HAL
open science

Facing the Challenge of Nondeterminism in MPSoC Debugging

Kiril Georgiev, Vania Marangozova-Martin

To cite this version:

Kiril Georgiev, Vania Marangozova-Martin. Facing the Challenge of Nondeterminism in MPSoC Debugging. 2015. hal-01103620

HAL Id: hal-01103620
https://inria.hal.science/hal-01103620v1

Preprint submitted on 15 Jan 2015

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

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

Facing the Challenge of Nondeterminism in MPSoC Debugging

Kiril Georgiev[a,b], Vania Marangozova-Martin[a,b]

[a] *Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France*
[b] *CNRS, LIG, F-38000 Grenoble, France*

1. Introduction

Recent years have witnessed a tremendous development of embedded systems. They find their place in numerous domains in our everyday life like transports, domotics and telecommunications. This omnipresence calls for new design methods targeting more complex applications, more efficiency and yet a shorter time to market.

Multi-Processor Systems on Chip (MPSoC) architectures have been proposed to meet these new requirements. They follow the "multi-core trend" and propose an increasing number of components allowing for bigger computational power at a lower energetic cost. The hardware design includes general purpose processors, specialized accelerators, shared, as well as distributed memory, numerous peripherals and Network-on-Chip (NoC) interconnections.

The increasing hardware complexity of MPSoC brings new challenges to the process of software development. Indeed, parallel computations and concurrent data accesses makes software execution *nondeterministic*. As a consequence, software debugging faces the problem of detecting and rooting the causes of nondeterministic errors which are hard to observe and reproduce. The problem is even more emphasized by the increasing number of execution entities (hardware components, application processes, threads...) and their possible interactions.

Email addresses: `kiril.georgiev.sf@gmail.com` (Kiril Georgiev), `vania.marangozova@imag.fr` (Vania Marangozova-Martin)

One way to tackle the problem of debugging nondeterministic systems is to prevent nondeterminism via adapted hardware, runtime or programming mechanisms [1, 2, 3, 4]. Such a solution simplifies debugging as it guarantees error reproduction but is costly in terms of hardware or development efforts. The alternative approach is to use deterministic record-replay (DRR). 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 [5].

In this paper we present an overview of existing deterministic record replay solutions and investigate their application in the context of MPSoC systems. We analyze the specific needs in MPSoC debugging and report on our experience in implementing a DRR-based debugger. We show how we reduce the error search space and zoom on problematic zones by applying spatial and temporal selection criteria. We present our general debugging methodology and our ReDSoC prototype with its trace collection, trace visualization, deterministic replay and partial replay support. The implementation, as well as the validation in the case of two multimedia applications on two different platforms have taught us important lessons about the application of DRR in embedded systems. We put forward the importance of DRR for MPSoC debugging and discuss the cost of use and implementation.

The paper is organized as follows. Section 2 introduces nondeterministic systems. Section 3 proposes a classification of existing DRR solutions and presents the works targeting embedded systems. Section 4 details the design principles and the implementation of our DRR-based debugger ReDSoC. Section 5 illustrates the application of ReDSoC for debugging multimedia applications. Finally, section 6 presents the conclusions of this work.

2. Introducing Nondeterminism

A nondeterministic system is a system which may follow different execution paths when executed with the same data input. This may or may not lead to different results [6].

If we picture a system as a multi-layer stack (cf. Figure 1), nondeterminism may be found at all levels. It turns out, however, that the amount of nondeterminism increases with lower levels. Indeed, if layer i enforces a deterministic behavior and layer $i+1$ is based entirely on the interfaces provided by i, then $i+1$ will also be deterministic. There will be no guarantees about the layer $i-1$. For example, the *dOS* system [5] 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 ressources accesses or distributed communications, stay nondeterministic.

The main sources of nondeterminism are the following.

- *Data inputs.* If we consider mono-processor systems and the corresponding sequential executions, as the instruction order is defined, the execution variations may be caused uniquely by the system inputs. So, a first source of nondeterminism are *data inputs*.
- *Scheduling.* If we consider a single processor system with multiple execution flows and guarantee the same data inputs, the execution will depend on *scheduling*. In multiprocessor systems, this source of nondeterminism gains undoubtedly in importance.

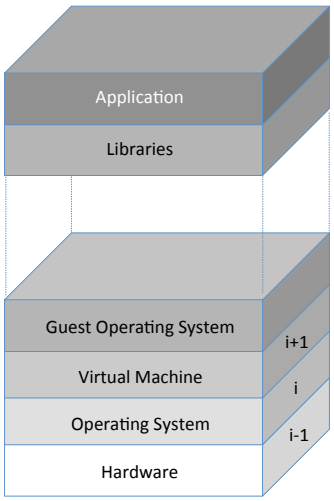

Figure 1: An Example of a Multi-Layered System: All layers are possibly nondeterministic

- *Data races.* If we consider parallel shared-memory systems, the system's behavior is defined by the interactions of multiple execution flows which access shared memory. If we guarantee the same data inputs and the same scheduling order, the execution will depend on the intermediary data manipulated by the execution flows. As this data may be shared, its value will depend on the execution flows' accesses and updates. This situation puts forward a third source of nondeterminism which is *data races*.
- *Interruptions.* Both inputs and scheduling are closely related to *interruptions*. Indeed, input data may either be ready and waiting in a given storage facility, or be made availably on-the-fly and be notified with an interruption. To reproduce the data input, one needs to reproduce the timing of the interruption as related to the number of already executed instructions. Interruptions also play a major role in scheduling when enforcing time sharing or real-time constraints.
- *Distributed Communications.* If we consider distributed systems, interactions are done via network communications. Execution flows thus manipulate

data received through the network. Nondeterminism arises when multiple senders address the same receiver or when the delivery of the message is asynchronous i.e may happen at different places of the execution path.

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.

3. Deterministic Record-Replay (DRR)

In this section we present the principle of deterministic record replay, present a classification of existing works and discuss solutions implemented in the domain of embedded systems.

3.1. The Idea

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. In Figure 2, the record phase identifies five nondeterministic situations and records the respective execution order or used data values.

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. When a nondeterministic execution point is reached, the replay process uses the recorded trace in order to enforce the recorded execution path. In Figure 2, the execution of operation 3 is delayed so as to happen after operation 2 and operation 4 uses the recorded data value.

3.2. The Design

Even if DRR has been investigated for more that 40 years now [6, 7, 8], recent technological evolutions have triggered a growing interest towards DDR techniques [9, 10, 11, 12, 13].

In the following, we identify the major design aspects of deterministic record replay solutions. Using these as classification criteria, we present a summary of the major works on the subject in Figure 3.

3.2.1. Application Domain

Our study distinguishes between works done in the domains of distributed systems, shared memory systems and embedded systems. DRR solutions for distributed systems consider nondeterminism due to network communications. DRR solutions for shared-memory systems either consider global centralized systems or put a major focus on data races. Finally, DRR solutions for embedded systems reflect the specific constraints of the target platforms.

3.2.2. Target System Architecture

DRR solutions evolve chronologically from simpler to more complex system architectures. Proposals start with mono-processor systems [14, 15], evolve to multi-processor architectures with increasingly relaxed consistency constraints [10, 16] and recently have considered modern multi-core platforms [17, 18, 19]. In addition to the hardware-level parallelism, many solutions also handle multithreading. Concerning distributed systems, there are dedicated DDR solutions considering C/C++ applications [20], the Java Virtual Machine [21] or Linux-like systems [22].

3.2.3. Target Sources of Nondeterminism

DRR solutions may be classified according to the sources of nondeterminism they consider. Data races, for example, are in the heart of numerous projects [23, 24, 25, 26, 9, 27, 12, 10]. The solutions reason either about individual memory accesses, or about periods (chunks, episodes) of

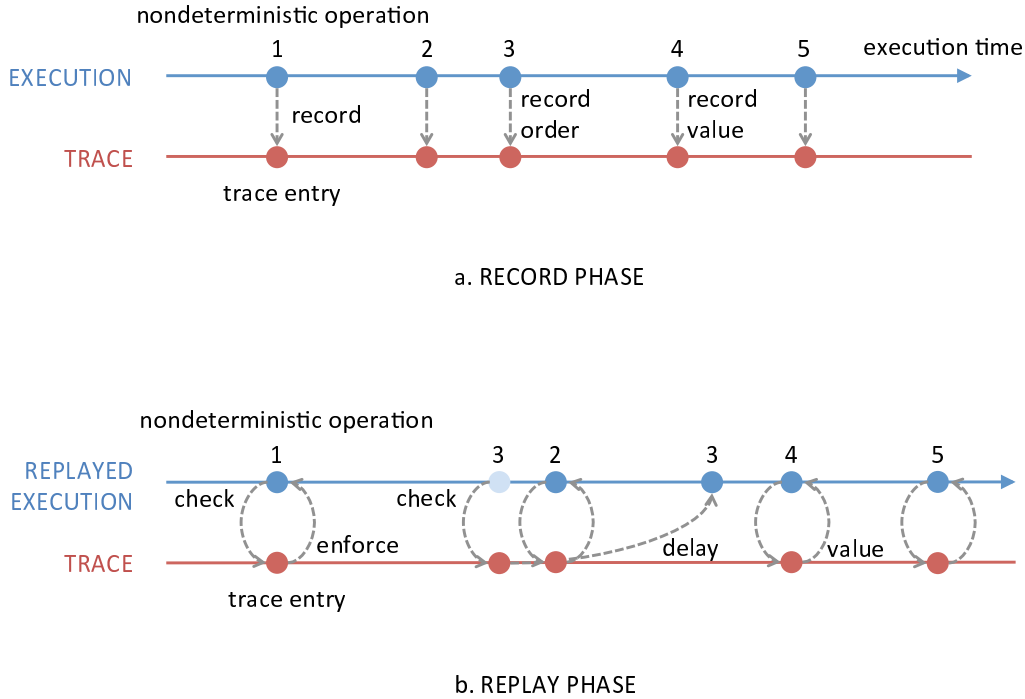

Figure 2: Deterministic Record-Replay

non conflicting accesses. They are all hardware-assisted in order to handle the tracing overhead due to the important number of operations.

Some works consider a larger set of sources of nondeterminism including not only data races but also interrupts and I/O [28, 19, 18, 16, 13, 17, 11]. BugNet [28], for example, allows the replay of a window of instructions before a system crash. Using system checkpoints created with a specific hardware, it reflects interrupts and data inputs and replays memory load and store operations. Crosscut [29] modifies the virtual machine layer to deterministically record-replay all instructions of the guest system execution. Scribe [17] modifies the operating system to guarantee deterministic record-replay of the user-level software.

Network communication nondeterminism is considered in works targeting distributed systems [20, 22].

As for embedded systems, the focus is on hardware interrupts because of their frequency and their impact on system performance [30, 31, 32].

3.2.4. Target Replay Layer

DRR solutions may target different system layers (cf. Figure 1). The decision has an impact on the implementation cost and on the replay precision. For example, data races may be considered at the operating system level or at the application level. At the operating system level, a DRR solution would need to manage an important number of fine-grained memory accesses. At the application level, it would need to tackle coarser-grained operations manipulating complex data such as objects or arrays. Replaying the system-level memory accesses would be more expensive but would guarantee the deterministic replay of the application-level operations. On the other hand, providing application-level only replay may, in many cases, be sufficient.

Existing DDR solutions target user-level software[14, 13], the virtual machine layer [15, 24] or the operating system layer [33, 18, 17, 11].

3.2.5. Implementation Level

Concerning the level of implementation of DRR solutions, a first classification is to distinguish between software- and hardware-based approaches [7, 8]. Hardware-based solutions have low tracing overhead but need expensive hardware extensions and cannot be applied to commodity systems. Software-based mechanisms, on the other hand, are more generic but usually come at a higher and even prohibitive execution overhead. The general approach is to decide the level of implementation according to the number of nondeterministic events to handle. So, typically, shared memory accesses are handled through hardware [12, 10, 25, 26, 28, 27], while other phenomena are handled with software [22, 15]. Some works benefit from both approaches and propose hybrid solutions [19].

At the software level, there are DDR solutions implemented at the operating system level, at the virtual machine layer and at the user level. The operating system level has the advantage of giving access to a full system state and allows the replay of different sources of nondeterminism [18, 17, 11]. Implementations at the virtual machine layer are platform-agnostic and manage time constraints easily. However, they usually come with voluminous logs and low record/replay speeds [15, 24, 29]. Implementations at the user level are lightweight in terms of development, usage and overhead [14, 13] but cannot help investigate problems at the lower levels.

3.2.6. Record Environment

Most DRR solutions targeting data races propose hardware-assisted solutions and therefore employ simulated environments [33, 28, 26, 25, 9]. Recent years have seen, however, the apparition of multiple proposals having reasonable overhead and operating on real platforms [11, 13, 18, 31].

3.2.7. Replay Environment

Most DDR solutions replay the system in the native environment i.e. in the environment in which took place the record phase. This is not a problem when the DRR solution targets a simulated environment [27, 16, 19]. However, this may be impractical when a real platform is used. Indeed, restoring a full system state, especially in a large scale environment, is a complex task. Additionally, if the replay concerns a client production environment, customers are usually not willing to provide replicas of their data. This is why, some replay solutions try to detach themselves from the native environment. The work presented in [31] records in a real platform but replays in a simulator. PinPlay [13] requires the same hardware platform but its replay facilities support multiple operating systems. Transplay [11] goes even further as it requires a compatible processor environment but does not need any libraries or application binaries.

3.2.8. Ease of Exploitation

What should be done to the target system to use a given DRR mechanism? Should the system be modified, how (automatically or by a developer) and at what cost? Hardware-based solutions demand specific hardware extensions which are usually costly to manufacture and difficult to apply in real production environments. Software-based solutions may require the installation of the specific software layer which implements the DRR. For DejaVu [15], for example, this is a modified Java Virtual Machine, while for DDOS [22], it is a modified operating system. In most cases, the target layers benefit from the DRR features in a transparent way.

3.2.9. DRR Optimization

One of the major questions for designing an effective DRR solution is to provide good performances for both the record and the replay phases.

Concerning the record phase, performance is directly related to the size of the record log. Indeed, the log size is

Project	Year	Application Domain	Target Architecture	Target Sources of Nondeterminism	Implementation Level	Replayed Layer
DejaVu [15]	1998	Shared-Memory Systems	Monoprocessor Multithreaded	Scheduling	Software (Virtual Machine)	User-Level Software
Flight Data Recorder [33]	2003	Shared-Memory Systems	Cache-coherent Multiprocessors Multithreaded	All	Hardware	Operating System
BugNet [28]	2005	Shared-Memory Systems	Sequential Consistency Multiprocessors Multithreaded	All	Hardware	User-Level Software
Jockey [14]	2005	Shared-Memory Systems	Monoprocessor	System calls Time-dependent CPU instructions	Software (User-Level)	User-Level Software
Strata [26]	2006	Shared-Memory Systems	Sequential Consistency Multiprocessors Multithreaded	Data races	Hardware	Operating System
SMPRevirt [24]	2008	Shared-Memory Systems	CREW Multiprocessors	All	Hardware + Software (Virtual machine)	Operating System
Rerun [25]	2008	Shared-Memory Systems	Sequential Consistency/ Total Store Ordering Multiprocessors Multithreaded	Data races	Hardware	Operating System
DeLorean [27]	2008	Shared-Memory Systems	Multiprocessor Multithreaded	Data races	Hardware	Operating System
Capo [19]	2009	Shared-Memory Systems	Multiprocessor Multithreaded	All	Hardware + Software (Operating System)	Operating System
Lreplay [16]	2010	Shared-Memory Systems	Multi-Core Processor	Data races, I/O	Hardware	Operating System
PinPlay [13]	2010	Shared-Memory Systems	Multi-core processor Multithreaded	All	Software (User-Level)	User-Level Software
Scribe [17]	2010	Shared-Memory Systems	Multiprocessor Multithreaded	All	Software (Operating System)	User-Level Software
Crosscut [29]	2010	Shared-Memory Systems	Multi-Core Processor	All	Software (Virtual machine)	Guest operating System
CoreRacer [9]	2011	Shared-Memory Systems	Total Store Ordering Multicore processors	Data races	Hardware	User-Level Software
Karma [12]	2011	Shared-Memory Systems	Directory cache coherence Multicore processors	Data races	Hardware	User-Level Software
DoublePlay [18]	2011	Shared-Memory Systems	Multicore processors	All	Software (Operating System)	Operating System (Linux process)
Transplay [11]	2011	Shared-Memory Systems	Multiprocessor Multithreaded	All	Software (Operating System)	User-Level Software
Rainbow [10]	2013	Shared-Memory Systems	Multiprocessor Multithreaded	Data races	Hardware	Operating System
RTReplayer [30]	2009	Embedded Systems	Monoprocessor Multithreaded	Interrupts	Software (Operating System)	Embedded Software
FlashBox [32]	2009	Embedded Systems	Monoprocessor	Interrupts	Hardware + Software (User-Level)	Embedded Software
Embedded Interrupts [31]	2012	Embedded Systems	Monoprocessor Monothreaded	Interrupts	Software (Operating System)	Embedded Software
Treadmarks [23]	1997	Distributed Systems	Cluster of Processors	Data races (synchronization)	Software (User-Level)	User-Level Software
Distributed DejaVu [21]	2000	Distributed Systems	Cluster of Processors Multithreaded	Distributed communications	Software (Virtual Machine)	User-Level Software
Liblog [20]	2006	Distributed Systems	Cluster of Processors Multithreaded	All	Software (User-Level)	User-Level Software
DDOS [22]	2013	Distributed Systems	Cluster of Processors Multithreaded	Distributed communications	Software (Operating System)	Operating System

Figure 3: DRR Projects

Record Environment	Replay Environment	Optimization	Ease of Exploitation
Real platform	Native	-	Transparent
Simulated Hardware (Simics)	Simulated	Trace size	Needs specific Hardware
Simulated Hardware (Simics)	Simulated	Time and architecture slicing	Needs specific Hardware
Real platform	Native	-	Transparent
Simulated Hardware (Simics)	Simulated	Trace size	Needs specific Hardware
Real platform	Native	-	Transparent
Simulated Hardware (GEMS)	Simulated	Trace size	Needs specific Hardware
Simulated Hardware (Simics + SESC)	Simulated	Configurable trade-off between trace size and replay speed	Needs specific Hardware
Simulated Hardware + Lunix	Native	Vertical and architecture slicing	Needs specific Hardware
Simulated Hardware (Xtreme-III)	Simulated	Trace size	Needs specific Hardware
Real platform	Native Hardware, OS independant	-	Transparent
Real platform	Same Hardware features	Record and Replay Speed	Transparent
Virtual machine	Native	Time, vertical and architecture slicing	Transparent
Simulated Hardware (x86 cycle accurate)	Simulated	-	Needs specific Hardware
Simulated Hardware (Simics)	Simulated	Trace size Record speed Parallel replay	Needs specific Hardware
Real platform	Native	Multiple Record Parallel Replay	Transparent
Real platform	Similar CPU No initial environment or code	Time and architecture slicing	Transparent
Simulated Hardware (Simics)	Simulated	Trace size Parallel replay	Needs specific Hardware
Real platform	Native	-	Transparent
Real platform	Native	-	Needs specific Hardware + Recompilation
Real platform	Simulated	Trace size Record speed	Transparent
Real platform	Native	Trace size	Transparent
Real platform	Native	-	API changes
Real platform	Native	-	Transparent
Real platform	Native	Node determinism	Transparent

proportional to the system slow down due to additional tracing instructions and to log storage. Reducing the log may go through efficient data encoding, memory buffering or minimal tracing. This puts, however, the effort on the replay phase which will need to do more complex computations to reproduce nondeterministic situations.

To optimize the record phase, most existing solutions do not consider the entire nondeterministic system but limit their scope (cf. Figure4). They may do so by choosing the target nondeterministic phenomena to consider and the target layer of interest (*vertical slicing*). These have been discussed in the paragraphs about the *Target Sources of Nondeterminism* and *Target Replay Layer*.

DRR solutions may further focus on different execution periods (*time slicing*) and consider some predefined entities (*architecture slicing*). *Time slicing* is used to replay only useful execution periods. At a fine grain, it is applied in period-based data race replay solutions [9, 18]. At a coarse grain, it may be based on checkpoints which define the boundaries of selectable time periods [13, 29, 11]

Architectural slicing requires a prior knowledge of the architecture of the target system. For example, in component-based applications, DRR solutions may focus only on a subset of application's components. When the target layer is the operating system, some existing DDR solutions allow for replaying only a subset of processes [19, 29]. In a distributed setting, it may be possible to replay a subgroup of nodes [20].

Concerning the replay phase, its optimization has been put into focus only in recent works. Approaches include the parallelization of the replay [18, 12], skipping operations by directly providing the result [19] and replaying only a part of the system [29].

3.3. DRR in Embedded Systems

There are few works on deterministic record replay for embedded systems [30, 31, 32] and they all focus on hard-

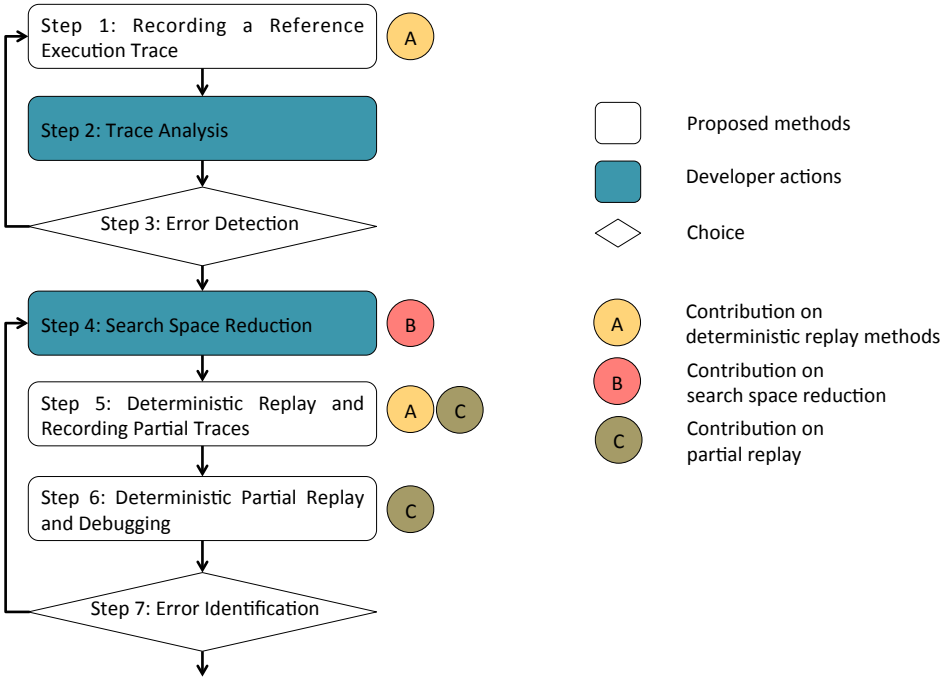

Figure 5: Debugging Cycle

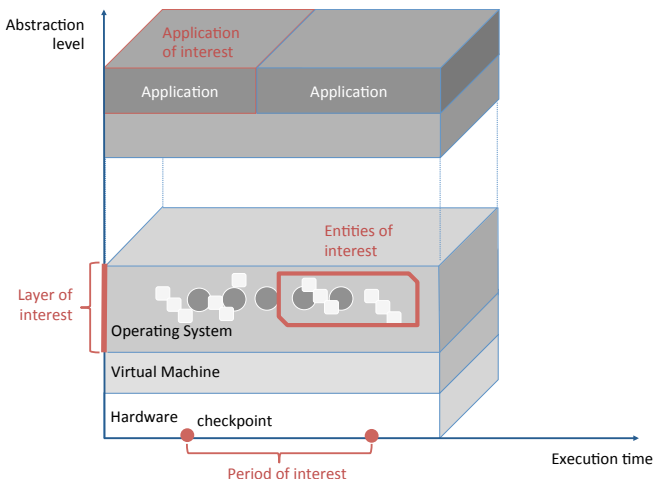

Figure 4: Taming Overhead in Deterministic Record-Replay

ware interrupts. Their motivation is that, on one hand, deterministically replaying the hardware allows for a deterministic replay of the software. On the other hand, the approach allows DRR to reflect hardware exploitation needed for performance analysis.

RT-Replayer [30] proposes a software-based DDR solution. Implemented at the operating system level, it logs hardware interrupts and the accompanying data in order to provide precise timing replay. To do so, RT-Replayer introduces the notion of virtual timestamps which is a combination of the program counter and the time difference between two consecutive events. During replay, it uses instruction hooking in order to intercept instructions and emulate interruptions. The implementation has been done in a research kernel running on an ARM920T processor. The implementation has considered a single processor and no I/O events, nor storage facilities.

FlashBox [32] proposes a hybrid hardware-software approach for capturing interrupts. The embedded system is instrumented with the aid of a specific compiler while the logging of interrupts is done using a flash memory and a microcontroller. The implementation has been done on an AVR butterfly board running a single Atmel ATmega169V 8-bit processor.

The work presented in [31] differs from previous approaches by proposing a quantative-based (speculative) approach for handling interrupts. The idea is not to record punctual interrupt events but record, at certain points, the system state produced by a selector function. The authors explore the questions of constructing a good selector

function and analyse the related timing overhead. The implementation is done in the EPOS operating system on the ATMEL AVR platform including a single ATmega16 processor.

Not only the three proposals consider exclusively interrupts, but they also address single monocore processor platforms. Our proposal with the RedSoC system, described in the next sections, pushes the effort of DRR for embedded systems further, by considering MPSoC architectures and working on a larger set of sources of nondeterminism.

4. ReDSoC: A DRR-Debugger for MPSoC

The goal of the RedSoC system is to propose a debugging solution for embedded systems respecting the following requirements:

- *Applicability to MPSoC Architectures*
  ReDSoC is to propose a DRR solution which adresses MPSoC architectures, namely the growing number of processors and the inherently distributed architecture of embedded boards.
- *Multiple Sources of nondeterminism*
  MPSoC architectures feature all sources of nondeterminism including data inputs, data races, scheduling, network communications and interrupts. As opposed to related work which mainly targets interrupts, ReDSoC is to consider multiple sources of nondeterminism.
- *Genericity*
  ReDSoC should not be specific to a specific embedded product or architecture. It should define the major steps of a debugging methodology and propose mechanisms that do not depend on particular hardware characteristics.
- *Scalability*
  RedSoC should be able to operate on platforms with an important number of hardware and software components.

In the following sections we detail our choices to meet these requirements.

4.1. Debugging Methodology

We use the DRR principles to propose a general debugging methodology, illustrated in Figure 5. The debugging cycle is composed of repeatable steps built around deterministic-record and partial-deterministic-replay actions. The steps are the following.

Step 1: Recording a Reference Execution Trace During this step, the execution of the whole MPSoC software is recorded to produce reference execution traces. These reference traces target the nondeterministic behavior to debug and are exploited in the next debugging steps. The data captured in these traces has been defined in close relation with the nondeterministic phenomena we have decided to target, as well as with the replay techniques we have chosen. Their volume is limited to minimize the tracing overhead during execution. The choice of target nondeterministic phenomena to debug and the identification of adapted replay algorithms represents our first contribution (A).

Step 2: Trace Analysis This step is performed by the developer who debugs the MPSoC software. Using available tools and his/her experience, the developer analyzes the reference traces in search of abnormal behavior.

Step 3: Error Detection At this step, 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.

Step 4: Spatial and Temporal Reduction of the Search Space During this step, the developer decides to focus on a particular part of the software execution thus reducing the search space. To do so, the developer may apply a

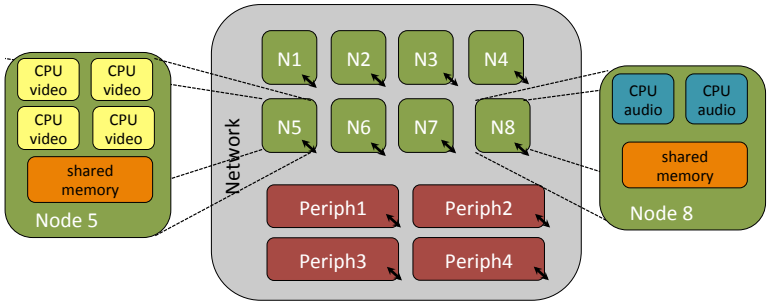

Figure 6: MPSoC Hardware Architecture

spatial and/or a temporal selection criteria. He/she selects a suspected part of the application to debug during a specific time interval. The definition of these criteria represents our second contribution (B).

Step 5: Deterministic Replay and Recording Partial Traces During this step, the reference trace is deterministically replayed to capture additional data reflecting the execution of the software part, selected in Step 4.

Step 6: Deterministic Partial Replay and Debugging During this step, only the selected software part is considered and the corresponding trace deterministically replayed (C). The replay mechanism is connected to a debugging tool, so the developer may debug the execution of the selected part and during the selected time interval in a standard way.

Step 7: Error Identification If the error source is not identified after Step 6, the developer goes back to Step 4. If the developer wants to focus on a different software part, the cycle goes through Step 5. If the developer considers the same software part but during a different time interval, there is no need for additional trace collection and the cycle continues directly with Step 6.

4.2. Design

If we consider the classification criteria presented in Section 3.2, RedSoC has operated the following choices.

4.2.1. Target System Architecture

ReDSoC considers an architecture featuring both shared-memory and distributed-memory. It is based on the generic hardware model showed in Figure 6.

MPSoC components include processors, memory blocs, peripherals and a communication network. Processors are computational units including general purpose processors, cores or accelerators. They are organized in a two-level hierarchy. Homogeneous processors form groups we call *nodes*. Thus there may be a node with audio processing units and another specialized in video decoding.

In a node, processors have access to and communicate through a shared memory bloc. Among nodes, memory is distributed and a processor from one node cannot access the memory of another node without passing through inter-node network connections.

Peripherals are the devices ensuring data exchange between the MPSoC and the external environment. Peripherals may include sensors, keyboards, screens, microphones, etc. The data they capture is communicated to the processors via the memory or the network.

As for MPSoC software, our assumptions are the following. The software execution is composed of a set of execution flows which is statically partitioned and scheduled on the MPSoC nodes. The execution flows scheduled on the same node communicate using the shared memory bloc and via synchronization. The execution flows scheduled on different nodes communicate using message-passing through the network. Data from peripherals is acquired either by polling, or through interrupts.

4.2.2. Target Sources of Nondeterminism

RedSoC focuses on shared data accesses, network communications and I/O operations. Our approach is thus complementary to related works focusing on interruption replay.

Given that recording all accesses to shared data implies a prohibitive execution overhead [34], our record-replay mechanism focuses on accesses to synchronization structures. Non-synchronized shared data accesses are considered to be errors, to be detected and corrected. We have chosen the algorithm proposed by Levrouw et al. in [35]. The algorithm uses Lamport clocks to identify accesses to different synchronization structures by different execution flows.

To trace and deterministically replay network communications, we have used the solution proposed in [36, 37]. For blocking network communications, the detection of race reception primitives is based on vector clocks. For non blocking reception operations, there is a need to record the number of executed *probes*, as well as their outcome (message available or not). This solution has minimal intrusion as it traces only race reception operations.

To deterministically replay input operations, we have decided to limit the intrusion of our mechanism by not recording interrupts and only consider polling requests. We suppose that the content of the input data is recorded by specific devices. We only record the input size in the reference execution trace (Step 1). During replay, the trace is read to decide that there is an input operation which is in turn acquired by executing a polling request to the specific recording device.

4.2.3. Target Replay Layer

ReDSoC replays the embedded application. We consider that a top-down approach is needed in order to locate the region where the application behaves incorrectly. Indeed, considering low-level hardware or operating system events from the beginning has two negative consequences. On one hand, the resulting execution log is voluminous and difficult to store. On the other hand, numerous low-level events are difficult to analyze as they do not provide a macroscopic view of the execution to the developper.

To partially replay MPSoC software execution, we apply two selection criteria concerning the software architecture (architecture slicing) and the execution duration (time slicing).

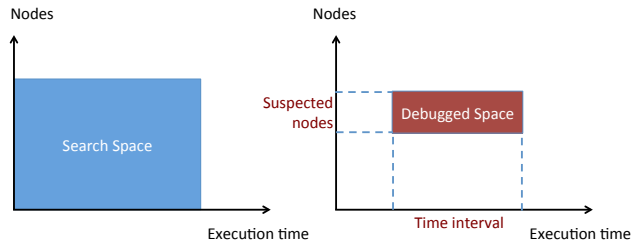

Figure 7: Search Space Reduction

The architecture slicing isolates a set of nodes on which the debugging can focus. The replay phase thus concerns only the execution flows running on the identified set of nodes. We call the set of nodes to be debugged, the *suspected* nodes. The unsuspected nodes are called the *correct* nodes.

To isolate suspected nodes from the correct ones, the tracing phase needs to differentiate the nodes and consider their message exchanges. Indeed, messages exchanged between correct nodes are not to be recorded as these would not participate in the replay. Messages exchanged between suspected nodes do not need to be recorded either, as they will be executed during replay. In the case of a message sent from a suspected node to a correct one, as the receive operation has no relevance to the replay, the replay may skip the send operation. In the case of a message sent by a correct node to a suspected one, the order and the content of the message need to be traced. During replay, the trace is used to decide whether to execute a message exchange operation and also to provide message values coming from the external/correct nodes.

Time slicing is based on the time sequence of events

recorded in the trace. The developer needs to delimit the interval to consider during debugging. This is done by choosing the interval limits which are two traced events. The choice is typically facilitated by a visualization tool which represents the trace. During replay, re-executed events are compared to the chosen interval beginning. When this event is reached, a debugger is launched and a standard debugging process may start. When the interval end is reached, the debugging phase terminates.

4.2.4. Implementation Level

ReDSoC is a purely software solution. It intercepts the API provided by the lower software layers to embedded applications. This approach does not need specific hardware, nor it needs to modify the application source code.

4.2.5. Record and Replay Environments

ReDSoC considers real execution platforms for the record phase and replays embedded applications in their native environment. The approach benefits from a good replay precision as it uses the same execution context and does not need to deal with inaccuracy due to a simulated environment. Moreover, it shows the applicability of ReDSoC design choices.

4.2.6. Ease of Exploitation

ReDSoC is easy to apply in an embedded platform as it does not require specific hardware, nor application source code modifications. ReDSoC also comes with a trace visualization tool which greatly helps the debugging work of the developer.

4.2.7. DRR Optimization

ReDSoC has been designed with the goal of minimizing intrusion during the record phase. The approach is to minimize the quantity of data when tracing nondeterministic events. As ReDSoC has been implemented as a proof of concept of a generic debugging methodology, many optimization possibilities remain. The experiments with RedSoC with real-world applications and platforms have shown that the speed of the replay phase should be improved, while the record speed and log volumes show good performances .

4.3. Implementation

The architecture of our prototype is given on Figure 8.

We consider a standard debugging configuration including a host platform connected to the target MPSoC platform. This is necessary as in many cases MPSoCs have limited resources and do not provide keyboard and screen peripherals.

ReDSoC is deployed both on the host machine and the target MPSoC machine. It is composed of four tools, namely a trace visualization tool, a partial replay tool, a trace collection tool and a deterministic replay tool. The trace collection tool, as well as the temporal selection management of the partial replay tool are deployed on the host machine. The other tools are deployed on the MPSoC, each MPSoC node having its own ReDSoC instance. The deployment on a MPSoC node is guided using a configuration file, provided by the developer. The file indicates the node number, the debugging phase to consider (Steps 1, 5 or 6 on Figure 5), as well as the identifiers of the suspected nodes.

The host machine is supposed to run a Linux-based system and have GDB for debugging. The MPSoC runs a MPSoC kernel characterized by a MPSoC API. The MPSoC API is inspired by the POSIX standard and includes basic functions for execution flow management, synchronization, network communications and I/O [38].

Our trace collection tool is deployed on each node of the MPSoC platform. As its purpose is to intercept the calls to the defined MPSoC API, it provides a simple interface including a `trace` function used for generating trace entries.

The tool for deterministic replay heavily relies on the MPSoC API. Shared data accesses are managed through

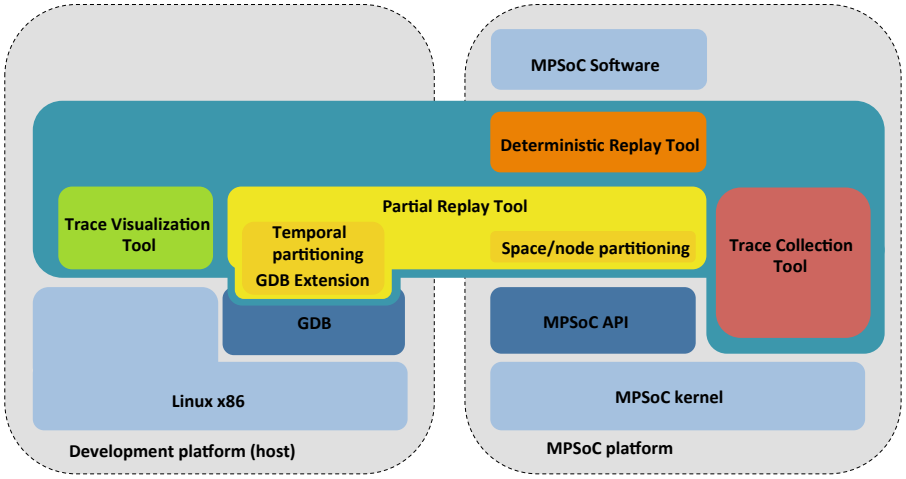

Figure 8: ReDSoC Architecture

tracking the synchronization operations of the API. Network communications are targeted using our message-based communication. Finally, I/O are addressed by the MPSoC file-oriented I/O operations.

To apply the space reduction criterion based on isolating suspected nodes, our partial replay tool needs to monitor and record all communications between normal and suspected nodes. During replay, each communication operation is intercepted to decide whether a normal node takes part in it or not. If yes, the operation is replayed by directly reading the needed values from the recorded trace. If the communication is between suspected nodes, the operation is normally executed.

To apply the time reduction criterion, we have implemented an extension for GDB and introduced a new type of breakpoint. We use *replay breakpoints* corresponding to the limits of the time interval that has been selected for debugging. Each replay breakpoint corresponds to an event recorded in the trace and is identified by a triple containing a node identifier, a task identifier and a timestamp.

During replay, each call to the MPSoC API is intercepted and compared to the limits of the selected time interval. If it does not correspond to any of them, the execution is pursued. If the call corresponds to the start of the time interval, the execution is suspended and the debugging starts. When the end of the time interval is reached, the debugging stops and the developer may choose a new time interval. If it is after the previous time interval, the execution continues. If not, it is launched from the beginning.

We have adapted the KPTrace Viewer of STMicroelectronics [39] to visualize our recorded traces. The viewer allows for representation of an event, characterized by a time, a timestamp, a process identifier and a number of arguments. We have provided for a tool formatting our traces according to the Pajé [40] format and adapted the KPTrace viewer to take into account its visualization.

An example of visualization is shown on Figure 9. The x dimension gives the time progression. The y dimension represents tasks. The arrows show three successive accesses of the tasks $T0$, $T2$ and $T1$ to a shared synchronization structure. The flags show peripheral operations, their color being specific to each peripheral device.

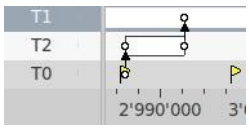

Figure 9: A fragment of trace visualization

5. Debugging Nondeterministic Multimedia Applications

We have validated our approach with a real-time game application on an MPSoC platform (Section 5.1) and a video-decoding application on a NUMA platform (Section 5.2). The performances of our framework are discussed in Section 5.3.

5.1. Debugging a Tetris Application on an MPSoC Platform

For this use case, we have used a Stagecoach expansion board having two OveroFE COM nodes (computer-on-module)[1]. Each node has an ARM Cortex-A8 600MHz processor with 256MB of DDR RAM, 256MB of NAND flash memory and a microSD port. The two nodes occupy the first and the third slot of the board. They are connected through a 100Mb/s Ethernet link and have distinct IP addresses. The RJ45 slot of the board is used to connect to an external network card which gives IP access to both nodes.

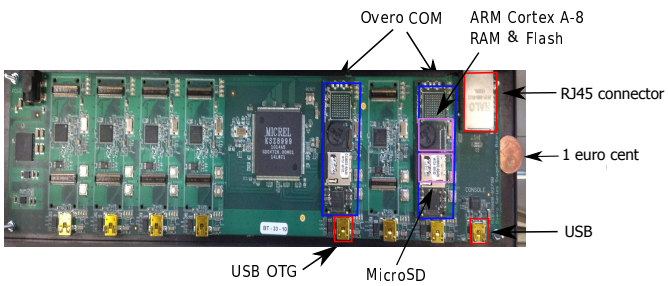

Figure 10: Stagecoach board with two Overo FE COM nodes

We have implemented our MPSoC API using the POSIX and the *libc* interfaces. We have installed the platform from scratch by creating a bootable microSD with the needed Linux distribution. The system image includes the 2.6 Linux kernel, *libc6*, a file system and the `ssh` service. To deploy the platform, we have used the cross-compiler provided in the Sourcery Codebench [2] to create a x86 executable. The executable contains the MPSoC application, the ReDSoC tools, as well as a GDB server.

The debugged MPSoC application is the Tetris game for two players (cf. Figure 11). The application's size is about 0,7MB and contains about 15000 lines of code. It is executed by two tasks that run respectively on the two MPSoC nodes.

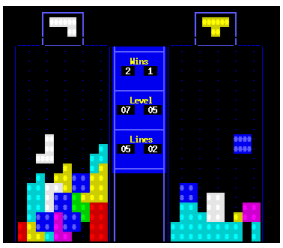

Figure 11: Two Player Tetris.

Both players see both Tetris boards. When a player succeeds in making disappear multiple lines, the other player's game becomes harder. The player whose board fills first, loses the game.

The Tetris pieces movements are controlled through the keyboard and also using the clock frequency. The keyboard is scanned for player commands, while the clock frequency is used to advance the pieces downwards.

In our use case, we needed to debug the application as, from time to time, one of the Tetris instances crashed and as a consequence the other player won. Following our debug cycle, we re-executed several times the Tetris application to obtain a reference trace containing the error (cf. Figure 12).

As the node to fail is node 1, this node is suspected and chosen as a target for the partial replay. To select the time interval for debugging, we focus and zoom the end of its trace (cf. Figure 13). We select the small interval containing three operations reading the system clock,

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

[2] http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/

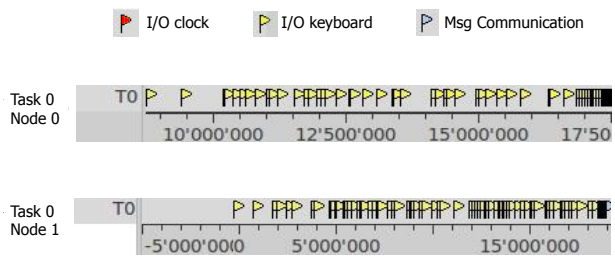

Figure 12: Visualization of the Tetris Reference Trace

four keyboard inputs and one message reception. As each event can be examined, we can see that the first event is a `GetTimerOp` operation, executed by task $T0$ at time $19'244'641\mu s$. The last event is a `NetRecvOp` executed by $T0$ at time $19'244'728\mu s$. These two events are defined as the two replay breakpoints for the debugging session.

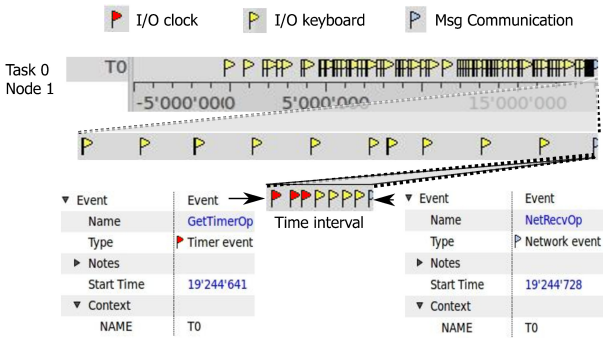

Figure 13: Time Interval Selection

ReDSoC needs to first deterministically replay the whole application to gather additional traces about the communications of node 1 with node 2. Once these traces are generated, ReDSoC may start the deterministic replay of node 1 and debug it during the selected time interval. Indeed, when the replay reaches the first replay breakpoint, ReDSoC starts a standard debugging session (cf. Figure 14).

The figure contains a screen capture of the debugging session when the first replay breakpoint is reached. The first line's information states clearly the number of the entry in the trace (202459), the type of the entry (IO), the node identifier (Node1) and the task identifier (Task0).

The `bt` GDB command given on the fourth line gives the function call stack. We observe the interaction between the GDB server and our GDB extension implemented in the `rdb_notify_event` function. The additional parameter information for `rdb_notify_syscall` confirms that the replay considers an IO operation of the task with `tid=0` on node `node=1`. Up the call stack, we see the replay function for IO operations (`replayIOsize`) and the MPSoC function calls.

When the debugging session reaches the last message reception operation, it is possible to investigate the received value. It appears that it is not correct and contains zero. This value is used in a division operation and the division by zero makes the node 1 to crash. To understand why the value is incorrect, we choose to suspect the other node, node 0. When we focus on the end of its trace, we observe a non regular behavior. Partially replaying node 0 and debugging it during a time interval at the end of its execution, makes us discover that there are many keyboard input operations resulting from a continuous pressing of a keyboard key. The input data being saved in a memory buffer, an error in the buffer management makes it overflow and results in sending an incorrect value.

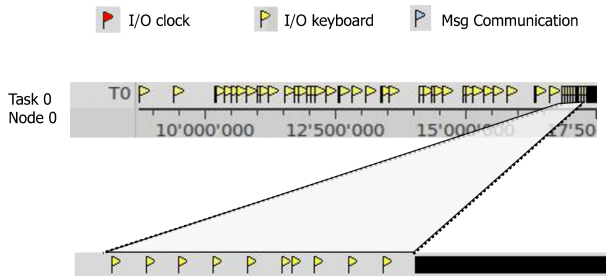

Figure 15: Considering a Different Node and a Different Time Interval

5.2. *Debugging a Video Decoding Application on a NUMA Platform*

To validate the scalability of our approach and given the unavailability of a large scale MPSoC platform at the time of the experience, we have developed the use case on a NUMA platform. The considered MPSoC software is the FFMPEG video decoder [41, 42].

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 14: Partial Debugging of the MPSoC Tetris Application

The NUMA architecture used in our experiments has four nodes, each having eight dual core 2.2 GHz AMD Opteron processors and 32GB of main memory.

In the final experimental setup, one node is considered to be the master one, and as such can access the file system, as well as the peripherals. The master node is also responsible for communicating input peripheral data to the other nodes. It occupies four of the NUMA processors, the other four being reserved for GDB. The other three nodes are MPSoC slave nodes.

The implementation of our MPSoC API uses the Linux2.6 interface, as well as the *libSDL*[3] and *libc* libraries. The task management and synchronization functions are based on the POSIX interface and use the system call `sched_setaffinity`. The I/O functions encapsulate the accesses to the file system, the screen, the keyboard, the audio card and the system clock. The file system is accessed using the *libc* functions. The audio and video peripherals are accessed through *libSDL* calls. Finally, the system clock is accessed using a dedicated Linux register. The network communication primitives are based on the inter-process socket-based communication of Linux.

From the FFMPEG suite, we have used the FFPLAY [43] and FFSERVER [44] components. FFSERVER is a video server, receiving video flows through different protocols (e.g., RTP or RTSP) and creating multiple output flows having different formats (H.264, DIVX, MPEG-4, etc). FFPLAY is a video decoder, receiving and synchronizing audio and video frames. Using these components, we have created a video mosaic application (cf. Figure 16). We have re-engineered the code to redirect all Linux function calls to our MPSoC API.

Figure 16: Video Mosaic Application

The video mosaic application exhibited a nondetermin-

[3] http://www.libsdl.org/

istic bug. During some executions, one or more videos were not visible. By tracing one of these executions, we captured the situation shown on Figure 17. The trace of Node0 (FFSERVER) shows the non blocking receptions of messages coming from FFPLAY components. The other three traces (FFPLAY components) show, in the beginning of their execution, receptions of messages from FFSERVER, followed by synchronization operations related to the work with memory buffers containing the audio/video data. We can clearly see that at one point, Task2 on Node2 blocks and causes the blocking of Task0 and Task1.

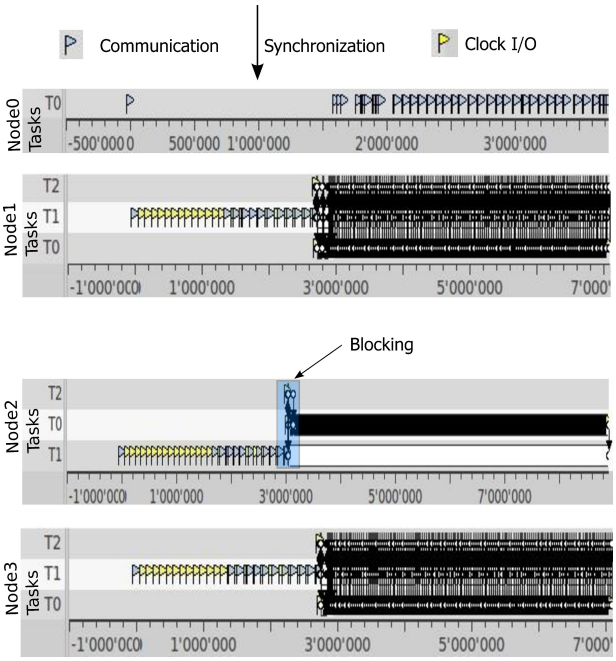

Figure 17: Visualization of Captured Traces.

Having selected this node as the suspected one, as well as the short time interval directly preceding the blocking, the debugging session proved rather straightforward. By tracking the accesses to synchronization structures, we observed that a condition variable is never signaled. During a second replay, we established the connection between this variable and the memory allocation for video frames. During a third replay, we discovered that the developer has forgotten to notify the frame memory allocation.

5.3. Performances

To evaluate the performances of our implementation, we have considered three criteria, namely the intrusion during normal execution, the trace volume and the execution speed during debugging. To evaluate the intrusion of ReDSoC during the recording phase, we have considered both the embedded and the NUMA platforms and have used the native execution time and the reference execution time. The native execution time reflects the execution duration of the software without ReDSoC. The measure is obtained as a mean value of thirty executions.The reference execution time is the mean execution time of the same software with the same inputs but running under the control of ReDSoC. This execution is logically slower due to the interception of function calls and the tracing mechanism. Using the two previous measures, the overhead gives the execution slowdown as a percentage.

The considered applications include a simple MJPEG decoder, the Tetris application and the video mosaic application. The results are given in Table 1.

In all use cases, the intrusion is very low (Overhead column, 4% for MJPEG and less than 1% in the other cases) and does not cause video glitch visible to the eye. In the case of the Tetris application, for example, this is explained by the fact that the time spent for moving the pieces is much smaller that the time between moves. As a consequence, tracing happens during this inactivity time and does not perturb the application. In the case of the video mosaic application, the tracing situation is similar: the application behavior is very regular and the tracing operations happen in between image decoding operations.

Obviously, this low intrusion cannot be generalized for all cases. However, this experiment confirms the utility to have a resource provisioning (here the management of time constraints) for the tracing operations. Indeed, in most MPSoC platforms, the architecture includes hardware tracing ports which do not perturb normal execution. It is interesting to apply this approach to tracing of the

Software	Native Time(s)	Reference Time(s)	Overhead (%)	Trace Data(KB)	Trace Entries
MJPEG					
Node0	139	144	3,59	2298	45471
Tetris					
Node0	62	62	< 1	333	887
Node1	60	60	< 1	201	530
Video Mosaic					
FFSERVER node	31	31	< 1	500	1345

Table 1: Intrusion Measures

upper software layers.

As nondeterministic behavior cannot be easily reproduced and captured, we also note that there is no general prediction about the number of executions a developer needs to run to obtain the reference trace.

Considering the trace volumes (Trace Data column), as we focus on a restrained type of events to record, in all cases the number of entries is rather small (Trace Entries column). In the MJPEG case, for example, due to the more intensive use of synchronization, the number of entries (45471) is more important, which explains the perceivable execution time overhead. The trace data volume is minimal, as we do not record the full data characterizing an event but only the information needed for deterministic replay.

To start the debugging session itself, the actual ReDSoC solution forces the developer to wait for the deterministic replay to happen and reach the selected time interval. In the worst cases, if the selected debugging region is at the end of the execution, the developer needs to wait for two replays, corresponding to the deterministic and partial trace recordings respectively. In the case of the Tetris application, for exemple, if the execution time of $Node0$ is $61s$, the waiting time for the developer to be able to debug $Node0$ is about $161s$. An interesting approach to accelerate the process would be to manage application snapshots allowing the deterministic replay to start in the middle of an application execution.

6. Conclusion

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.

This work presents ReDSoC, a software-level DRR solution targeting MPSoC and multiple sources of nondeterminism. Considering a generic hardware model of MPSoC systems and presupposing the existence of a 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 inter-

val. 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 de-zooms 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 for guaranteeing the reproduction of the execution or for re-executing only the selected 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, 46].

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 [47] but the approach is to be investigated for post-mortem analysis.

The ReDSoC project helped us identify the major difficulties in applying DRR techniques in embedded systems.

Embedded systems constraints versus DRR intrusion. First of all, DRR demands additional computational and storage ressources and directly affects the performances of a constrained embedded system. The various DRR solutions minimizing the execution overhead in an ad-hoc manner are difficult to evaluate and reuse. Modeling and formally estimating the cost of a given DRR technique will allow for cost predictions and will greatly facilitate the choice between different solutions. However, experience shows that such models are challenging to build because of the complexity and the dynamicity of systems.

Embedded systems hardware diversity. A more promising approach, already applied in many cases of embedded system design [48] and promoted in embedded design books [49] is to dimension embedded systems with additional hardware resources for record/replay. However, embedded systems are known for their architectural and hardware diversity. Defining a hardware support in one architecture case will be a specific solution difficult to apply in a different embedded system [30, 32, 31]. The definition of a standard hardware targeting record-replay is a perspective to be considered a logic continuation of the effort of defining standard tracing architectures such as JTAG

[50].

Lack of standards for embedded software. The difficulty to implement hardware-assisted DRR leads to the alternative of implementing the needed mechanisms at the software level. However, most embedded systems come with a proprietary software including diverse operating systems, application programming models and interfaces. With the lack of standard software API, software-level solutions are platform-specific and moreover cannot efficiently address hardware events which are a key element to performance. It is our belief that the domain will evolve and crystallize different types of embedded systems with clearer software layered architectures and standardized interfaces.

Acknowledgments

The ReDSoC development has been done in a collaboration with the IDTEC department of STMicroelectronics, Crolles, France.

References

[1] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, "Dmp: Deterministic shared memory multiprocessing," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 85–96. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508255

[2] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. 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*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 53–64. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736029

[3] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble, "Deterministic process groups in dos," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–16.

[4] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir, "Parallel programming must be deterministic by default," in *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, ser. HotPar'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 4–4. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855591.1855595

[5] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging operating systems with time-traveling virtual machines," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247361

[6] M. Ronsse, K. D. Bosschere, and J. C. D. Kergommeaux, "Execution replay and debugging," in *Proceedings of the Fourth International Workshop on Automated Debugging (AADEBUG2000*, 2000.

[7] 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*, vol. 13(4), 2009.

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

[9] G. Pokam, C. Pereira, S. Hu, A.-R. Adl-Tabatabai, J. Gottschlich, J. Ha, and Y. Wu, "Coreracer: A practical memory race recorder for multicore x86 tso processors," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 216–225. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155646

[10] X. Qian, H. Huang, B. Sahelices, and D. Qian, "Rainbow: Efficient memory dependence recording with high replay parallelism for relaxed memory model," in *HPCA*, 2013, pp. 554–565.

[11] D. Subhraveti and J. 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*, ser. SIGMETRICS '11. New York, NY, USA: ACM, 2011, pp. 109–120. [Online]. Available: http://doi.acm.org/10.1145/1993744.1993757

[12] A. Basu, J. Bobba, and M. D. Hill, "Karma: Scalable deterministic record-replay," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011, pp. 359–368. [Online]. Available: http://doi.acm.org/10.1145/1995896.1995950

[13] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. 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*, ser. CGO '10. New York, NY, USA: ACM, 2010, pp. 2–11. [Online]. Available: http://doi.acm.org/10.1145/1772954.1772958

[14] Y. 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*. ACM Press, 2005, pp. 69–76.

[15] J.-D. Choi and H. Srinivasan, "Deterministic replay of java multithreaded applications," in *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, ser. SPDT '98. New York, NY, USA: ACM, 1998, pp. 48–59.

[16] Y. Chen, W. Hu, T. Chen, and R. Wu, "Lreplay: A pending period based deterministic replay scheme," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 187–197. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815985

[17] O. Laadan, N. Viennot, and J. Nieh, "Transparent, lightweight application execution replay on commodity multiprocessor operating systems," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 1, pp. 155–166, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1811099.1811057

[18] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy, "Doubleplay: Parallelizing sequential logging and replay," *SIGPLAN Not.*, vol. 47, no. 4, pp. 15–26, Mar. 2011. [Online]. Available: http://doi.acm.org/10.1145/2248487.1950370

[19] P. Montesinos, M. Hicks, S. T. King, and J. 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*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 73–84. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508254

[20] D. Geels, G. Altekar, S. Shenker, and I. Stoica, "Replay debugging for distributed applications," in *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ser. ATEC '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 27–27. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267359.1267386

[21] 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*, 2000, pp. 219–227.

[22] N. Hunt, T. Bergan, L. Ceze, and S. D. Gribble, "Ddos: Taming nondeterminism in distributed systems," *SIGPLAN Not.*, vol. 48, no. 4, pp. 499–508, Mar. 2013. [Online]. Available: http://doi.acm.org/10.1145/2499368.2451170

[23] M. Ronsse and W. Zwaenepoel, "Execution Replay for TreadMarks," in *PDP*. IEEE Computer Society, 1997, pp. 343–350.

[24] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution replay of multiprocessor virtual machines," in *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '08. New York, NY, USA: ACM, 2008, pp. 121–130. [Online]. Available: http://doi.acm.org/10.1145/1346256.1346273

[25] D. R. Hower and M. D. Hill, "Rerun: Exploiting episodes for lightweight memory race recording," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 265–276. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2008.26

[26] S. Narayanasamy, C. Pereira, and B. Calder, "Recording shared memory dependencies using strata," *SIGPLAN Not.*, vol. 41, no. 11, pp. 229–240, Oct. 2006. [Online]. Available: http://doi.acm.org/10.1145/1168918.1168886

[27] P. Montesinos, L. Ceze, and J. Torrellas, "Delorean: Recording and deterministically replaying shared-memory multiprocessor execution ef?ciently," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 289–300. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2008.36

[28] S. Narayanasamy, G. Pokam, and B. Calder, "Bugnet: Continuously recording program execution for deterministic replay debugging," *SIGARCH Comput. Archit. News*, vol. 33, no. 2, pp. 284–295, May 2005. [Online]. Available: http://doi.acm.org/10.1145/1080695.1069994

[29] J. Chow, D. Lucchetti, T. Garfinkel, G. Lefebvre, R. Gardner, J. Mason, S. Small, and P. M. Chen, "Multi-stage replay with crosscut," in *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '10. New York, NY, USA: ACM, 2010, pp. 13–24. [Online]. Available: http://doi.acm.org/10.1145/1735997.1736002

[30] J. C. Maeng, J.-I. Kwon, M.-K. Sin, and M. Ryu, "RT-Replayer: A Record-Replay Architecture for Embedded Real-time Software Debugging," in *Proceedings of the 2009 ACM Symposium on Applied Computing*, ser. SAC '09. New York, NY, USA: ACM, 2009, pp. 1670–1675. [Online]. Available: http://doi.acm.org/10.1145/1529282.1529656

[31] G. Gracioli and S. Fischmeister, "Tracing and recording interrupts in embedded software," *J. Syst. Archit.*, vol. 58, no. 9, pp. 372–385, Oct. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.sysarc.2012.06.003

[32] S. Choudhuri and T. Givargis, "Flashbox: A system for logging non-deterministic events in deployed embedded systems," in *Proceedings of the 2009 ACM Symposium on Applied Computing*, ser. SAC '09. New York, NY, USA: ACM, 2009, pp. 1676–1682. [Online]. Available:

http://doi.acm.org/10.1145/1529282.1529657

[33] M. Xu, R. Bodik, and M. D. Hill, "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ser. ISCA '03. New York, NY, USA: ACM, 2003, pp. 122–135. [Online]. Available: http://doi.acm.org/10.1145/859618.859633

[34] M. Ronsse and W. Zwaenepoel, "Execution Replay for TreadMarks," in *PDP*. IEEE Computer Society, 1997, pp. 343–350.

[35] L. Levrouw, K. Audenaert, and J. 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.* IEEE, 1994, pp. 471–478.

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

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

[38] K. Georgiev and V. Marangozova-Martin, "Deterministic Partial Replay for MPSoC Debugging," INRIA, Tech. Rep. 5815, 2014.

[39] STMicroelectronics, "KPTrace," http://www.stlinux.com/devel/traceprofile/kptrace.

[40] J. C. de Kergommeaux, B. Stein, and P. Bernard, "Pajé, an interactive visualization tool for tuning multi-threaded parallel applications," *Parallel Computing*, vol. 26, no. 10, pp. 1253 – 1274, 2000.

[41] S. Tomar, "Converting Video Formats with FFmpeg," 2006.

[42] "FFMPEG Website." [Online]. Available: http://ffmpeg.org/ffmpeg.html

[43] Y. Ahn, Y.-S. Hwang, and K.-S. Chung, "Flexible framework for dynamic management of multi-core systems," in *SoC Design Conference (ISOCC), 2009 International*, Nov 2009, pp. 237–240.

[44] A. Pura and C. V. Raghu, "Design of a wireless adapter for multimedia projectors," in *Wireless Communication, Vehicular Technology, Information Theory and Aerospace Electronic Systems Technology (Wireless VITAE), 2011 2nd International Conference on*, Feb 2011, pp. 1–4.

[45] J.-D. Fekete, "Visual analytics infrastructures: From data management to exploration," *Computer*, vol. 46, no. 7, pp. 22–29, July 2013.

[46] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[47] K. Pouget, P. L. Cueva, M. Santana, and J.-F. Mhaut, "Interactive debugging of dynamic dataflow embedded applications." in *IPDPS Workshops*. IEEE, 2013, pp. 345–354.

[48] S. Griffith, "On-chip debug units maximize real-time embedded systems." [Online]. Available: http://www.aeroflex.com/ams/news/articles/cos-LEONOnChipdebugArticle.pdf

[49] J. G. Ganssle, *The art of designing embedded systems*. Burlington (US), Oxford (GB): Newnes, 2008. [Online]. Available: http://opac.inria.fr/record=b1128837

[50] JTAG, "Joint test action group." [Online]. Available: http://www.siliconfareast.com/jtag.htm