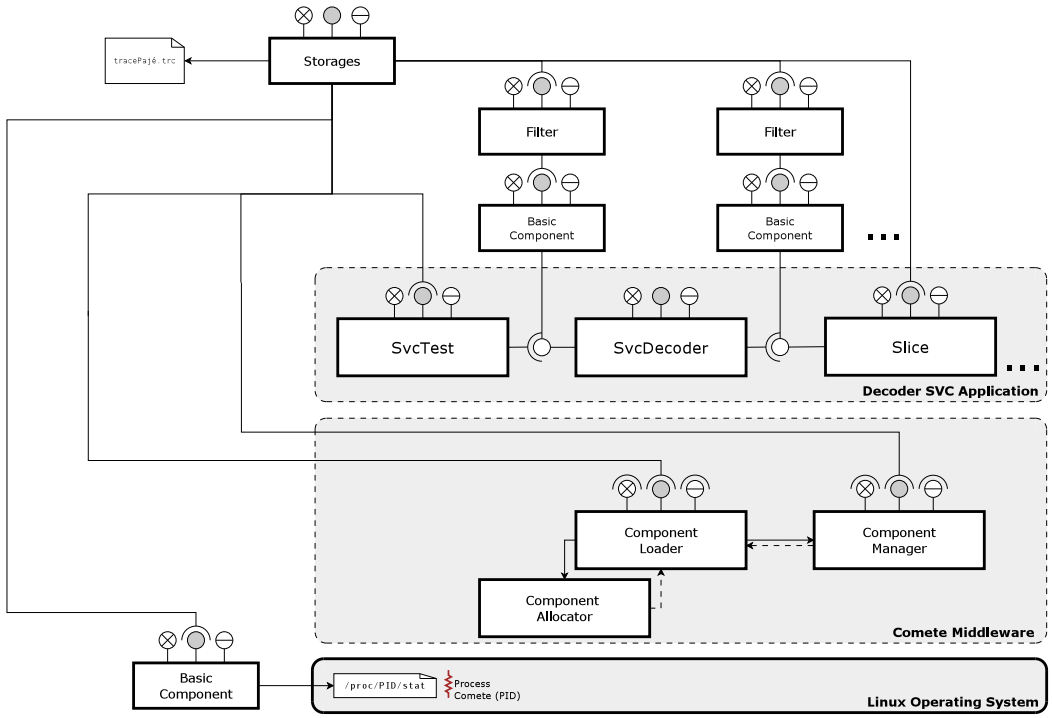


Figure 2.5 – Partial representation of the EMBera Observation Components in the SVC use case [190]

In terms of target platforms, the first two applications have been deployed on two embedded boards from the STi7200 family [132]. As at the time of the experience, SMP embedded platforms were not available, in the third use case the target hardware platform is a Linux SMP with four six-core Intel Xeon processors.

In all cases, the experiences included the porting of the EMBera software for the used platform and the instrumentation of the available software. Observations are multi-level and target the system, the middleware and the applications. In addition to providing useful insight about the design of the software, EMBera observations revealed performance problems related to memory management and non optimal processor usage. Moreover, they helped compare different application deployment schemes and identify the most consuming software components.

2.3 Conclusions and Research Perspectives

The EMBera framework shows that components can be successfully employed to manage data collection in a generic way. Its concepts may be applied in the context of non component-oriented software running on other than embedded systems.

EMBera has been used to provide multi-level observation in an unified way. Capturing the same data without EMBera would have implied the usage of several heterogenous tools. Using heterogenous tools is a complex issue as it requires simultaneous activation of the tools, the management of the heterogenous data formats and data correlation. In the case of EMBera, a simple time-related correlation has been sufficient and easy to put into practice. However, multi-level observation needs information about causal relations among system events whose discovery, maintenance and management are subject to research [140, 4, 52].

It is our conviction that observation should be configurable and, most of all, well targeted. In all EMBera use cases, the needed observation data has been clearly identified before defining and instantiating EMBera observation components. The fact to capture uniquely the needed information has greatly reduced the data volume to manage and facilitated data understanding and correlation. Our conclusion from the EMBera experience is that there should be a two-level instrumentation mechanism. There should be a base level ensuring access to different types of data at minimal cost and a higher configuration level taking care of the observation targets, parameters and correlation issues.

EMBera does not manage intrusion explicitly. Different experiences have produced different intrusion results and span from 7% to 1000% slow down. We consider that intrusion is acceptable as far as it does not change the behavior of the target system. However, as this is complex to verify, it is our belief that the target system should be dimensioned with observation-dedicated resources. As for the cost of EMBera components, it may be optimized by flattening their runtime structure and thus preventing indirection overhead.

Even if EMBera has focused on post mortem processing, its concepts may very well be applied to online activities. The features of selective observation and configuration may be used to find the reasonable trade-off between observation quality and overhead. Indeed, if the observed information is not sufficient or not well targeted, the observation process could be reconfigured online so as to capture more relevant data.

3 Nondeterministic Error Debugging with ReDSoC

One major problem with interactive debugging is the difficulty to track nondeterministic errors. Indeed, there are, for example, *(*Heisenbugs) i.e bugs that disappear when interactive debugging is *on*. In more general terms, in nondeterministic systems, factors such as the system load, the number of concurrently executing entities, the temperature or the the hardware components' age cause system execution to take different execution paths, to produce different outputs and to bug in different ways. Moreover, repeated execution of the system does not guarantee the reproduction of previous behavior.

There are two approaches to tackle the problem: either nondeterminism is made impossible via adapted hardware, runtime or programming mechanisms [38, 20, 21, 24], or debugging is done *post factum*. The idea is to trace an execution which exhibits a nondeterministic error and then use the trace as a support for debugging. Debugging thus targets not a live execution but an execution replay. A major advantage of this approach is *backward* debugging in which the information about the recorded buggy behavior is used as a starting point for the debugging analysis [77].

3.1 Deterministic Record-Replay (DRR)

A nondeterministic system is a system which may follow different execution paths when executed with the same data input [117]. The main causes are data inputs, scheduling, data races, interruptions and distributed communications.

If we picture a system as a multi-layer stack (cf. Figure 3.1), nondeterminism concerns all levels but is easier to find at lower levels. Indeed, if layer i enforces a deterministic behavior and layer $i+1$ is based entirely on the interfaces provided by i, then $i+1$ is also deterministic. There are no guarantees about the layer $i-1$. For example, the *dOS* system [77] enforces determinism upon process groups at the operating system level. Thus, above *dOS*, all sources of nondeterminism such as scheduling or conflicting shared memory accesses are eliminated. However, operations involving non controlled operations such as accesses to physical resources accesses or distributed communications, stay nondeterministic.

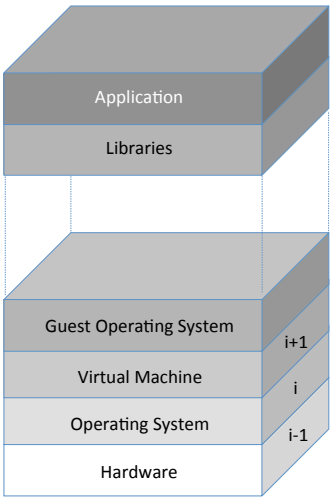


Figure 3.1 – An Example of a Multi-Layered System: All layers are possibly nondeterministic

The idea of deterministic record-replay is to record a system's execution and then deterministically replay the record in order to examine the system's behavior. The *record phase* needs to produce an execution trace containing all the necessary elements reflecting and allowing the reproduction of the system execution. The record phase may be executed several times in order to capture some target abnormal behavior. The *replay phase* replays the execution under the constraints defined by the captured execution trace. The replay may re-execute the system or simulate its execution.

There are numerous DRR solutions which differ in their target application domains, implementation and performance [161]. Indeed, there are DRR proposals in the domains of distributed systems, shared memory systems and embedded systems. They evolve chronologically from simpler, mono-processor systems [119, 30], to more complex architectures such as modern multi-core platforms [81, 145, 96]. Numerous projects focus on data races [116, 40, 66, 97, 110, 95, 16, 114] while others have a more global approach [81, 32]. At the implementation level, DRR solutions are hardware- and/or software-based [109, 64] and may work on simulated [66, 110] or real platform [136, 108, 145, 54] environments. DRR optimizations are various and include efficient log management, system slicing and parallel replay techniques.

MPSoC systems are subject to all cited sources of nondeterminism. Indeed, the numerous peripherals are sources of hardware nondeterminism. As for the software level, data races, scheduling nondeterminism and nondeterministic network communications come as a natural consequence of the increasing number of processors and the introduction of NoCs.

Even though MPSoC systems are subject to all sources of nondeterminism, there are few DRR proposals [88, 54, 31] and they all focus on hardware interrupts and address single mono-core processor platforms. The RedSoC system pushes the effort of DRR for embedded systems further, by considering MPSoC architectures and working on a larger set of sources of nondeterminism.

3.2 ReDSoC: A DRR-Debugger for MPSoC

RedSoC proposes a general debugging methodology for MPSoC systems. It focuses on DRR for shared data accesses, network communications and I/O operations. Its approach is thus complementary to related works focusing on interruption replay.

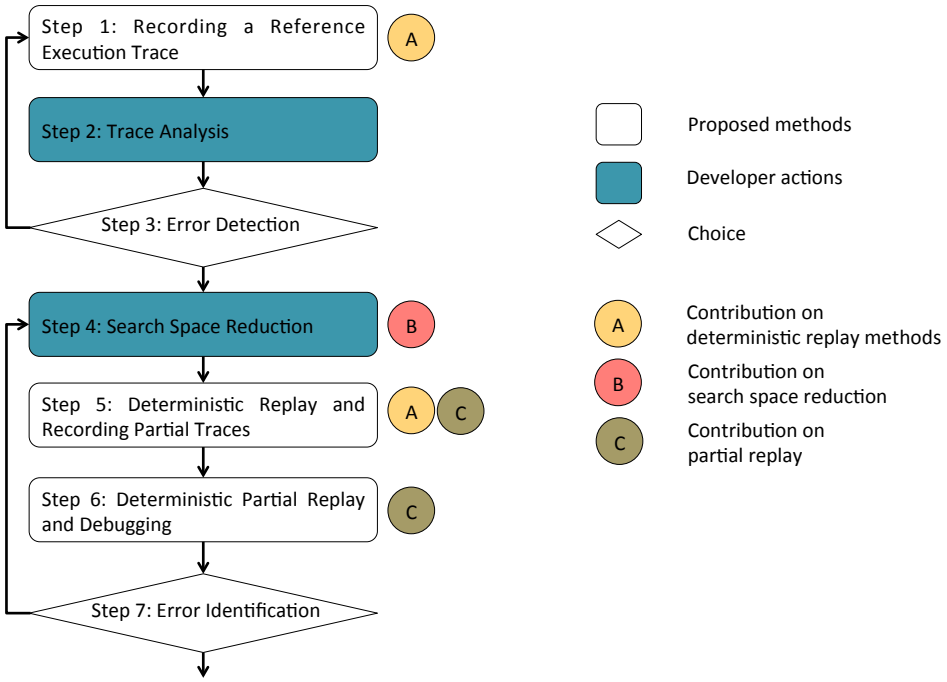


Figure 3.2 – Debugging Cycle

The ReDSoC debugging cycle is shown in Figure 3.2. During *Step 1*, the execution of the whole MPSoC software is recorded to produce reference execution traces. During *Step 2*, the developer analyzes the reference traces in search of abnormal behavior. At *Step 3*, the developer decides whether a problem has been recorded and should be investigated, in which case the cycle continues with Step 4. Otherwise, typically if a targeted nondeterministic error has not yet been recorded, the cycle may restart with Step 1. During *Step 4* the developer decides to focus on a particular part of the software execution thus reducing the error search space. To do so, he/she selects a suspected part of the application to debug during a specific time interval. *Step 5* deterministically replays the reference trace to capture additional data reflecting the execution of the selected software part. *Step 6* deterministically replays the selected software part with the possibility for standard debugging. At *Step 7*, if the error source is not identified, the developer goes back to Step 4.

The architecture of RedSoC is given on Figure 3.3. ReDSoC considers standard debugging configurations including a host platform connected to a target MPSoC platform. It is composed of four tools, namely a trace visualization tool, a partial replay tool, a trace collection tool and a deterministic replay tool.

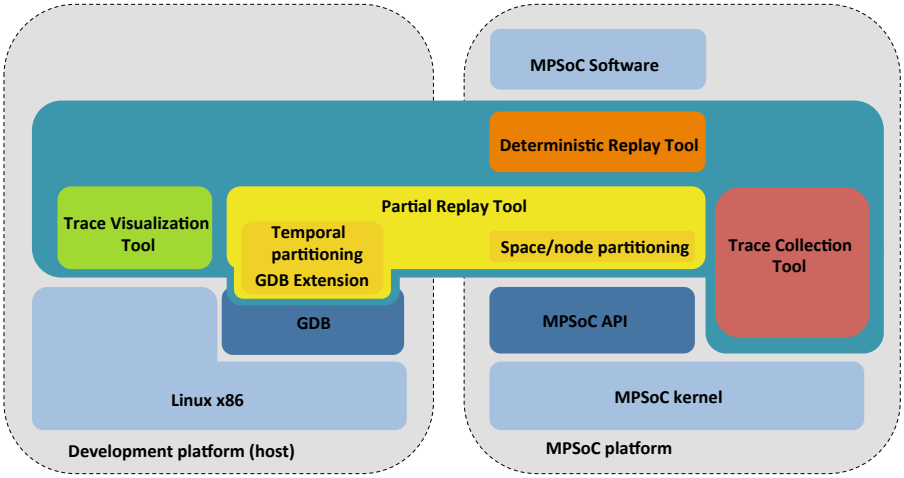


Figure 3.3 – ReDSoC Architecture

The trace collection tool is in charge of capturing nondeterministic events and generating the corresponding execution traces. Trace capture is based on interception of the calls to a predefined MPSoC API which is POSIX-inspired and features operations for task management, synchronization, network communication and I/O.

The tool for deterministic replay enforces DRR through proven deterministic replay algorithms concerning synchronization operations [84], network communications [100, 33] and I/O [118].

The partial replay tool monitors the replay phase to decide which operations are relevant to the selected (suspected) software part. To apply the time reduction criterion, we have implemented an extension for GDB to introduce *replay breakpoints* corresponding to the limits of the time interval that has been selected for debugging.

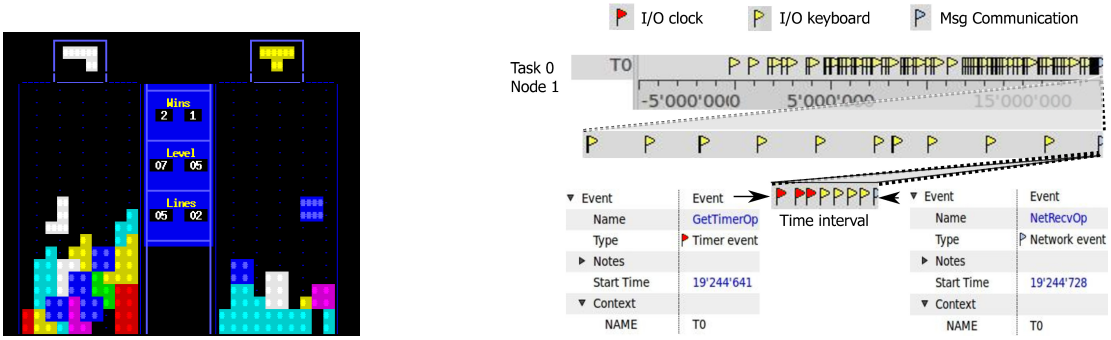
To visualize traces, RedSoC uses the Pajé [106] trace format and has adapted the KPTrace Viewer of STMicroelectronics [129].

In terms of performances, ReDSoC shows a very low intrusion and small trace logs in all considered cases of multimedia application debugging.

One example of successful application of ReDSoC is in a use case of a Tetris game for two players (cf. Figure 3.4a), running on a Stagecoach expansion board with two OveroFE COM nodes[1]. To investigate a nondeterministic crash, ReDSoC obtains a reference trace containing the error. The important number of execution events, however, does not allow the immediate identification of the problem. Being the node to fail, node 1 is chosen as a target for the partial replay. The time interval to debug is chosen to contain its last execution events (cf. Figure 3.4b).

After deterministically replaying the whole application and gathering additional traces about the communications between node 1 and node 2, RedSoC uses a standard debugging session to investigate the problem (cf. Figure 3.5).

[1] https://store.gumstix.com/index.php/products/247/



(a) Two Player Tetris.

(b) Zoom on the last execution events of the failing Tetris node

Figure 3.4 – Debugging a Multimedia Application

The session clearly identifies the trace entry with its number (202459), type (IO), node identifier (Node1) and task identifier (Task0) (line 1). The `bt` GDB shows the interaction between the GDB server and our GDB extension. Up the call stack, we see the replay function for IO operations (`replayIOsize`) and the MPSoC function calls.

The debugging session shows that the crash is due to an incorrect value sent by the other Tetris node which in turn is suspected and debugged. Examining its execution detects non regular behavior and a buffer overflow problem.

Reaching the first replay breakpoint, start of the selected time interval

```
Stopped at replay breakpoint #1 at Id=2.02459 Type=IO, Task=0, Node=1
Last trace info: IO#2.02459, Task=0
[Switching to Thread 0x7f102a2bd700 (LWP 30239)]
(gdb) bt
```

Interaction between the GDB server, executed on the MPSoC and our GDB extension, executed on the host

```
#0  rdb_notify_event () at replay_db.c:11
#1  0x00007f102cab76d0 in rdb_notify_syscall (id=2.02459, type=IO, tid=0, node=1)
    at replay_db.c:24
```

Deterministic replay

```
#2  0x0000000000412812 in replayIOsize ()
    at /replay_mechanism/src/replay.c:146
```

MPSoC software function calls

```
#3  0x000000000041e6c4 in gettm (a=0)
    at /game/src/timer.c:88
#4  0x0000000000415700 in play_round ()
    at game/src/2p.c:506
#5  0x0000000000415c6b in startgame_2p ()
    at /game/src/2p.c:570
#6  0x0000000000419443 in startgame ()
    at /game/src/game.c:115
#11 0x00007f102cf788ba in start_thread () from /lib/libpthread.so.0
#12 0x00007f102c82502d in clone () from /lib/libc.so.6
#13 0x0000000000000000 in ?? ()
```

Figure 3.5 – Partial Debugging of the MPSoC Tetris Application

3.3 Conclusions and Research Perspectives

With the increasing scale, complexity and nondeterminism of computing systems, deterministic record replay (DRR) has recently regained interest as a promising solution to software design and debugging. Applied in various contexts, DRR targets different sources of nondeterminism and proposes different trade-offs between performance and precision. In the domain of embedded systems, however, its application has been limited and has primarily considered the record and replay of interrupts.

ReDSoC is a software-level DRR solution targeting MPSoC and multiple sources of nondeterminism. Considering a generic hardware model of MPSoC systems and standard API for embedded applications, it defines a debugging methodology applying space and time reduction criteria to the error search space. The ReDSoC tools facilitate human comprehension as they are able to focus on a specific part of the target software and consider a limited time interval. ReDSoC has been implemented in real experimental platforms including an embedded system and a multicore NUMA system. It has been successfully used to debug several multimedia applications.

Concerning the debugging methodology, the selection of the suspected software parts and the time interval to debug is a delicate issue which for now relies on the developer experience. It would be highly beneficial and interesting to couple the proposed debugging methodology with techniques able to automatically delimit "problem zones". The automatic detection of abnormal behavior may be based on different methods including statistical analysis, data mining, probabilistic prediction evaluations, etc.

The idea of zoom debugging is not new. Indeed, every developer implicitly zooms and dezooms during the analysis of a system. The developer executes an analysis cycle during which he/she decides to focus on a given part of the execution and strives to replay this part and obtain more information. However, in most cases there is no explicit support to guarantee the reproduction of the execution or to delimit the suspected part. The contribution of ReDSoC is to provide a set of tools to facilitate such a debugging cycle. The idea of zooming into an application by considering the different hierarchical levels of its architecture proves to be highly beneficial. However, in most cases and especially in the case of embedded systems, there is a need to explore lower levels of abstraction. The question is, however, how to marry acceptable performance with the possibility to zoom both horizontally and vertically?

ReDSoC uses trace visualization which greatly facilitates the debugging task of the developer. Our belief is that a visual support, representing the execution history of a target system, with the possibility of going back and examining past events beyond the current call stack, becomes a necessary feature for future development environments. The question of trace visualization and the possibility of browsing trace data is related to the hot topic of data visualization [45, 48].

Our proposal is independent from execution platforms as it is based on a general model for MPSoC and an MPSoC API. However, task-based programming models are not the only ones used in the embedded system domain. We think that the future of debugging techniques is to consider higher levels of the application stack and namely the used programming models.

The developer needs to be able to work in a top-down approach, starting by the human-comprehensive application entities and interactions before going down to operating system details. Some works exist in the domain of interactive debugging [111] but the approach is to be investigated for post-mortem analysis.

4 Trace Management with Framesoc

Nowadays tracing becomes a major aspect in system performance evaluation and enhancement. However, most existing tracing solutions focus mainly on data collection and not on data exploitation. The goal pursued in the SoC-TRACE project [175] is to research an advanced approach to trace storage and analysis in the domain of embedded systems.

This chapter introduces the major research challenges in trace management and presents the Framesoc open trace-management solution, developed in the context of SoC-TRACE.

4.1 Trace Management Challenges

Trace management includes the aspects of trace collection, trace storage, trace access and trace analysis. If trace collection with minimum execution overhead is well managed in the domain of embedded systems, trace storage and trace access are usually addressed in a proprietary and ad hoc manner. As for trace analysis, most existing tools limit themselves to time-chart visualizations and some basic statistics [171].

It is our belief that an effective trace management solution should provide the following features:

Support for Big Traces One of the major challenges in the current *big data* time is that all types of tools capture important volumes of all kinds of data. Trace management is not an exception. The increasing scale of the execution platforms and the growing complexity of software translates directly in bigger traces. Depending on the domain, they can size from GB to TB. In embedded systems, a short multimedia decoding of several seconds produces a gigabyte trace counting several million of events. Instead of tuning the tracing solution so as to minimize the quantity of captured data, new solutions for trace exploitation should be proposed.

Support for Heterogenous Trace Formats The important variety of tools producing trace data brings the question of their heterogenous trace formats[106, 78, 133, 1, 149, 11, 150]. Indeed, different trace formats are incompatible as they use different data models, data semantics and data organization. The exploitation of trace data is thus possible only in dedi-

cated tools and environments. There should be a more generic approach to trace exploitation preventing multiple partitioned efforts which frequently implement the same set of core functions.

Work with Multiple Traces In embedded systems, it is usual to work with a single trace during a debugging session. However, a single trace is not always representative of the system behavior. Moreover, a trace is characteristic for a given execution software configuration and hardware platform. A trace management system should provide for a trace repository with suitable catalogue functions for traces. The possibility to manipulate multiple traces would facilitate the reuse of the debugging/optimization experience from previous work and help identify execution variations dues to platform differences.

Configurable Set of Analysis Treatments Developers are hindered in their trace exploitation by the variety and the number of tools they need to master in order to analyze the traces in the desired way. In a typical working session with a trace tool, the developer may consult some computed statistics but will mostly examine the visual representation of the trace. To carry more sophisticated or simply different analyses, he/she would need to switch to other tools including ad hoc scripts, statistics tools, other visualization tools, etc. A trace management system should provide a configurable and extensible set of trace analysis treatments. It should be configurable in order to allow the developer to use the ones which are useful to him/her. It should be extensible and allow for integration of new analysis treatments.

Storage of Analysis Results Trace analysis may produce some concrete results or simply give a developer an idea for further investigation. A trace management system should be able to save results or developer annotations together with the initial trace. Not only this may save time in future trace analysis but results may be used to build the history of the trace analysis process.

Trace Analysis Workflow Trace analysis should strive for the maturity of data analysis in other scientific fields such as biology or physics in which data goes through well specified scientific workflows. A trace management system should provide means to sequence different trace analysis treatments and to automate such sequences. It should be possible to consult the way traces have been exploited and to reproduce (globally or partially) the experience.

4.2 Framesoc: An Open Trace Management Infrastructure

Framesoc [181, 173, 177, 170, 179, 178, 180] is an open trace management infrastructure. It is open both because it is open-source and is designed to be extensible. It targets trace manipulation after the trace collection phase.

The originality of Framesoc is expressed in the following features.

- *Generic trace representation and manipulation.* Started as a project dedicated to the embedded system domain, Framesoc has now proven successful in the domains of HPC

and operating systems. Its generic trace manipulation facilities allow the integration of traces from different formats and are the basis for collaboration between different trace analysis tools.

- *Core set of trace analysis treatments.* Inspired by the functionalities of existing trace manipulation tools in the domains of embedded, HPC and operating systems, Framesoc identifies and provides a common set of statistics and visualization features. Given the generic trace representation of Framesoc, these features may be applied to traces originating from various domains. Moreover, the implementation is done in the Eclipse [41] environment which has become a *de facto* standard for development.

- *Advanced Trace Manipulation Features.* The last developments of Framesoc consider work with big traces and the possibility of interactive manipulation of partial trace data.

The above points are presented more in detail in the following.

4.2.1 Framesoc Architecture

The Framesoc architecture is presented in Figure 4.1.

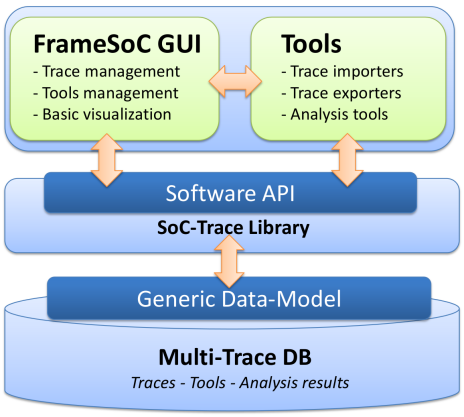


Figure 4.1 – Framesoc Architecture

The bottom storage layer keeps initial traces, as well as trace analysis results. Among the trace formats currently handeled by Framesoc are CTF[1] [149], OTF[2] [78], Pajé [106], KPTrace [133] and gstreamer [59].

The middle layer contains a library ensuring the interface between the storage and the tools working with traces.

The top level is composed of tools for trace manipulation. Currently Framesoc provides basic visualization and statistics tools and integrates a data mining tool by UJF/HADAS [75], a

[1] This format is used in LTTng, a major solution for tracing Linux

[2] This format has become a reference trace format in the HPC domain. It is promoted in the Score-P [124] project and used in the Vampir [143] and Scalasca [120] tools.

statistics tool by ProbaYes [113], a KPTrace viewer by STMicroelectronics and an aggregation visualization tool [39].

Framesoc is part of the SET1.0 product of the SoC-TRACE project and as such has been publicly released since mid 2014 [3].

Framesoc itself is publicly available as a GitHub project [4]. The site provides un up to date vision of the latest developments and features of the framework.

4.2.2 Generic Trace Representation

To tackle the problem of format heterogeneity, we propose the use of an innovative generic data-model for traces [179](Figure 4.2). The model addresses trace metadata, trace raw data, analysis results and tools metadata.

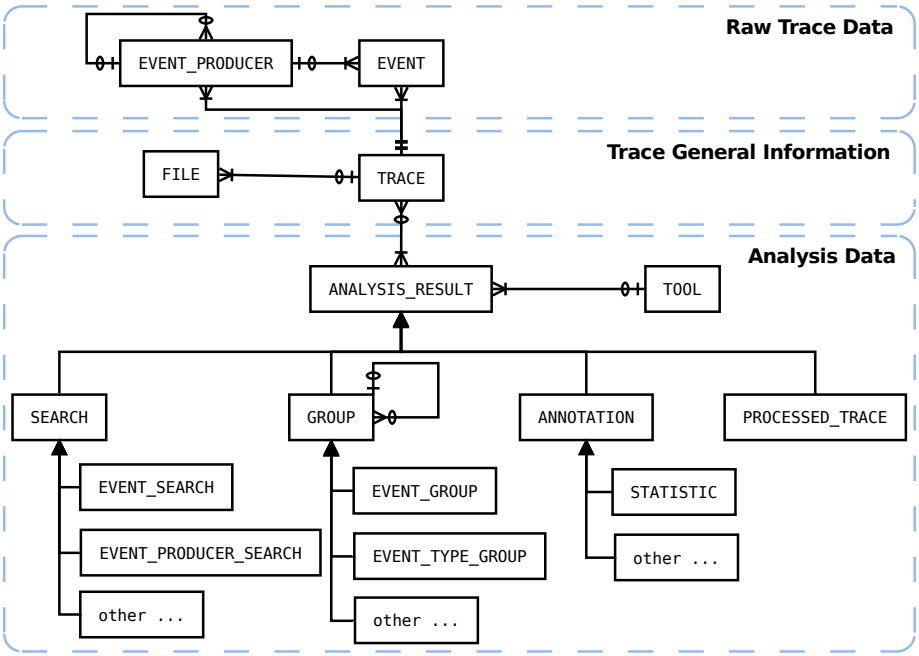


Figure 4.2 – Generic data-model for trace management (Crow's Foot notation)

Events and traces are modeled using the self-defining pattern illustrated at Figure 4.3 for events. This pattern allows to reflect both the type and the corresponding values of an entity. The attributes of the **Event** entity define the general event characteristics i.e the characteristics that are common to all system events. The information contained in **EventParam** entities is *custom data* which reflects the differences in the events' structure and semantics.

To reveal some of the system semantics and facilitate its access and exploitation, Framesoc events have been enriched to reflect the notions of punctual event, system state, link (causal

[3] http://soc-trace.minalogic.net
[4] http://soctrace-inria.github.io/framesoc/

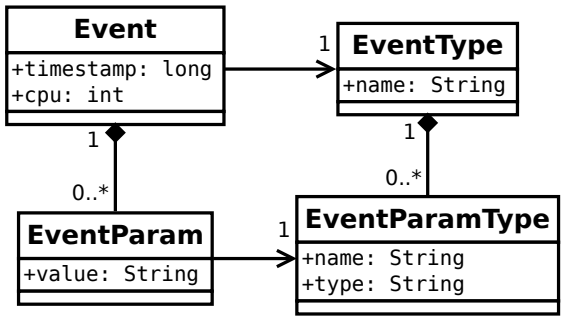
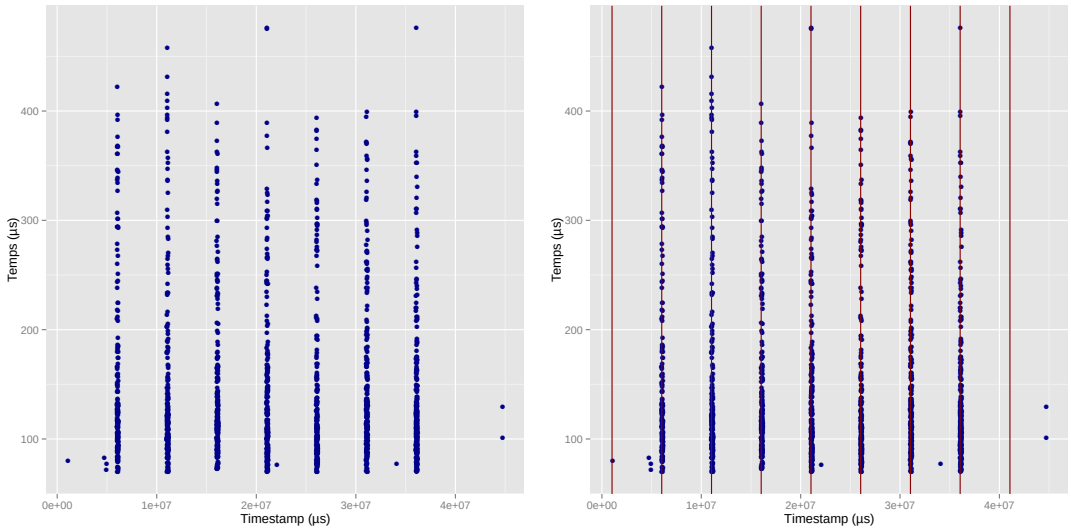


Figure 4.3 – UML diagram of the self-defining pattern (*EVENT* entity)

relation) and variable [173]. These generic notions are widely accepted and used in numerous trace exploitation tools, either at the analysis or the visualization level [107, 120, 143, 129, 72]. If in existing tools the information characterizing these notions is part of the trace analysis process, in Framesoc, we put it as a foundation on which higher-level trace analyses may be executed.



(a) Problematic calls to *softIRQ*

(b) Correlation with calls to *flush*

Figure 4.4 – Example of trace analysis using the *state* notion

In a real use case provided by STMicroelectronics concerning a buggy multimedia decoder, the knowledge of the above notions has helped to accelerate the identification of the problem. Instead of considering all the data contained in the raw trace, the analysis has mainly considered the generic event characteristics. Not only the data retrieval has been dozens times faster but the analysis has been facilitated by the manipulation of less data with explicit semantics. Using the *state* notion, we have identified abnormally long executions of the `softIRQ` function (Figure4.4a) and have correlated them with to calls of the `flush` function (cf. Figure 4.4b) The correction of this result has been confirmed by STMicroelectronics developers.

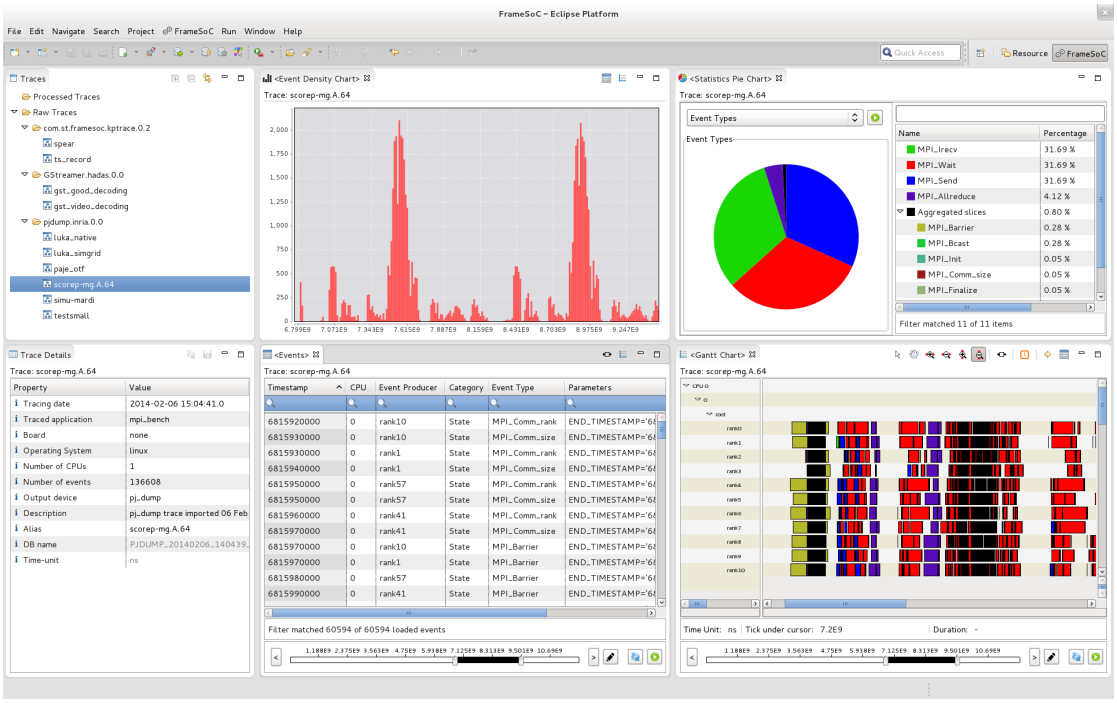


Figure 4.5 – The Framesoc Workbench

4.2.3 Extensible Set of Correlated Views

Framesoc provides a set of views which are intuitive to use and which support simple interactions for trace data analysis. The set of views is not predefined and allows for the integration of new types of views. During a working session, the user may choose the views that are useful for his/her work and have multiple instances per type of view.

Framesoc views are consistent and represent the same trace data, during the same time interval and with the same color code. To this purpose, it provides a skeleton for view development, a Publish-Subscribe architecture for inter-view communication and . view controllers taking care of group membership and naming issues.

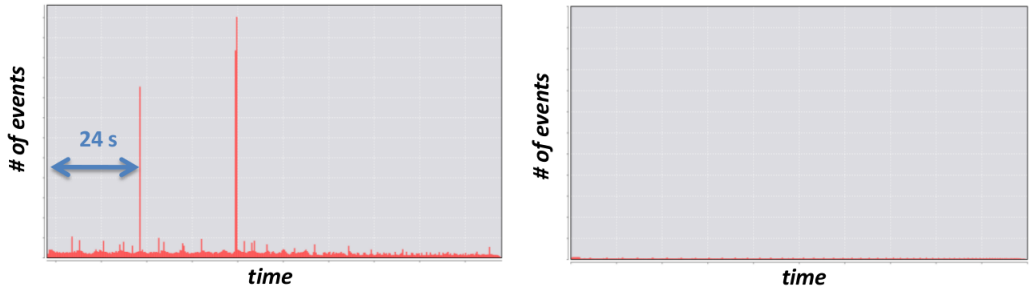
Framesoc (Figure 4.5) includes management views (on the leftmost column) and analysis views (in the main workbench body). Management views include a trace browser and a trace metadata viewer/editor. The first lists the available traces in the system, grouped by format, while the second shows the metadata of the currently selected trace(s).

Analysis views include an event density chart and a statistics pie-chart, on top, and a table of events and a Gantt chart, at the bottom. The event density chart shows the distribution of all events on the scale of the whole execution captured in the trace. The statistics pie-chart gives information about the number of event occurrences compared to the total number of events. It may characterize the event types or the event producers. The table of events gives access to the raw event data and allows for querying using regular expressions. Finally, the Gantt chart shows the execution history as recorded in the trace. In 2014, the Gantt has evolved to

use the open source TraceCompass viewer [125], originating from the LTTng project. This development has greatly improved the performance of the trace visualization and has created collaboration possibilities with a rich open-source community.

The usefulness of Framesoc is illustrated in the following use case comparing a real world platform execution with a simulated version [128]. It compares a native trace, issued by executing a parallel application on a real system containing 4 GPU and a simulated trace, obtained by running the same application within the Simgrid simulator [10].

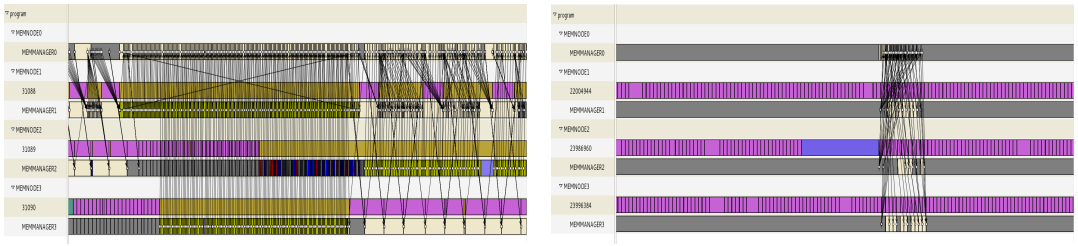
Visualizing an overview of the two traces, using an event density chart per trace with the same time scale (Figure 4.7a and Figure 4.7b) shows that the traces are different. Indeed, the native trace has more than 3 millions of events, while the simulated one has only about 900 thousands of events.



(a) Native trace (b) Simulated trace

Figure 4.6 – Event-density chart

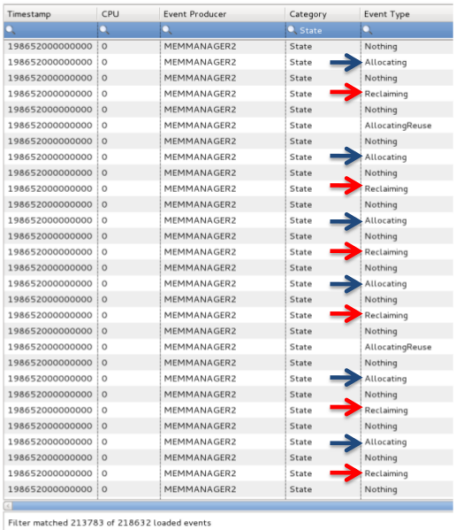
A zoom around the first peak (24s) on both traces and opening the Gantt-chart representations gives Figure 4.7. The native trace has a lot more communications among the different processes and a lot more state changes, mostly in the central part of the time interval.



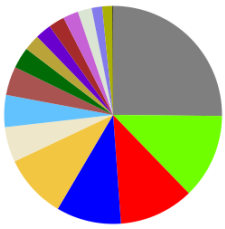
(a) Native trace (b) Simulated trace

Figure 4.7 – Trace interval centered on the first event peak (24 s)

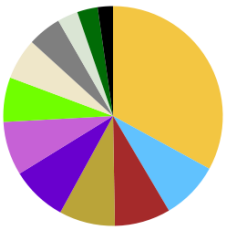
The table view of the central part contains a pattern: the sequence of states *Allocating* and *Reclaiming*, highlighted in blue and red respectively.(Figure 4.8a). The event-type statistics pie chart for the native trace gives the information that these events represent actually about 20% of all trace events (Figure 4.8b, blue and red pies). However, they are not at all present in the



(a) Event table showing a pattern



(b) Native trace: the blue-red pattern takes 20%



(c) Simulated trace: blue and red events are missing

Figure 4.8 – Missing event pattern in a simulated trace

simulated trace. Indeed, we find that the simulator considers GPUs to have infinite memory therefore ignoring all RAM swapping operations.

4.2.4 Interactive Partial Trace Manipulation

A major point that has not been explicitly addressed in the SET1.0 product is the management of big traces. Indeed, in SET1.0, the user may find himself/herself waiting for quite a long time to finally fail to load a trace. Another possible situation is to successfully load the trace but wait for an analysis result. The user may get frustrated as during a long analysis treatment, SET1.0 does not provide any feedback.

A step forward, proposed by the latest version of Framesoc, is to allow a partial manipulation of a trace. All the Framesoc tools have evolved to allow an incremental data manipulation with immediate visual feedback. This means that traces are loaded by portions and that the visual representation is updated on the fly. With this feature not only the user may follow he computing process but he/she may start manipulating the available data. If there are not enough system resources or the treatments are taking too long, the user may stop the process without blocking or failing the system.

Figure 4.9 illustrates this feature with the Gantt representation of a trace. At the bottom, the black bar and an explicit note indicate that the trace is partially (29%) loaded. On the right top corner, the Gantt indicates that it shows only 42.2% of the available information to not saturate the representation.

Indeed, when representing a whole trace, if there is no explicit management of what is visualized, not only the result is nondeterministic depending on the graphics card but it may be totally useless, as shown in Figure 4.10

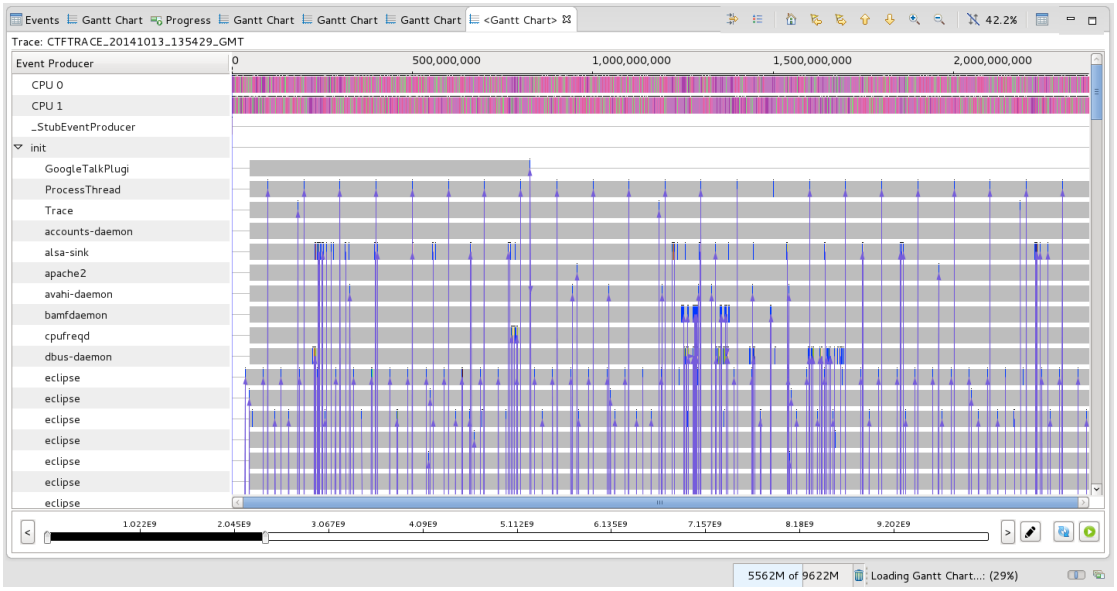
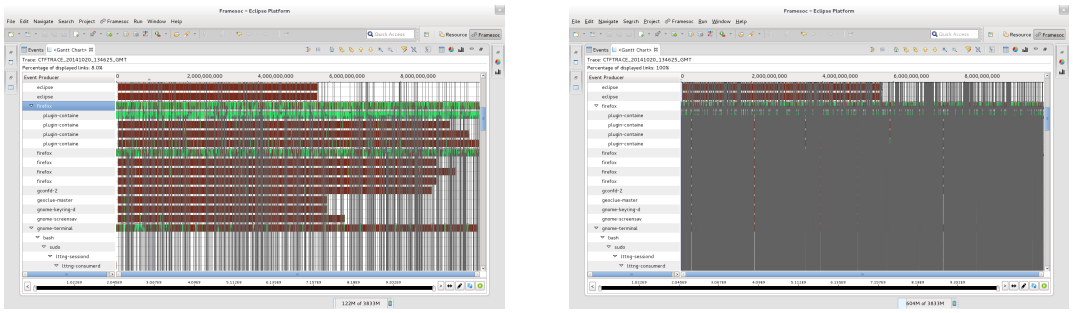


Figure 4.9 – Partial trace loading and visualization



(a) Filtered Visualization

(b) Non Filtered Visualization

Figure 4.10 – Filtered visualization

With its partial manipulation of traces, Framesoc has succeeded in working with a $12GB$[5] trace including about 200 million of events. The importation of such a trace into the Framesoc database takes about 2,7 hours but thanks to partial trace manipulation the first results are provided to the user in seconds. Loading the whole trace would have required a huge memory capacity and about 30 hours. A small trace of 1 million events, comparable in size to standard traces provided by STMicroelectronics in the SoC-TRACE context, takes $13s$ to be imported and several seconds to provide first results to the user.

4.3 Conclusions and Research Perspectives

The increasing complexity of computing systems in general, and embedded systems in particular, calls for new techniques and tools for system development. Trace collection and analysis,

[5] trace size in terms of Framesoc storage

a classic and necessary means for understanding and enhancing the behavior of computing systems, is unfortunately still developed in an ad hoc manner. Framesoc aims at defining an open and generic trace management infrastructure. It handles different trace formats using a generic trace model capturing all base notions commonly found in traces. Concerning trace manipulation operations, Framesoc provides an extensible kernel for trace access and statistic profiles computations. For more sophisticated trace analysis treatments, Framesoc provides an integration framework for tools which can both access and produce traces and trace analysis results. Framesoc facilitates the work of developers by providing a graphical environment (an Eclipse IDE) and addressing the issues of interactive work with big traces.

The points of interest for future research are the following.

Framesoc performances are strongly related to the issues of data management including persistent storage and memory management. The current solution of persistent storage based on freely available DBMS (SQLite or MySQL) limits the performance of trace access operations. Other storage solutions including advanced SQL DBMS, such as Oracle, or noSQL platforms are to be investigated.

Another perspective for enhanced data manipulation performance is the management of data caches containing trace or analysis results information. However, generic data cache management and cache sharing among different trace manipulation tools is a challenge. Indeed, even if low-level trace notions, such as events, states and links are shared among tools, different trace manipulations usually work with different higher-level semantics. In the case of the SET product, for example, the Ocelotl visualisation tool works with notions of time and space aggregation, the FrameMiner data mining tool works with frequent patterns and the MegaLog probabilistic tool manipulates sequences. All three tools have different internal data structures and therefore would need different, per tool, caches. To be able to manage a shared data cache, different tools should have common higher-level trace representations. Such representations, still to be researched and defined, would also help the definition of more advanced collaborations among trace analysis tools.

To help the user gain more control on trace analysis and not limit him to point-and-click interfaces with predefined operations, it is my belief that there should be workflow solutions for trace analysis. Such a solution is to provide the user with the possibility to choose which analysis treatments to apply, to define how to sequence them and to decide what to do with the results. Moreover, a workflow support will help to automate and reproduce the experience, either to verify results or to apply (reuse) it to a different use case.

The major developer of Framesoc is Generoso Pagano, an engineer working for the SoC-TRACE project and actively collaborating with engineers from the SoC-TRACE partners. His work on Framesoc brought him to become a contributor to the TraceCompass (ex LTTng viewer) project. The aspects about trace analysis workflow are investigated by Alexis Martin in the context of his Ph.D.

5 Conclusion and Perspectives

Observation management is closely related to the "Big Data" question we are facing today. Embedded systems, like all others, increase their scale, count more components, are managed by more complex software. The quantity and diversity of data produced and manipulated during system operation explodes. We research today new ways of organizing, filtering and analyzing the data in order to detect interesting behaviors and make them explicit to the human.

It is my belief that the effervescence in the embedded system domain will evolve towards clearer software layered architectures and standardized interfaces. The observation features will become an integral part of embedded system design, not only at the hardware but also at the software level. This will come along with formal frameworks establishing provable relationships between embedded components, with their functional properties and performance, on one hand, and observed data, on the other.

Three challenging aspects I am interested in are:

- *Goal-Driven Observation Configuration*

  All aspects of my work have shown the importance of configuring the observation process in order to capture and analyze *relevant* data. EMBera has shown the importance of observation configuration to minimize intrusion. ReDSoC configures the partitioning of the system for focused debugging. Framesoc applies analysis treatments on a pre-filtered set of data.

  In most works, the capture of execution-related data is performance-driven i.e strives for intrusion minimization. The goal is to generate maximum data with minimum perturbation. However, this is done without a prior definition of what the relevant data should be in relation to the pursued observation goal.

  In the domain of embedded systems where performance is a critical issue, the trend is to relegate the capture of observation information to the hardware level. Embedded design evolves towards over-dimensioned architectures which include electronic components

for computation, as well as components for observation and control. The impact of such an architectural shift is that collecting data about a system's execution will come with a zero cost. The important question is then, if it is possible to observe everything about a system, what is really useful to observe? There is a need to establish a relation between the properties of the system we are interested in, the system entities and actions that define these properties and the respective observational data.

- *Open Framework for Observation Data Analysis*

  To analyze the data from a system execution, the user has two possibilities. He/she can either use an existing observation tool or implement an ad hoc analysis treatment. In the first case, the user is faced with the challenge of choosing a suitable tool which can both manage the data coming from the target system and provide the required data analysis treatment. If existing tools do not meet his/her needs, the user may develop his/her own analysis treatment. However, the user will be faced with the complexity of data management and computation optimization. The resulting treatment will most likely be specific to the user case, difficult to evolve and impossible to reuse in a different application context.

  A challenging research direction, explored in the context of Framesoc, is to define an open analysis framework. The idea is to make possible the reuse and recombination of data analysis treatments. The reuse aspect is to be ensured by a data analysis library proposing a core set of data analysis treatments. This work implies the study, reverse engineering and re-engineering of data analyses to be found in existing toolsets (e.g Scalasca [120], Vampir [143], LTTng [151], etc.). The library is to be open and allow for addition of new analysis treatments. The recombination aspect aims at the possibility to construct multi-step data analyses i.e analysis workflows.

- *Online Analysis*

  In the domain of embedded systems, the classical approach to observation is to capture execution-related data and analyze it a posteriori. However, with the emergence of *IoT*, the ubiquity of embedded systems and the possibility to generate observational data at zero cost, the quantity of observational data explodes. As only a fraction of this data is relevant for further use, the captured observation data should be analyzed and minimized online i.e while it is being produced. The challenge is thus to find new ways of resolving the two previous issues but under the constraints of live executions. In other words, observation configuration and data analysis should be able to take into account the dynamics of the execution environment, adapt to available computational resources and manage incomplete observations.

References

[1] A.Chan and al. An Efficient Format for Nearly Constant-Time Access to Arbitrary Time Intervals in Large Trace Files. *Scientific Programming*, 16(2-3), 2008.

[2] OSGi Alliance. The OSGi Architecture. http://www.osgi.org/Technology/WhatIsOSGi.

[3] JoseBacelar Almeida, MariaJoao Frade, JorgeSousa Pinto, and Simao Melo de Sousa. An overview of formal methods tools and techniques. In *Rigorous Software Development*, Undergraduate Topics in Computer Science, pages 15–44. Springer London, 2011.

[4] Lorenzo Alvisi, Karan Bhatia, and Keith Marzullo. Causality tracking in causal message-logging protocols. *Comput*, 15:1–15, 2002.

[5] Sihem Amer-Yahia, Noha Ibrahim, Christiane Kamdem Kengne, Federico Ulliana, and Marie Christine Rousset. SOCLE: towards a framework for data preparation in social applications. *Ingénierie des Systèmes d'Information*, 19(3):49–72, 2014.

[6] Building Your First Android App. http://developer.android.com/training/basics/firstapp/index.html?hl=p.

[7] Apple. Start Developing iOS Apps Today . https://developer.apple.com/library/ios/referencelibrary/GettingStarted/RoadMapiOS/.

[8] Arduino Open-Source Electronics Platform. http://arduino.cc/.

[9] ARM. ARM CoreSight. http://www.arm.com/products/system-ip/debug-trace/trace-macrocells-etm/index.php.

[10] SimGrid. http://simgrid.gforge.inria.fr/.

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

[12] Mike Barlow. Real-Time Big Data Analytics: Emerging Architecture. http://www.oreilly.com/data/free/big-data-analytics-emerging-architecture.csp.

[13] Michael Barr and Anthony Massa. *Programming embedded systems - with C and GNU development tools: thinking inside the box: includes real-time and Linux examples (2. ed.)*. O'Reilly, 2006.

[14] M. Bartlett, I. Bate, and J. Cussens. Learning bayesian networks for improved instruction cache analysis. In *Machine Learning and Applications (ICMLA), 2010 Ninth International Conference on*, pages 417–423, Dec 2010.

[15] Len Bass, James Ivers, Mark Klein, and Paulo Merson. Reasoning frameworks. Technical Report CMU/SEI-2005-TR-007, Carengie Mellon University, 2005.

[16] Arkaprava Basu, Jayaram Bobba, and Mark D. Hill. Karma: Scalable deterministic record-replay. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 359–368, New York, NY, USA, 2011. ACM.

[17] Friedrich L. Bauer. From specifications to machine code: Program construction through formal reasoning. In *Proceedings of the 6th International Conference on Software Engineering*, ICSE '82, pages 84–91, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[18] M. Bauer. *Linux Server Security*. O'Reilly Series. O'Reilly Media, 2005.

[19] L. Benini, E. Flamand, D. Fuin, and D. Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 983–987, March 2012.

[20] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: A compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 53–64, New York, NY, USA, 2010. ACM.

[21] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dos. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[22] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe multithreaded programming for c/c++. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 81–96, New York, NY, USA, 2009. ACM.

[23] Danilo Beuche, Abdelaziz Guerrouat, Holger Papajewski, Wolfgang Schroder-preikschat, Olaf Spinczyk, and Ute Spinczyk. The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In *In 2nd IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '99*, pages 45–53, 1999.

[24] Robert L. Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 4–4, Berkeley, CA, USA, 2009. USENIX Association.

[25] Manfred Broy, Anton Deimel, Juergen Henn, Kai Koskimies, Frantisek Plasil, Gustav Pomberger, Wolfgang Pree, Michael Stal, and Clemens A. Szyperski. What characterizes a (software) component? *Software - Concepts and Tools*, 19(1):49–56, 1998.

[26] Éric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The Fractal Component Model and its Support in Java. *Software – Practice and Experience (SP&E)*, 36(11-12):1257–1284, September 2006. Special issue on "Experiences with Auto-adaptive and Reconfigurable Systems".

[27] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley Publishing Company, USA, 2nd edition, 2010.

[28] Tao Chen, Arif Khan, Markus Schneider, and Ganesh Viswanathan. iblob: Complex object management in databases through intelligent binary large objects. In Alan Dearle and RobertoV. Zicari, editors, *Objects and Databases*, volume 6348 of *Lecture Notes in Computer Science*, pages 85–99. Springer Berlin Heidelberg, 2010.

[29] Yunji Chen, Weiwu Hu, Tianshi Chen, and Ruiyang Wu. Lreplay: A pending period based deterministic replay scheme. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 187–197, New York, NY, USA, 2010. ACM.

[30] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT '98, pages 48–59, New York, NY, USA, 1998. ACM.

[31] Siddharth Choudhuri and Tony Givargis. Flashbox: A system for logging non-deterministic events in deployed embedded systems. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1676–1682, New York, NY, USA, 2009. ACM.

[32] Jim Chow, Dominic Lucchetti, Tal Garfinkel, Geoffrey Lefebvre, Ryan Gardner, Joshua Mason, Sam Small, and Peter M. Chen. Multi-stage replay with crosscut. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10, pages 13–24, New York, NY, USA, 2010. ACM.

[33] C. Clemencon, J. Fritscher, M. Meehan, and R. Rühl. An Implementation of Race Detection and Deterministic Replay with MPI. *EURO-PAR'95 Parallel Processing*, pages 155–166, 1995.

[34] Ivica Crnkovic. Component-Based Approach for Embedded Systems. In *Proceedings of Ninth International Workshop on Component-Oriented Programming*, 2004.

[35] Ivica Crnkovic, Severine Sentilles, Aneta Vulgarakis, and Michel R. V. Chaudron. A classification framework for software component models. http://www.idt.mdh.se/kurser/cd5490/2014/lectures/tse_classificationFramework.pdf.

[36] Scalasca : Cube 4.x series. http://www.scalasca.org/software/cube-4.x.

[37] M. Desnoyers and M.R. Dagenais. The LTTng tracer: A Low Impact Performance and Behavior Monitor for GNU/Linux. In *Proceedings: Ottawa Linux Symposium*, pages 209–224, 2006.

[38] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Dmp: Deterministic shared memory multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 85–96, New York, NY, USA, 2009. ACM.

[39] Damien Dosimont, Robin Lamarche-Perrin, Lucas Mello Schnorr, Guillaume Huard, and Jean-Marc Vincent. A spatiotemporal data aggregation technique for performance analysis of large-scale execution traces. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pages 149–157, Sept 2014.

[40] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 121–130, New York, NY, USA, 2008. ACM.

[41] Eclipse. http://www.eclipse.org.

[42] A (Non-Exhaistive) List of Embedded Operating Systems. http://en.wikipedia.org/wiki/List_of_operating_systems#Embedded.

[43] ENTERPRISE MANAGEMENT ASSOCIATES® (EMA™). Essential IT Monitoring: Ten Priorities for Systems Management. http://content.solarwinds.com/Creative/pdf/Whitepapers/EMA-SolarWinds_ITMonitoring_Systems-0913-WP.PDF.

[44] Jean-Philippe Fassino. Nomadik Multiprocessing Framework, a Component-based Programming Model for MP-SoC,. In *International Forum on Application-Specific Multi-Processor SoC*, 2007.

[45] J.-D. Fekete. Visual Analytics Infrastructures: From Data Management to Exploration. *Computer*, 46(7):22–29, July 2013.

[46] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. *Proceedings of 16th ACM Symposium on Operating Systems Principles (SOSP)*, 1997.

[47] Eran Gabber, Christopher Small, John Bruno, José Brustoloni, and Avi Silberschatz. The Pebble Component-based Operating System. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 20–20, Berkeley, CA, USA, 1999. USENIX Association.

[48] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[49] Jack G. Ganssle. *The art of designing embedded systems*. Newnes, Burlington (US), Oxford (GB), 2008.

[50] Hubert Garavel and Radu Mateescu. Seq.open: A tool for efficient trace-based verification. In Susanne Graf and Laurent Mounier, editors, *SPIN*, volume 2989 of *Lecture Notes in Computer Science*, pages 151–157. Springer, 2004.

[51] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 27–27, Berkeley, CA, USA, 2006. USENIX Association.

[52] Elena Giachino, Ivan Lanese, and ClaudioAntares Mezzina. Causal-consistent reversible debugging. In Stefania Gnesi and Arend Rensink, editors, *Fundamental Approaches to Software Engineering*, volume 8411 of *Lecture Notes in Computer Science*, pages 370–384. Springer Berlin Heidelberg, 2014.

[53] Jean-Philippe Gouigoux. *Practical Performance Profiling: Improving the Efficiency of .NET Code*. Red gate books, United Kingdom, 2012.

[54] Giovani Gracioli and Sebastian Fischmeister. Tracing and recording interrupts in embedded software. *J. Syst. Archit.*, 58(9):372–385, October 2012.

[55] Green Hills. Green Hills Probe™. http://www.ghs.com/products/probe.html.

[56] Steve Griffith. On-chip debug units maximize real-time embedded systems.

[57] Khronos Group. The open standard for parallel programming of heterogeneous systems (OpenCL).

[58] Object Management Group. OpenCCM. http://www.omg.org/technology/documents/formal/components.htm.

[59] gstreamer: Open Source Media Framework. http://gstreamer.freedesktop.org/.

[60] Gumstix. Gumstix: Software and Development. https://www.gumstix.com/software/.

[61] C. Hamacher, Z. Vranesic, S. Zaky, and N. Manjikian. *Computer Organization and Embedded Systems*. McGraw-Hill Education, 2011.

[62] S. Heath. *Embedded Systems Design*. Elsevier Science, 2002. embedded.

[63] Damien Hedde and Frederic Petrot. A Non Intrusive Simulation-Based Trace System to Analyse Multiprocessor Systems-on-Chip Software. In *IEEE International Symposium on Rapid System Prototyping*, pages 106–112. IEEE, May 2011.

[64] Ahmad Heydari and Saeed Azimi. A survey in deterministic replaying approaches in multiprocessors. *International Journal of Electrical & Computer Sciences IJECS-IJENS*, 10(4), 2010.

[65] Scott Hissam, James Ivers, Daniel Plakosh, and Kurt C. Wallnau. Pin component technology (v1.0) and its c interface. Technical Report CMU/SEI-2005-TN-001, Carnegie Mellon University, 2005.

[66] Derek R. Hower and Mark D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 265–276, Washington, DC, USA, 2008. IEEE Computer Society.

[67] K.A Huck, AD. Malony, R. Bell, and A Morris. Design and implementation of a parallel performance data management framework. In *Parallel Processing, 2005. ICPP 2005. International Conference on*, pages 473–482, June 2005.

[68] Nicholas Hunt, Tom Bergan, Luis Ceze, and Steven D. Gribble. Ddos: Taming nondeterminism in distributed systems. *SIGPLAN Not.*, 48(4):499–508, March 2013.

[69] University of Tenessee ICL. Performance Application Programming Interface (PAPI). http://icl.cs.utk.edu/papi/.

[70] IEEE. 1149.1-2013 - IEEE Standard for Test Access Port and Boundary-Scan Architecture. http://standards.ieee.org/findstds/standard/1149.1-2013.html.

[71] IEEE. IEEE Standard for Reduced-Pin and Enhanced-Functionality Test Access Port and Boundary-Scan Architecture. *IEEE Std 1149.7-2009*, pages c1–985, Feb 2010.

[72] Intel® Trace Analyzer and Collector 8.1. https://software.intel.com/en-us/intel-trace-analyzer.

[73] Kalray. MPPA®: the supercomputing on a chip™ solution. http://www.kalrayinc.com/kalray/products/#processors.

[74] C. Kaner, J.L. Falk, and H.Q. Nguyêñ. *Testing computer software*. VNR computer library. Van Nostrand Reinhold, 1993.

[75] C.K. 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. In *Data Mining Workshops (ICDMW), 2012 IEEE 12th International Conference on*, pages 595–602, Dec 2012.

[76] C.K. Kengne, N. Ibrahim, M.-C. Rousset, and M. Tchuente. Distance-based trace diagnosis for multimedia applications: Help me ted! In *Semantic Computing (ICSC), 2013 IEEE Seventh International Conference on*, pages 306–309, Sept 2013.

[77] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association.

[78] Andreas Knüpfer, Holger Brunst, and Ronny Brendel. Open Trace Format. Specification, Center for Information Services and High Performance Computing (ZIH) Technische Universität Dresden, Germany, November 2011.

[79] R. Konuru, H. Srinivasan, and J.D. Choi. Deterministic replay of distributed java applications. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 219–227. IEEE, 2000.

[80] R. Konuru, H. Srinivasan, and Jong-Deok Choi. Deterministic replay of distributed java applications. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 219–227, 2000.

[81] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. *SIGMETRICS Perform. Eval. Rev.*, 38(1):155–166, June 2010.

[82] R. Lamarche-Perrin, Y. Demazeau, and J.-M. Vincent. The best-partitions problem: How to build meaningful aggregations. In *Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2013 IEEE/WIC/ACM International Joint Conferences on*, volume 2, pages 399–404, Nov 2013.

[83] T.J. LeBlanc and J.M. Mellor-Crummey. Debugging parallel programs with instant replay. *Computers, IEEE Transactions on*, 100(4):471–482, 1987.

[84] L.J. Levrouw, K.M.R. Audenaert, and J.M. Van Campenhout. A New Trace and Replay System for Shared Memory Programs based on Lamport Clocks. In *Parallel and Distributed Processing, 1994. Proceedings. Second Euromicro Workshop on*, pages 471–478. IEEE, 1994.

[85] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the role of burst buffers in leadership-class storage systems. In *In Proceedings of the 2012 IEEE Conference on Massive Data Storage*, 2012.

[86] Lynuxworks. SpyKer: Embedded System Trace Tool. http://www.lynx.com/products/development-tools/spyker-embedded-system-trace-tool/.

[87] Hugh Maaskant. A robust component model for consumer electronic products. In Peter van der Stok, editor, *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, volume 3 of *Philips Research*, pages 167–192. Springer Netherlands, 2005.

[88] Ji Chan Maeng, Jung-Il Kwon, Min-Kyu Sin, and Minsoo Ryu. RT-Replayer: A Record-Replay Architecture for Embedded Real-time Software Debugging. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1670–1675, New York, NY, USA, 2009. ACM.

[89] Diego Melpignano, Luca Benini, Eric Flamand, Bruno Jego, Thierry Lepley, Germain Haugou, Fabien Clermidy, and Denis Dutoit. Platform 2012, a many-core computing accelerator for embedded socs: Performance evaluation of visual analytics applications. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1137–1142, New York, NY, USA, 2012. ACM.

[90] Microsoft COM Model. http://www.microsoft.com/com/default.mspx.

[91] Microsoft. Microsoft .NET. http://www.microsoft.com/net.

[92] MIPI Allience. MIPI Allience. Debug Working Group. http://www.mipi.org/working-groups/debug.

[93] Iso/iec 15444-3:2007, "information technology—jpeg2000 image coding system—part 3: Motion jpeg 2000". http://www.iso.org/iso/catalogue_detail.htm?csnumber=41570, 2012.

[94] Micrium Embedded Software. http://micrium.com/.

[95] Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution ef?ciently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 289–300, Washington, DC, USA, 2008. IEEE Computer Society.

[96] Pablo Montesinos, Matthew Hicks, Samuel T. King, and Josep Torrellas. Capo: A software-hardware interface for practical deterministic multiprocessor replay. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 73–84, New York, NY, USA, 2009. ACM.

[97] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording shared memory dependencies using strata. *SIGPLAN Not.*, 41(11):229–240, October 2006.

[98] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. *SIGARCH Comput. Archit. News*, 33(2):284–295, May 2005.

[99] NASA Langley Formal Methods. http://shemesh.larc.nasa.gov/fm/.

[100] R.H.B. Netzer and B.P. Miller. Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 502–511. IEEE Computer Society Press, 1992.

[101] G. Nicolescu and P.J. Mosterman. *Model-Based Design for Embedded Systems*. Computational Analysis, Synthesis, and Design of Dynamic Systems. Taylor & Francis, 2010.

[102] Tammy Noergaard. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. Newnes, 2005.

[103] NVIDIA. Compute Unified Device Architecture (CUDA).

[104] ObjectWeb. xxx. http://openccm.objectweb.org/doc/index.html.

[105] Opari2. http://wwwpub.zih.tu-dresden.de/~silctest/opari2/user/current/html/.

[106] Paje Trace Format. http://paje.sourceforge.net/download/publication/lang-paje.pdf.

[107] Paraver. http://www.bsc.es/computer-sciences/performance-tools/paraver/general-overview.

[108] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 2–11, New York, NY, USA, 2010. ACM.

[109] G. Pokam, C. Pereira, K. Danne, L. Yang, S. King, and J. Torrellas. Hardware and software approaches for deterministic multi-processor replay of concurrent programs. *Intel Technology Journal*, 13(4), 2009.

[110] Gilles Pokam, Cristiano Pereira, Shiliang Hu, Ali-Reza Adl-Tabatabai, Justin Gottschlich, Jungwoo Ha, and Youfeng Wu. Coreracer: A practical memory race recorder for multicore x86 tso processors. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 216–225, New York, NY, USA, 2011. ACM.

[111] Kevin Pouget, Patricia López Cueva, Miguel Santana, and Jean-François Méhaut. Interactive debugging of dynamic dataflow embedded applications. In *IPDPS Workshops*, pages 345–354. IEEE, 2013.

[112] R. Preissl, T. Kockerbauer, M. Schulz, D. Kranzlmuller, B. Supinski, and D.J. Quinlan. Detecting patterns in mpi communication traces. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pages 230–237, Sept 2008.

[113] ProbaYes.
http://www.probayes.com.

[114] Xuehai Qian, He Huang, Benjamin Sahelices, and Depei Qian. Rainbow: Efficient memory dependence recording with high replay parallelism for relaxed memory model. In *HPCA*, pages 554–565, 2013.

[115] NOKIA Ronan Mac Laverty. Robocop: Robust open component based software architecture for configurable devices project.
http://www.hitech-projects.com/euprojects/robocop/deliverables_public/robocop_wp1_deliverable15_18july2003.pdf.

[116] M. Ronsse and Willy Zwaenepoel. Execution Replay for TreadMarks. In *PDP*, pages 343–350. IEEE Computer Society, 1997.

[117] Michiel Ronsse, Koen De Bosschere, and Jacques Chassin De Kergommeaux. Execution replay and debugging. In *Proceedings of the Fourth International Workshop on Automated Debugging (AADEBUG2000*, 2000.

[118] Michiel Ronsse, Koen De Bosschere, Mark Christiaens, Jacques Chassin de Kergommeaux, and Dieter Kranzlmüller. Record/replay for nondeterministic program executions. *Commun. ACM*, 46(9):62–67, September 2003.

[119] Yasushi Saito. Jockey: A user-space library for record-replay debugging. In *In AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 69–76. ACM Press, 2005.

[120] Scalasca.
http://www.scalasca.org/.

[121] Lucas Schnorr. *Some Visualization Models applied to the Analysis of Parallel Applications*. PhD thesis, Institut National Polytechnique de Grenoble - INPG Universidade Federal do Rio Grande do Sul - UFRGS (2009-10-26), 2010.

[122] 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, 2012.

[123] H. Schwarz, D. Marpe, and T. Wiegand. Overview of the scalable video coding extension of the h.264/avc standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 17(9):1103–1120, Sept 2007.

[124] SCORE-P: Scalable Performance Measurement Infrastructure for Parallel Codes. http://www.vi-hps.org/projects/score-p/.

[125] Trace Compass. https://projects.eclipse.org/proposals/trace-compass.

[126] J. Seward, N. Nethercote, and J. Weidendorfer. *Valgrind 3.3 - Advanced Debugging and Profiling for GNU/Linux Applications*. Network Theory Ltd., 2008.

[127] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, VL '96, pages 336–, Washington, DC, USA, 1996. IEEE Computer Society.

[128] Luka Stanisic, Samuel Thibault, Arnaud Legrand, Brice Videau, and Jean-François Méhaut. Modeling and Simulation of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures. Research Report RR-8509, INRIA, March 2014.

[129] STMicroelectronics. KPTrace. http://www.stlinux.com/devel/traceprofile/kptrace.

[130] STMicroelectronics. Multi-Target Trace API. http://www.st.com/st-web-ui/static/active/jp/resource/technical/document/reference_manual/DM00053272.pdf.

[131] STMicroelectronics. Sti7105. http://www.st.com/web/en/catalog/mmc/FM131/SC999/SS1629/PF216978.

[132] STMicroelectronics. Sti7200. http://www.st.com/web/en/catalog/mmc/FM131/SC999/SS1629/PF160130.

[133] STMicroelectronics. STLinux Trace Viewer Trace Format. http://www.stlinux.com/stworkbench/interactive_analysis/stlinux.trace/kptrace_traceFormat.html.

[134] Apache Storm. https://storm.apache.org/.

[135] STWorkbench. http://stlinux.com/stworkbench/.

[136] Dinesh Subhraveti and Jason Nieh. Record and transplay: Partial checkpointing for replay debugging across heterogeneous systems. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '11, pages 109–120, New York, NY, USA, 2011. ACM.

[137] Sun Microsystems Inc. Enterprise JavaBeans Technology. http://java.sun.com/products/ejb/.

[138] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco: An aspect-oriented approach tailored for component based software development. In *Proceedings of the 2Nd International Conference on Aspect-oriented Software Development*, AOSD '03, pages 21–29, New York, NY, USA, 2003. ACM.

[139] Clemens Szyperski. Component technology: What, where, and how? In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 684–693, Washington, DC, USA, 2003. IEEE Computer Society.

[140] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Salsa: Analyzing logs as state machines. In *Proceedings of the First USENIX Conference on Analysis of System Logs*, WASL'08, pages 6–6, Berkeley, CA, USA, 2008. USENIX Association.

[141] TAU Website. http://www.cs.uoregon.edu/research/tau/home.php.

[142] H. Thane and H. Hansson. Using deterministic replay for debugging of distributed real-time systems. In *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*, pages 265–272, 2000.

[143] Vampir. http://www.vampir.eu/.

[144] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The koala component model for consumer electronics software. *Computer*, 33(3):78–85, March 2000.

[145] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. *SIGPLAN Not.*, 47(4):15–26, March 2011.

[146] VisualVM Website. http://visualvm.java.net/.

[147] Intel® VTune™ Amplifier 2015. http://software.intel.com/en-us/intel-vtune-amplifier-xe.

[148] Nan Wang, Jizhong Han, Haiping Fu, Xubin He, and Jinyun Fang. Reproducing non-deterministic bugs with lightweight recording in production environments. In *IPCCC'10*, pages 89–96, 2010.

[149] CTF : Common Trace Format.
http://www.efficios.com/ctf.

[150] HDF5.
http://www.hdfgroup.org/HDF5/.

[151] LTTng.
http://lttng.org/.

[152] Wiipedia. Component-based software engineering.
http://en.wikipedia.org/wiki/Component-based_software_engineering.

[153] Wayne Wolf. *High-Performance Embedded Computing: Architectures, Applications, and Methodologies*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[154] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 122–135, New York, NY, USA, 2003. ACM.

Publications

[155] Marcio Bastos Castro, Kiril Georgiev, Vania Marangozova-Martin, Jean-François Mehaut, L. G. FERNANDES, and Miguel Santana. Analyzing Software Transactional Memory Applications by Tracing Transactions. Technical Report 7334, INRIA, 2010.

[156] Marcio Bastos Castro, Kiril Georgiev, Vania Marangozova-Martin, Jean-François Mehaut, Luiz Gustavo Fernandes, and Miguel Santana. Analysis and Tracing of Applications Based on Software Transactional Memory on Multicore Architectures. In *Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP)*, pages 199–206, Ayia Napa, Cyprus, 2011. IEEE Computer Society.

[157] Thomas Ferrandiz and Vania Marangozova. Managing Scheduling and Replication in the LHC Grid. In *Proceedings of CoreGRID Workshop on Grid Middleware*, July 2007.

[158] Kiril Georgiev. *Débogage des systèmes embarqués multiprocesseur basé sur la ré-exécution déterministe et partielle*. PhD thesis, Université de Grenoble, France, 2012.

[159] Kiril Georgiev and Vania Marangozova-Martin. Deterministic Partial Replay for MPSoC Debugging. Technical Report 5815, INRIA, 2014.

[160] Kiril Georgiev and Vania Marangozova-Martin. MPSoC Zoom Debugging: A Deterministic Record-Partial Replay Approach. In *12th IEEE/IFIP International Conference on Embedded and UbiquitousComputing (EUC 2014)*, Milan, Italy, August 2014.

[161] Kiril Georgiev and Vania Marangozova-Martin. Facing the Challenge of Nondeterminism in MPSoC Debugging. Submitted to the JSA Elsevier Journal, January 2015.

[162] Vania Marangozova. Patrons de conception pour la duplication de composants. In *Proceedings de l'Association ACM-SIGOPS de France (ASF)*, April 2002.

[163] Vania Marangozova. *Duplication et cohérence configurables dans les applications réparties à base de composants*. These, Université Joseph-Fourier - Grenoble I, June 2003. theses/2003/Marangozova.Vania theses/2003/Marangozova.Vania.

[164] Vania Marangozova and Fabienne Boyer. Using reflective features to support mobile users. ECOOP'2000 Workshop on Reflection and Metalevel Architectures., May 2000.

[165] Vania Marangozova and Daniel Hagimont. Adaptation d'une application répartie pour la disponibilité. In *Deuxième Conférence Française sur les Systèmes d'Exploitation (CFSE 2)*, April 2001.

[166] Vania Marangozova and Daniel Hagimont. Availability through adaptation: a distributed application experiment and evaluation. European Research Seminar on Advances in Distributed Systems (ERSADS)., May 2001.

[167] Vania Marangozova and Daniel Hagimont. An Architectural Approach to Replication Configuration. In *Proceedings of 6th International Conference on Principles of Distributed Systems (OPODIS'2002)*, December 2002.

[168] Vania Marangozova and Daniel Hagimont. An Infrastructure for CORBA Component Replication. In *Proceedings of the First International IFIP/ACM Working Conference on Component Deployment*, June 2002.

[169] Vania Marangozova and Daniel Hagimont. Non-functional Replication Management in the Corba Component Model. In *Proceedings of 8th International Conference on Object-Oriented Information Systems (OOIS 2002)*, September 2002.

[170] Vania Marangozova-Martin and Generoso Pagano. SoC-TRACE: Handling the Challenge of Embedded Software Design and Optimization. In *Proceedings of the ACM/IFIP/Usenix International Middleware Conference*, Montreal, Canada, December 2012.

[171] Vania Marangozova-Martin and Generoso Pagano. Gestion de traces d'exécution pour le systèmes embarqués : contenu et stockage. Research Report RR-LIG-046, LIG, Grenoble, France, 2013.

[172] Alexis Martin and Vania Marangozova-Martin. Analyse de traces d'exécutions pour les systèmes embarqués : détection d'anomalies par corrélation temporelle. Technical Report RT-0450, Inria, October 2014.

[173] Alexis Martin, Generoso Pagano, Jérôme Correnoz, and Vania Marangozova-Martin. Analyse de systèmes embarqués par structuration de traces d'exécution. In Pascal Felber, Laurent Philippe, Etienne Riviere, and Arnaud Tisserand, editors, *ComPAS 2014 : conférence en parallélisme, architecture et systèmes*, Neuchâtel, Suisse, April 2014.

[174] http://soc-trace.minalogic.net.

[175] Projet SoC-TRACE. http://tinyurl.com/minalogic-soc-trace.

[176] Poliana Oliveira, Henrique Cota de Freitas, Christiane Pousa Ribeiro, Marcio Bastos Castro, Vania Marangozova-Martin, and Jean-François Mehaut. Performance Evaluation of WiNoCs for Parallel Workloads Based on Collective Communications. In *IADIS International Conference on Applied Computing (AC)*, Rio de Janeiro, Brazil, 2011. IADIS Press.

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

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

[179] Generoso Pagano and Vania Marangozova-Martin. SoC-Trace Infrastructure. Technical Report RT-0427, INRIA, July 2012.

[180] Generoso Pagano and Vania Marangozova-Martin. SoC-Trace Infrastructure Benchmark. Rapport Technique RT-0435, INRIA, June 2013.

[181] Generoso Pagano and Vania Marangozova-Martin. The framesoc software architecture for multiple-view trace data analysis. In *Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '14, pages 217–222, New York, NY, USA, 2014. ACM.

[182] Kevin Pouget, Miguel Santana, Vania Marangozova-Martin, and Jean-François Mehaut. Debugging Component-Based Embedded Applications. In *Joint Workshop Map2MPSoC (Mapping of Applications to MPSoCs) and SCOPES (Software and Compilers for Embedded Systems)*, St Goar, Germany, May 2012. published in the ACM library.

[183] Christiane Pousa Ribeiro, Marcio Bastos Castro, Jean-François Mehaut, Vania Marangozova-Martin, Henrique Cota de Freitas, and Carlos A. P. S. Martins. Investigating the Impact of CPU and Memory Affinity on Multi-core Platforms: A Case Study of Numerical Scientific Multithreaded Applications. In *IADIS International Conference on Applied Computing (AC)*, Rio de Janeiro, Brazil, 2011. IADIS Press.

[184] Christiane Pousa Ribeiro, Márcio Castro, Vania Marangozova-Martin, Jean-François Mehaut, Henrique Cota de Freitas, and Carlos Augusto Paiva da Silva Martins. Evaluating CPU and Memory Affinity for Numerical Scientific Multithreaded Benchmarks on Multicores. *IADIS International Journal on Computer Science and Information Systems (IJCSIS)*, 7(1):79–93, 2012.

[185] Christiane Pousa Ribeiro, Vania Marangozova, Jean-François Mehaut, Fabrice Dupros, and Alexandre Carissimi. Explorando Afinidade de Memória em Arquiteturas NUMA. In *Proceedings of IX Simpósio em Sistemas Computacionais (WSCAD-SSC 2008)*, Campo Grande, Brazil, October 2008.

[186] Carlos Prada, Vania Marangozova, Kiril Georgiev, Jean-François Mehaut, and Miguel Santana. Towards a Component-based Observation of MPSoC. In *Proceedings of 4th IEEE International Symposium on Embedded Multicore Systems-on-Chip*, September 2009.

[187] Carlos Prada, Vania Marangozova-Martin, Jean-François Mehaut, and Miguel Santana. A Generic Component-Based Approach to MPSoC Observation. In *9th IEEE/IFIP International Conference on Embedded and UbiquitousComputing (EUC 2011)*, Melbourne, Australia, October 2011.

[188] Carlos Prada-Rojas, Vania Marangozova, Kiril Georgiev, Jean-François Mehaut, and Miguel Santana. Observation de systèmes embarqués : une approche à base de composants. In *Conférence Française sur les Systèmes en Exploitation (CFSE)*, Toulouse, France, September 2009.

[189] Carlos Prada-Rojas, Vania Marangozova, Kiril Georgiev, Jean-François Mehaut, and Miguel Santana. Towards a Component-based Observation of MPSoC. Research Report 6905, INRIA, April 2009.

[190] Carlos Hernan Prada Rojas. *Une approche à base de composants logiciels pour l'observation de systèmes embarqués.* These, Université de Grenoble, June 2011.

[191] Luka Stanisic, Brice Videau, Johan Cronsioe, Augustin Degomme, Vania Marangozova-Martin, Arnaud Legrand, and Jean-François Mehaut. Performance Analysis of HPC Applications on Low-Power Embedded Platforms. In *Proceedings of the Conference on Design, Automation, Test in Europe (DATE'13)*, Grenoble, March 2013. Special Day on High-Performance Low-Power Computing.

[192] Brice Videau, Vania Marangozova-Martin, and Johan Cronsioe. BOAST: Bringing Optimization through Automatic Source-to-Source Tranformations. In *Proceedings of the 7th International Symposium on Embedded Multicore/Manycore System-on-Chip (MCSoC)*, Tokyo, Japan, September 2013. IEEE Computer Society.

[193] Brice Videau, Vania Marangozova-Martin, Luigi Genovese, and Thierry Deutsch. Optimizing 3D Convolutions for Wavelet Transforms on CPUs with SSE Units and GPUs. Research Report RR-LIG-032, LIG, Grenoble, France, 2012.

[194] Brice Videau, Vania Marangozova-Martin, Luigi Genovese, and Thierry Deutsch. Optimizing 3D Convolutions for Wavelet Transforms on CPUs with SSE Units and GPUs. In *Proceedings of the 19th Euro-Par International Conference*, Aachen, Germany, August 2013. Springer Berlin Heidelberg.