MPSoC Zoom Debugging: A Deterministic Record-Partial Replay Approach

Kiril Georgiev, Vania Marangozova-Martin

MPSoC Zoom Debugging: A Deterministic Record-Partial Replay Approach

Kiril Georgiev*†‡ and Vania Marangozova-Martin*†‡
*Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
†CNRS, LIG, F-38000 Grenoble, France
‡Email: kiril.georgiev.sf@gmail.com, vania.marangozova@imag.fr

***Abstract*—This work presents a debugging methodology for MPSoC based on deterministic record-replay. We propose a general model of MPSoC and define a debugging cycle targeting 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 replay a part of the application during a specific execution interval. The proposed mechanisms are connected to GDB and allow for a visual representation of the considered part of the trace. The approach is validated on two execution platforms and two multimedia applications.**

I. 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 has called 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 and validation. Indeed, parallel computations and concurrent data accesses makes software execution *non deterministic*. As a consequence, software validation is faced with the problem of detecting and rooting the causes of non deterministic errors which are hard to observe and reproduce. The problem is even more emphasized by the important number of components (threads, tasks, processes) taking part in an execution and their possible interactions.

In this paper we describe our approach to debugging non deterministic embedded software. We propose a record-replay mechanism in which non deterministic errors are first captured in execution traces and then tracked through debugging a deterministic replay of the recorded traces. The main contribution of this work is a debugging methodology reducing the error search space by applying spatial and temporal selection criteria. We describe our implementation, ReDSoC, and its trace collection, trace visualization, deterministic replay and partial replay support. ReDSoC has been validated on two different platforms with two multimedia applications.

The rest of the paper is organized as follows. After presenting the general methodology for debugging based on deterministic partial record-replay (Section II), we describe the general principles of our ReDSoC system and its implementation (Sections III and IV). We illustrate its application in the cases of a Tetris and a Video Mosaic application and discuss performance issues (Section V). Related work and future perspectives conclude the paper (Sections VI and VII).

II. Record-Replay Debugging Methodology

We propose the following methodology (Figure 1).

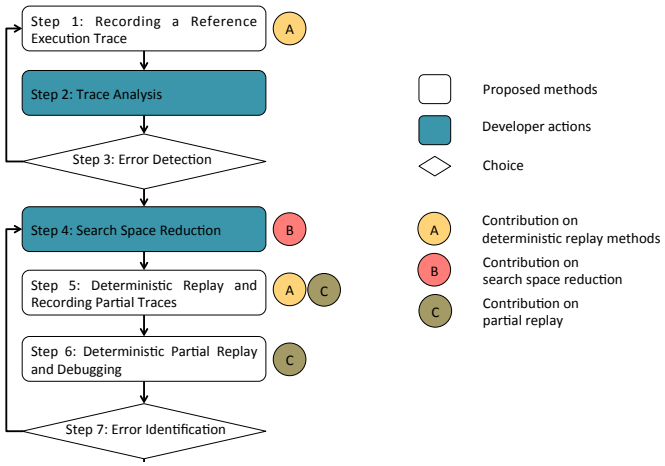

Fig. 1. Debugging Cycle

- *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 non deterministic 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 non deterministic 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. Yet, the recorded data is sufficient for a deterministic replay. The choice of target non deterministic phenomena to debug and the identification of adapted replay algorithms represents our first contribution.

- *Step 2: Trace Analysis*
  The step is performed by the developer who debugs the MPSoC software. Using different available tools but mainly 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 non deterministic 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 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.
- *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. 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 want 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.

III. REDSOC OVERVIEW

To define the debugging selection criteria(Section III-B), we use an abstract MPSoC architecture model (Section III-A). We have studied existing algorithms and based our solution on the ones with minimal intrusion in terms of tracing overhead (Section III-C).

A. MPSoC Model

Our work is based on the generic hardware model showed in Figure 2.

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

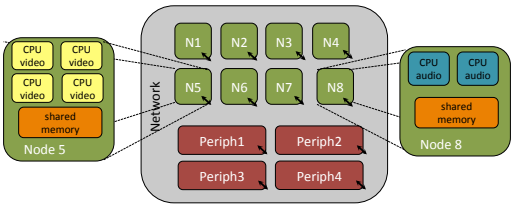

Fig. 2. MPSoC Hardware Architecture

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.

The network connections organize components in a hierarchical way.

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 using interruptions.

B. Partial Replay

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

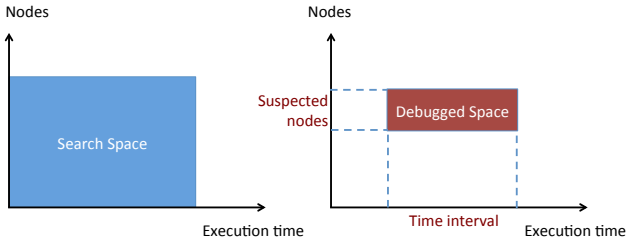

Fig. 3. Search Space Reduction

With space reduction, the idea is to isolate 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 non suspected nodes are called the *correct* nodes.

To isolate suspected nodes from the correct nodes, 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 nodes 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.

The reduction of the search space concerning time 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 tools 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.

C. Record-Replay Algorithms

We focus on replaying shared data accesses, network communications and I/O operations.

Given that recording all accesses to shared data implies a prohibitive execution overhead [1], our record-replay mechanisms 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 [2]. The algorithm uses Lamport clocks to identify accesses to different synchronization structures by different execution flows.

Two network communication situations may be non deterministic. First, when multiple sources send messages to a single destination. The reception order may depend on numerous factors like network protocols, connection speed, routing, system load, etc. The second situation concerns non blocking reception operations. In this case, the reception operation relies on a verification of the data availability (*probe*) which is itself non deterministic.

To trace and deterministically replay network communications, we have used the solution proposed in [3], [4]. 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.

Output operations have no effect on replay techniques. Input operations, however, are important, as they influence the execution path of MPSoC software. Input operations are based either on interrupts, or on busy waiting (polling). Interrupts, however, are a challenge to embedded record-replay [5].

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.

IV. ReDSoC Implementation

The architecture of our prototype is given on Figure 4.

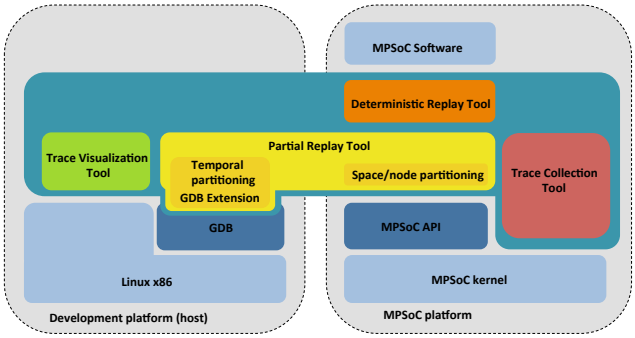

Fig. 4. ReDSoC Architecture

We consider a standard configuration in which part of the debugging operations are deported on 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 1), 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 the POSIX standard and includes basic functions for execution flow management, synchronization, network communications and I/O.

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 implements the algorithms presented in Section III using as basis the MPSoC API. Shared data accesses are managed through tracking the synchronization operations of the API. Network communications

are targeted using our message-based communication. Finally, I/O are addresses 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 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 [6] 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 [7] format and adapted the KPTrace viewer to take into account its visualization.

An example of visualization is shown on Figure 5. The x dimension gives the time progression. The y dimension represent containers, in this case tasks. The links, represented using 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 for each peripheral device.

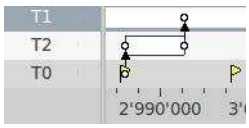

Fig. 5. A fragment of trace visualization

V. Debugging Non Deterministic Multimedia Applications

We have validated our approach in two settings: the debugging of a real-time game application on an MPSoC platform (Section V-A) and the debugging of a video-decoding application on a NUMA platform (Section V-C).

A. 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.

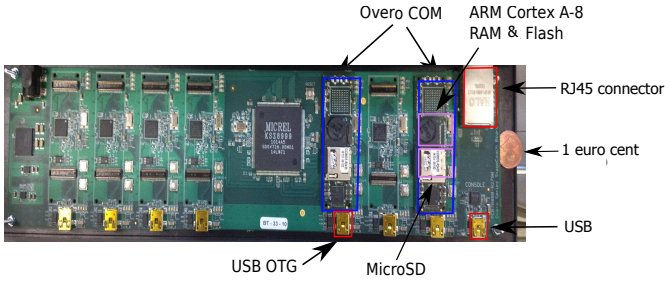

Fig. 6. Stagecoach board with two Overo FE COM nodes

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 7). The application's size is about 0,7MB and contains about 15000 lines of code. It is executed by two tasks run respectively on the two MPSoC nodes.

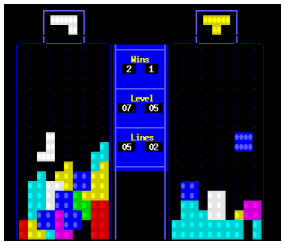

Fig. 7. Two Player Tetris.

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

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

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

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 8).

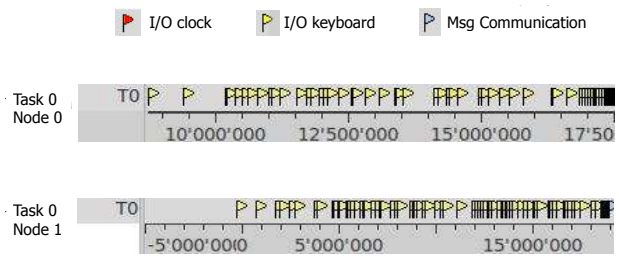

Fig. 8. Visualization of the Tetris Reference Trace

As, in our case, the node to fail is node 1, we suspect this node and choose it 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 9). We select the small end time interval containing three operations reading the system clock, 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.

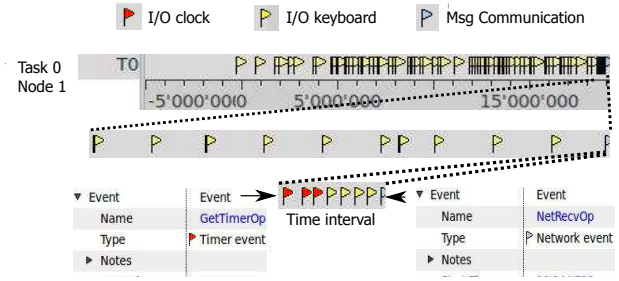

Fig. 9. Time Interval Selection

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 10).

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

```
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
#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
#2  0x0000000000412812 in replayIOsize ()
    at /replay_mechanism/src/replay.c:146
#3  0x000000000041e6c4 in gettm (a=0)
    at /game/src/timer.c:88
#4  0x0000000000415700 in play_round ()
    at game/src/2p.c:506
#5  0x000000000415c6b 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 ?? ()
```

Fig. 10. Partial Debugging of the MPSoC Tetris Application

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 form 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.

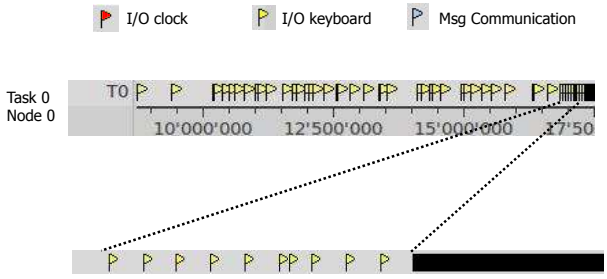

Fig. 11. Considering a Different Node and a Different Time Interval

B. 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 [8], [9].

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 [10] and FFSERVER [11] 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 12). We have re-engineered the code to redirect all Linux function calls to calls to our MPSoC API.

Fig. 12. Video Mosaic Application

The video mosaic application exhibited a non deterministic bug. During some executions, one or more videos were not visible. By tracing one of these executions, we captured the situation showed on Figure 13. 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.

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

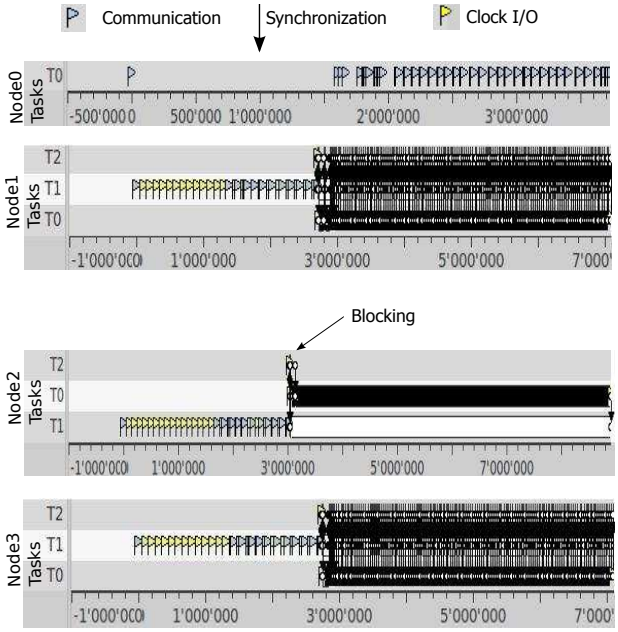

Fig. 13. Visualization of Captured Traces.

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.

C. 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 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 I.

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

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 I. INTRUSION MEASURES

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 upper software layers.

As non deterministic 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.

VI. RELATED WORK

There are numerous deterministic record-replay solutions that focus and limit themselves on different sources of non determinism. In a shared memory setting, projects reproduce scheduling decisions only [12] or consider data races. The latter consider all shared data accesses [13], [14], [15] or the accesses to synchronization structures [16]. Alternative approaches relax the exact replay of data accesses and focus on application outputs [17]. Others eliminate non determinism by using an adapted execution support [18]. In a distributed setting, record-replay solutions focus on the data exchanges among nodes [19], [20]. To apply to realistic embedded platforms, ReDSoC considers all sources of non determinism and combines record-replay techniques from both shared and distributed memory settings.

In the domain of embedded systems, record-replay mechanisms for multi-tasking embedded systems mainly focus on the reproduction of context switches [21], [22], [5]. The works investigate the unique identification of context switches, needed for a precise replay, as well as various algorithms for efficient computing of the system state fingerprint. This approach can be used in hard real-time embedded systems but does not apply to multi-core concurrent executions which are considered in ReDSoC.

Record-replay mechanisms strive for a trade-off between cost of implementation, precision and generality. Indeed, hardware-based mechanisms [23] impose minimal intrusion on the traced system but require costly non commodity hardware. Virtual machine mechanisms [24], [13] provide for a comparable level of detail but rarely consider multi-processor platforms. In addition, their cost is prohibitive for embedded systems. System mechanisms [14] provide for transparent record-replay which does not require application modification. They are, however, tightly coupled with the specific operating system they consider. ReDSoC is at the level of application and library mechanisms [20], [25] which require some modification (instrumentation, recompilation) of the target application but provide better portability.

Partial replay has been considered in parallel and distributed systems which exhibit too much components and interactions for a total record-replay. Recent works [26], [27] on many-core High Performance Computing (HPC) architectures reproduce selected groups of processes. However, as their mechanisms are based on their programming models API, they cannot be applied to embedded system environments. As for distributed systems [25], [28], existing partial replay solutions limit themselves to considering a single node.

VII. CONCLUSION

In this paper we argue that with the increasing scale and complexity of embedded systems, classic debugging techniques cannot be applied "as is". Non deterministic systems with numerous components need our debugging methodology which applies space and time reduction criteria to the error search space. For human comprehension, debugging should indeed be able to focus on a specific part of the target software and consider a limited time interval. We have shown the usefulness of this approach in our experiences with two multimedia applications on two different platforms. The debugging experiences have proven successful and our system has performed with minimal intrusion.

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".

Intrusion is a major issue in record-replay systems and the usual answer is to provide ad-hoc solutions for minimizing

the execution overhead. However, modeling and formally estimating the cost of a given tracing/replaying technique will allow for cost predictions and will greatly facilitate the choice between different solutions. This would also be the basis for provisioning hardware resources for record/replay in embedded systems.

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.

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 [29] but the approach should be also investigated for post-mortem analysis.

ACKNOWLEDGMENT

This work has been done in a collaboration with the IDTEC department of STMicroelectronics, Crolles, France.

REFERENCES

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

[2] 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.

[3] 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.

[4] 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.

[5] G. Gracioli and S. Fischmeister, "Tracing and Recording Interrupts in Embedded Software," *Journal of Systems Architecture*, vol. 58, pp. 372–385, Oct 2012.

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

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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*. ACM, 2010, pp. 2–11.

[16] M. Ronsse and K. De Bosschere, "Recplay: a fully integrated practical record/replay system," *ACM Trans. Comput. Syst.*, vol. 17, no. 2, pp. 133–152, 1999.

[17] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. Chen, and J. Flinn, "Respec: efficient online multiprocessor replayvia speculation and external determinism," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1. ACM, 2010, pp. 77–90.

[18] 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.

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

[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*. USENIX Association, 2006, pp. 27–27.

[21] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson, "Replay debugging of real-time systems using time machines," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 288.2–.

[22] J. Maeng, J. Kwon, M. 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*. ACM, 2009, pp. 1670–1675.

[23] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, "Dmp: Deterministic shared memory multiprocessing," *SIGARCH Comput. Archit. News*, vol. 37, no. 1, pp. 85–96, Mar. 2009. [Online]. Available: http://doi.acm.org/10.1145/2528521.1508255

[24] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen, "Revirt: enabling intrusion analysis through virtual-machine logging and replay," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 211–224, 2002.

[25] Y. Saito, "Jockey: A user-space library for record-replay debugging," in *In AADEBUG05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*. ACM Press, 2005, pp. 69–76.

[26] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. Voelker, "Mpiwiz: Subgroup reproducible replay of mpi applications," *ACM SIGPLAN Notices*, vol. 44, no. 4, pp. 251–260, 2009.

[27] F. Gioachin, G. Zheng, and L. Kalé, "Robust non-intrusive record-replay with processor extraction," in *Proceedings of the 8th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*. ACM, 2010, pp. 9–19.

[28] 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

[29] 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.