Deterministic Partial Replay for MPSoC Debugging

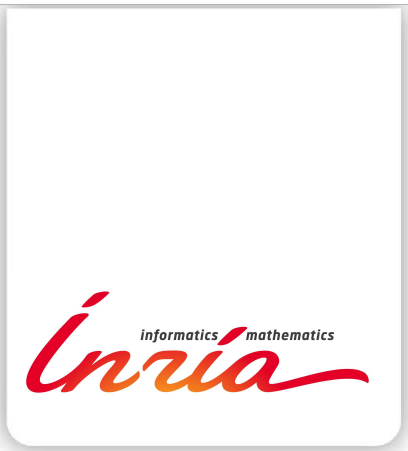
Kiril Georgiev, Vania Marangozova-Martin

Deterministic Partial Replay for MPSoC Debugging

Kiril Georgiev, Vania Marangozova-Martin

RESEARCH REPORT
N° 8515
March 2014
Project-Team MESCAL

ISSN 0249-6399 ISRN INRIA/RR--8515--FR+ENG



Deterministic Partial Replay for MPSoC Debugging

Kiril Georgiev[*†], Vania Marangozova-Martin[‡†]

Project-Team MESCAL

Research Report n° 8515 — March 2014 — ?? pages

Abstract: This work reports on a debugging methodology for MPSoC based on deterministic record-replay. It defines a general model of MPSoC, identifies the major sources for non determinism and selects a set of adapted algorithms for the record and replay of non deterministic errors. The contribution of this work the definition of a debugging cycle targeting errors by applying temporal and spatial selection criteria. is the proposal of and . 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 a visual representation of the considered part of the trace. The approach has been validated on two execution platforms and two multimedia applications.

Key-words: MPSoC, non determinism, deterministic record-replay, debugging, methodology

* kiril.georgiev.sf@gmail.com
† Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France, CNRS, LIG, F-38000 Grenoble, France
‡ vania.marangozova-martin@imag.fr

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

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

Ré-exécution déterministe partielle pour le débogage de MPSoC

Résumé :

Ce rapport présente une méthodologie de débogage pour les systèmes MPSoC basée sur l'enregistrement et la ré-exécution déterministe de traces d'exécution. Ce travail propose un modèle général des systèmes MPSoC, identifie les principales sources de non-déterminisme et propose l'application d'algorithmes adaptés pour l'enregistrement et la ré-exécution d'erreurs non-déterministes. L'originalité du travail réside dans la définition d'un cycle de débogage permettant de cibler la recherche des erreurs en appliquant des critères de sélection spatiale et temporelle. La sélection spatiale consiste à ne considérer qu'une partie de l'application en exécution. La sélection temporelle permet de ne considérer qu'un intervalle spécifique d'exécution. Les mécanismes sont connectées à l'outil de débogage standard GDB tout en fournissant une représentation visuelle de la portion de trace considérée. L'approche est validée sur deux types de plateformes et avec deux applications multimédia.

Mots-clés : MPSoC, non-déterminisme, ré-exécution déterministe, débogage, méthodologie

1 Introduction

Recent years have witnesses a tremendous development of embedded systems [?, ?, ?, ?]. They find their place in numerous domains in our everyday life like transports, domotics and telecommunications. This omni-presence has called for new design methods targeting more complex applications, more efficiency and yet 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, the possibility to execute multiple treatments in parallel and to have 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 report we describe our approach to debugging non deterministic embedded software. Our work is at the intersection of the two classical validation techniques of interactive debugging [ref] and execution trace analysis [ref]. We propose a record-replay mechanism [ref] in which non deterministic errors are first captured in recorded execution traces and then tracked through debugging a deterministic replay of the recorded traces. The main contribution of this work is a debugging methodology allowing for zooming into suspicious software parts by applying spatial and temporal selection criteria. The methodology is designed to be scalable, i.e. support MPSoC with a great number of components, and efficient, i.e induce minimal execution overhead.

The proposed debugging methodology is based on the following basic bricks:

- *Target Sources for Non Determinism*
  Modern MPSoC software include parallel, distributed and time-constrained interactions. As a consequence, they need to face the global set of non deterministic execution situations identified in other computation domains. Namely, MPSoC non deterministic executions may be due to concurrent data accesses, to I/O activities, to message-oriented communication or to scheduling. Our methodology proposes a trade-off between precision and intrusion by targeting not all but a set of non deterministic phenomena.

- *Algorithms for Deterministic Record-Replay*
  After a study of the state of the art, we have selected a set of algorithms for recording and reproducing non deterministic phenomena in MPSoC. The choice is based on the algorithms' performance characteristics and their compatibility with the efficiency requirements in MPSoC. Typically proposed in the parallel systems domain, our work explores their applicability to MPsOC.

- *General Architectural Model for MPSoC*
  Considering architectural trends in modern MPSoC, we abstract from heterogeneity and propose a general model representing major architecture principles. This model allows for defining a general software API [1] used in the definition of the deterministic partial record-replay mechanism.

[1] Application Programming Interface

- *Software Partitioning using Spatial and Temporal Criteria*
  Even with the most efficient mechanisms for deterministic record replay, the number of MPSoC interacting components makes global debugging a daunting task. To reduce the error search space, our methodology allows for considering only a part of the MPSoC software (spatial reduction) during a specific execution interval (temporal reduction). Spatial reduction exploits our architectural MPSoC model, while temporal reduction is based on the recorded traces timeline.

- *Debugging Cycle with Partial Replay*
  Our methodology defines an iterative debugging cycle during which the developer may replay different parts of the recorded software execution. During the replay, it is possible to capture additional execution information to zoom in and analyze it more in detail.

The rest of the report is organized as follows. Section **??** presents the state of the art concerning record-replay techniques. Section **??** introduces the general concepts behind our proposal. It details the debugging cycle phases, presents the proposed MPSoC model and sketches the used record-replay algorithms. The implementation of these features in our prototype system ReDSoC [2] is detailed in Section **??**. Section **??** describes our experiences with RedSoC debugging on two execution platforms and two multimedia applications. Finally, Section **??** concludes and gives future perspectives.

2 Record-Replay of Non Deterministic Phenomena

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 [**?**] or consider data races. The latter consider all shared data accesses [**?**, **?**, **?**, **?**] or the accesses to synchronization structures [**?**]. Alternative approaches relax the exact replay of data accesses and focus on application outputs [**?**, **?**]. Others eliminate non determinism by using an adapted execution support [**?**]. In a distributed setting, record-replay solutions focus on the data exchanges among nodes [**?**, **?**, **?**]. 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 [**?**, **?**, **?**, **?**]. 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 [Scribe] impose minimal intrusion on the traced system but require costly non commodity hardware. Virtual machine mechanisms [**?**, **?**] provide for a comparable level of detail but rarely consider multi-processor platforms. In addition, their cost is prohibitive for embedded systems. System mechanisms [**?**] provide for transparent record-replay which does not require application modification. Their are, however, tightly coupled with the specific operating system they consider. ReDSoC is at the level of application and library mechanisms [**?**, **?**] which impose some modification (instrumentation, recompilation) of the target application but provide for better portability.

[2] **Re**cord-Replay for **D**eterministic **SoC** Debugging

Partial replay has been considered in parallel and distributed systems which exhibit too much components and interactions for a total record-replay. Recent works [**?**, **?**] 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 [**?**, **?**], existing partial replay solutions limit themselves to considering a single node.

3 ReDSoC Overview

In this section, we present RedSoC, our prototype system for deterministic partial record-replay of MPSoC. We first describe the proposed debugging methodology, enumerate its different steps and explicitly point out our contributions (Section **??**). Before presenting the logic for our partial replay mechanism (Section **??**), we introduce our general MPSoC model (Section **??**). Section **??** gives details about the chosen record-replay algorithms.

3.1 Debugging Cycle

The proposed debugging methodology is represented in the Figure **??**. It is composed of the following steps:

- *Step 1: Recording a Reference Execution Trace*
  During this step, the execution of the whole MPSoC software is recorded to produce reference execution traces. 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. The design aims at recording minimal but sufficient data. It limits the volume of recorded data 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 analyses the trace in 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 Step1.

- *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. 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.

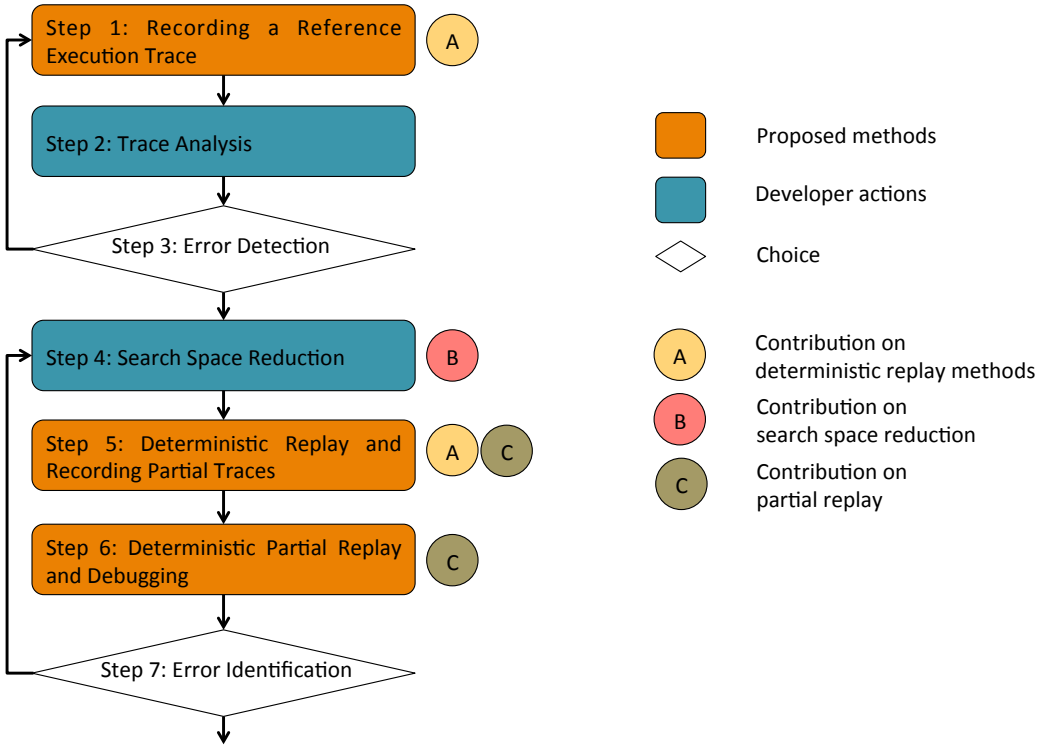


Figure 1: Debugging Cycle

- *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 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.

3.2 MPSoC Model

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

The different components are include processors, memory blocs, peripherals and a communication network.

- *Processors*

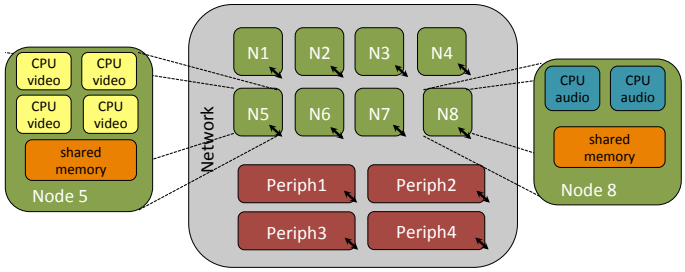


Figure 2: MPSoC Hardware Architecture

Processors[3] are organized in a two-level hierarchy. There may be heterogenous processors but homogenous processors form groups we call nodes. Thus there may be a node with audio processing unit and another specialized in video decoding.

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

- *Peripherals*
  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.

- *Communication Network*
  The network connexions organize the other 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 poling, or using interruptions.

3.3 Partial Replay

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

The reduction of the search space concerning the software architecture is based on our model of execution flows deployed on MPSoC nodes. The idea is to isolate a set of nodes on which the

[3]We call processors all computational units including general purpose processors, cores or accelerators.

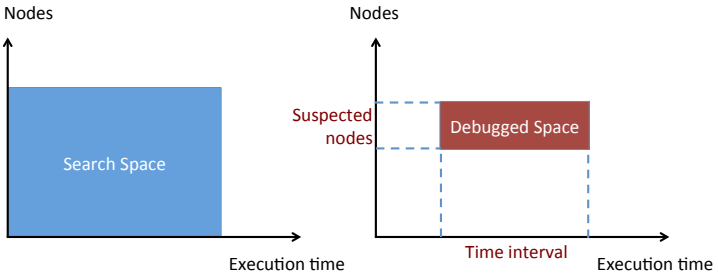


Figure 3: Search Space Reduction

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 debugged, the *suspected* or the 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, reexecuted 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.

3.4 Record-Replay Algorithms

In this section we briefly present our motivation and the chosen algorithms for deterministic replay for shared data accesses, network communications, I/O operations and scheduling.

3.4.1 Deterministic Replay for Shared Data Accesses

Given that recording all accesses to shared data implies a prohibitive execution overhead [**?**], 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 [**?**]. The algorithm uses Lamport clocks to identify accesses to different synchronization structures by different execution flows. There is one scalar clock per synchronization structure (LC_{ss}) and one scalar clock per execution flow f (LC_f). When there is an access to a synchronization structure SS by an execution flow f, both clocks LC_{ss} and LC_f advance using the rule $LC_f = LC_{ss} = max(LC_f, LC_{ss}) + 1$. The

trace of the execution flow records the identifier of the synchronization structure SS, as well as the couple of the old and new values of the LC_f clock.

During replay, the accesses to the synchronization structure SS are forced to follow the recorded order. To this purpose, the replay mechanism considers the recorded LC_f values. It identifies the next access to the synchronization structure by deciding of the next expected value of the clock which is the smallest one. The execution flows trying to access the structure while not having the smallest LC_f are blocked.

3.4.2 Deterministic Replay for Network Communications

Two network communication situations may be non deterministic. The first concerns the case when multiple sources send messages to a single destination. The reception order may depend on numerous factors like network protocols, connexion 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 [**?**, **?**]. This solution has minimal intrusion as it traces only race reception operations.

For blocking network communications, the detection of race reception primitives is based on vector clocks. For a system of N execution flows, vector clocks have N components. For execution flow E_i the i-th component is its scalar clock and accounts for the number of local operations. All other components account for the operations that happened within the other execution flows and that causally precede the current operation. Thus, if there are two receive operations on the same execution flow, one from E_a with timestamp V_a and another from E_b with vector stamp V_b, if the a-th component of V_a is bigger than the a-th component of V_b, the operations are independent i.e race receptions. For the purposes of deterministic replay, the receiving execution flow traces the identifier of the sending flow, as well as the number of current receptions. During replay, if a message is received but its source does not correspond to the one recorded, the reception is delayed.

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). For positive probes, the trace is to contain the identifier of the source execution flow, as well as the size of the received data. During replay, a receive operation is delayed, as long as the source identifier does not match, or the data is not available.

3.4.3 Deterministic Replay for Scheduling Operations

The scheduling mechanism is responsible for deciding which execution flow is to be executed on a given processor. The scheduling operation may be triggered either by a change of status of an execution flow (terminated or blocked), or by a timing mechanism (time-sharing or priorities). If the first case is natural to record and replay, the timing mechanism is related to interruptions. Given the interrupt frequency and the complexity of recording the exact context for an interrupt [**?**], however, recording interrupts with an acceptable cost in a real MPSoC environment is still a challenge. For this reason, we have decided to leave interrupt replay aside for future investigation.

3.4.4 Deterministic Replay for I/O 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). As we decide to limit the intrusion of our mechanism by not recording interrupts, we consider only polling requests. We suppose that the content of the input data is recorded by specific devices and take care of recording its 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 ReDSoC Implementation

The general architecture of our prototype is represented on Figure **??**.

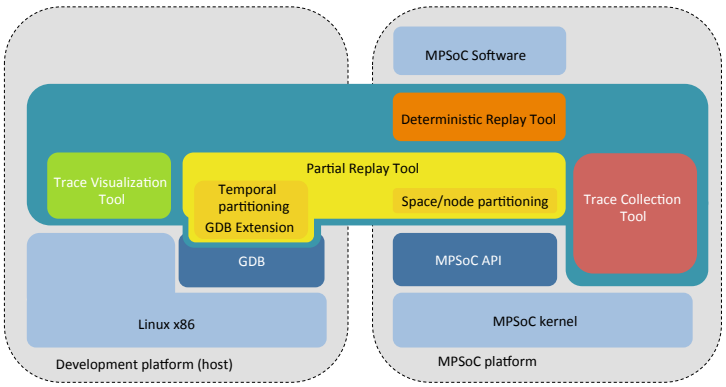


Figure 4: ReDSoC Architecture

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 (Section **??**), a partial replay tool (Section **??**), a trace collection tool (Section **??**) and a deterministic replay tool (Section **??**). The Trace Collection Tool, as well as the temporal selection criteria 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 **??**), as well as the numbers 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 (Section **??**).

4.1 MPSoC API

to explicitly address non deterministic behavior of MPSoC and respect the proposed MPSoC architectural model (cf. Section **??**), we have defined our own MPSoC API. Our API is inspired by the the POSIX standard [**?**]. It includes basic functions for execution flow management, synchronization, network communications and I/O. The API is listed in Figure **??**.

```
/* Task Management */
int taskCreate(task_t *task,void *(*funct)(void *),void *args, int node);
int taskJoin(task_t *task);

/* Synchronization */
int synCreateMutex/synDestroyMutex(mutex_t *m);
int synCreateCond/synDestroyCond(cond_t *c);
int synLock(mutex_t *m);
int synUnlock(mutex_t *m);
int synWait(cond_t *c, mutex_t *m);
int synSignal(cond_t *c);

/* Communication */
int tcpCreate(tcp_socket_t s,int p, int type);
int tcpListen(tcp_socket_t s);
tcp_socket_t tcpAccept(tcp_socket_t s);
int tcpConnect(tcp_socket_t s,node_id num,int p);
int tcpClose(tcp_socket_t s);
int tcpSend(tcp_socket_t s, char *buff, int size);
int tcpRecv(tcp_socket_t s, char *buff, int size);

/* Input/Output */
int pOpen(const char *name);
int pClose(int fd);
int pRead(int fd, void *buf, size_t count);
int pWrite(int fd, void *buf, size_t count);
```

Figure 5: MPSoC API

- *Task Management*
  As in many embedded system kernels [**?**, **?**], in our model, an MPSoC software is executed by several execution flows, called tasks. Our API provides two functions. The first (`taskCreate`) is used for task creation function, the parameters being the function to execute, the arguments for that function and the node on which to launch the task. The second function (`taskJoin`) allows a function to wait for the termination of another task.

- *Synchronization*
  Synchronization functions are applied to tasks executing on the same node. They manage standard mutex and condition synchronization structures with their respective interfaces. The synchronization structures may be created (`synCreate` functions) and destroyed (`synDestroy` functions). Mutexes are manipulated using the `synLock/synUnlock` interface, while conditions are used with `synWait/synSignal`.

- *Communication*
  Communication functions provide for message passing using *sockets* [**?**]. In our model, a communication link is established between two tasks running on two different nodes. The communication protocol is TCP [**?**]. The connection establishment is done via the `tcpConnect` , `tcpListen` and `tcpAccept` functions. It is closed using `tcpClose`. The message sending and receiving functions are respectively `tcpSend` and `tcpRecv`. The choice between blocking or non blocking communication is done at the socket creating with the `tcpCreate` function.

- *Input/Output*
  I/O functions follow the UNIX logic. They provide for reading of (`pRead`) or non-blocking writing (`pWrite`) to peripherals. Peripherals are addressed using the identifier returned by the `pOpen` function.

4.2 Trace Collection Tool

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 of four functions (cf. Figure **??**).

```
int getNodeId();
int getTaskId();
unsigned int getTimestamp();
int trace(int type, char *traceData, int size, vect_t Vector);
```

Figure 6: Tracing API

The first three functions return respectively the identifier of the current node, the identifier of the current task and the node timestamp. The timestamp information is used to the visual representation of the collected traces using the trace visualization tool.

The `trace` function writes an entry in the trace. The `type` parameter is used to distinguish between the different types of entries corresponding to the different non deterministic situations traced. The buffer `traceData` contains the data of size `size` to record. The `Vector` parameter gives a mask indicating what is the structure of the trace entry. At most, the entry contains five fields including a timestamp, a node identifier, a task identifier, an entry type and the actual data.

4.3 Deterministic Replay Tool

The tool for deterministic replay implements the algorithms presented in Section **??**. In the following, we give the implementation details about the three target non-deterministic situations. Shared data accesses are managed through tracking the synchronization operations of our MPSoC API. Network communications are targeted using our message-based communication. Finally, I/O are addresses by the MPSoC file-oriented I/O operations .

4.3.1 Synchronization

In the case of synchronization, our tool records and replays the accesses to the shared synchronization structures, namely the mutexes and the conditions. In the following, we focus on mutexes, as the treatment for conditions is the same.

The access to mutexes and the output of the `synLock` function is non deterministic as the access to critical sections depends on the system scheduling which we cannot control. As a consequence, our tool records and reproduces the access order to the critical section. To do so, the `synLock` function is encapsulated in a `RRsynLock` function, as showed in Figure **??**.

During the record phase, the mutex is taken and the access order is recorded using `TraceMechanism` function. The TraceMechanism creates an entry indicating that its type is related to synchronization and containing the couple of old and new Lamport clock values. During the replay phase, the task is blocked until it is its turn according to the recorded trace. This behavior is implemented in the `WaitSynchVar` function, given in Figure **??**.

A task gains access to a mutex using the `lowestSliceCounterValue` function. The function verifies if the task counter (`SliceCounter[eFlowID]`) is the smallest among all recorded values contained in the shared array `SliceCounter`. When the access is replayed, the data structures for tracking the access order are updated in the `NextSliceCounterValue` function.

```
RRsynLock(mutex_t *m){
...
  if RecordPhase() {
    synLock(m);
    TraceMechanism(m);
  } else
    WaitSynchVar();
    synLock(m);
    NextSliceCounterValue();
...
}
```

Figure 7: Interception of the `synLock( )` function

```
void WaitSynchVar() {
...
  while  (!lowestSliceCounterValue()) {
    blockedTasks++;
    synCondition(&waitCond,&waitSynchVar);
  }
  for (i=0;i<blockedTasks;i++)
    synSignal(&waitCond);
  }
...
}
```

Figure 8: Respecting the recorded synchronization access order

```
//(firstTracedLC,lastTracedLC) is the current traced couple of Lamport clocks

void NextSliceCounterValue() {
...
  if (SliceCounter[eFlowID] == firstTracedLC) {
    SliceCounter[eFlowID] = lastTracedLC;
    GetCoupleFromTrace(&firstTracedLC,&lastTracedLC);
  }
  else
    SliceCounter[eFlowID]++;
  }
...
}
```

Figure 9: Updating synchronization replay structures

4.3.2 Message Passing Communication

Non deterministic situations concerning multiple sends to a single destination, are reflected by the communication protocol implemented using `tcpConnect` and `tcpAccept` functions. Indeed, `tcpAccept` creates the communication link (the socket) with the first task which executes `tcpConnect` with the needed arguments. To reproduce deterministically these situations, we record the (source, destination) couples by intercepting `tcpConnect` and `tcpAccept` and executing respectively the `RRtcpConnect` (Figure **??**) and `RRtcpAccept` (Figure **??**)functions.

```
1  int RRtcpConnect(tcp_socket_t s,node_id num,int p) {
2
3     ret = tcpConnect(s,num,p);
4     MessageProduction(num,buffer,&size);
5     tcpSend(s,buffer,size);
6     return ret;
7  }
```

Figure 10: Deterministic Replay for `tcpConnect`

The role of the `RRtcpConnect` is to send the node identifier to the destination node to allow the tracing of the established communication couple (sending node, receiving node). To do so, the function sends an additional message (`MessageProduction` and `tcpSend`, lines 4 and 5) containing the node identifier.

```
1  int RRtcpAccept(int s) {
2  ...
3    if RecordPhase() {
4       ret = tcpAccept(s);
5       tcpRecv(s,buffer,size);
6       generateTcpAcceptTraceData(traceData,buffer);
7       trace(TcpAcceptTraceType,traceData,tcpAcceptVector);
8    } else if (!SearchTcpSocket(sock,&ret)) {
9      ret = tcpAccept(sock);
10    tcpRecv(sock,buffer,size);
11    while (!compare(getExpectedNodeid(),getRecvNodeid(buffer))) {
12       AddSocket(getRecvNodeid(buffer),sock,ret,getSocketsArray());
13       ret = tcpAccept(sock);
14       tcpRecv(sock,buffer,size);
15    }
16   }
17   return ret;
18 }
```

Figure 11: Deterministic Replay for `tcpAccept`

In a symmetric way, during the record phase, the `tcpAccept` function, received the sending node identifier, established the connection link and generates the trace entry (`generateTcpAcceptTraceData` and `trace` calls on lines 6 and 7) containing the couple of identifiers of the two communicating nodes. For the replay phase, the function manages a buffer of created sockets with the corresponding (sender, receiver) couples. When a connection demand arrives, the function checks whether the socket is already created and available (`SearchTcpSocket`) and returns it. Otherwise, it creates the socket. If the socket identified by a couple (sender, receiver) does not conform to the trace recordings (line 11), the socket is stored in the sockets array via (`AddSocket`).

For non blocking receptions, the trace records the identifier of the receiving task, the number of calls to `tcpRecv` receiving the same size of data, as well as the size itself. The replay executes

a loop waiting for the data to be available.

4.3.3 I/O

Input functions are executed using the `pRead` function. The record phase registers the identifier of the task calling this function, as well the number of calls and the data size. The logic is identical to the records of non blocking `tcpRecv` calls. During replay, the `pOpen` function call is not executed as the input data is recorded and available in a separate trace.

4.4 Partial Replay Tool

to apply the space reduction criterium based on isolating suspected nodes, our partial replay tool needs to monitor and record all communications between normal and suspected nodes. These communications are based on the network communication primitives of our MPSoC API, namely `tcpConnect`,`tcpAccept`, `tcpSend` and `tcpRecv`. In the following, we explain the treatments for `tcpConnect` and `tcpRecv`, the treatments for `tcpAccept` and `tcpSend` being identical.

During the establishment of a connection with a suspected node using `tcpAccept`, the tool verifies if the sender node is a correct one (cf. Figure **??**). If this is the case, the trace mechanism records the node identifier and the respective socket. It also records a counter (`CFC`) which is used to track the order of the communication functions on the suspected node.

```
1  int PartialRecordTcpConnect(tcp_socket_t s, node_id *num, int retTcpConnect) {

2    MessageProduction(num,buffers,&size);
3    tcpRecv(s,bufferr,size);
4    tcpSend(s,buffers,size);
5    if (CorrectNode(bufferr)) {
6      RegisterNodeSocket(bufferr,s);
7      GeneratePartialTcpConnectTraceData(traceData,CFC,retTcpConnect);
8      trace(PartialTcpConnectTraceType,traceData,tcpConnectVector);
9    }
10  CFC++;
11}
```

Figure 12: Tracing `tcpConnect()`

During communication, the function `tcpRecv` is encapsulated in `PartialRecordtcpRecv`. If the reception is on a socket which indicates a connection to a normal node, the operation is traced to record the return value, the received data and the communication operation number.

```
int PartialRecordTcpRecv(tcp_socket_t s, int retTcpRecv, char *buffer) {

  if (registredSocket(s)) {
    GeneratePartialTcpRecvTraceData(traceData,CFC,retTcpRecv,buffer);
    trace(PartialTcpRecvTraceType,traceData,tcpRecvVector);
  }
  CFC++;
}
```

Figure 13: Tracing `tcpRecv()`

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 criterium, 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 followed by a call to the `gdbNotify` function. This function is in charge of comparing the current API call 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.

4.5 Trace Visualization Tool

We have adapted the KPTrace Viewer of STMicroelectronics [**?**] to represent 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.

to visualize our traces, we have first provided for a tool formatting our traces according to the Pajé [**?**, **?**] format. Second, we have modified the KPTrace viewer to allow Pajé format visualization.

Pajé considers a trace as a sequence of events. It is a self-defining format as it allows for definition of the event types and establishment of a type hierarchy. It introduces the concepts of *container*, *state*, *event*, *variable* and *link*.

- *Container* : A container represents an entity with dynamic behavior. It may represent, for example, a processor, a network kink or a thread. It may reference other containers to establish an hierarchy.

  In our tool, the root container is the MPSoC software. This container has child containers representing the nodes of the system. The third level of the hierarchy are the task containers which are referenced by the node containers.

- State : A state is a duration with explicit start and ending. A state models the fact that a container has the same state during an interval of time. The definition of the semantics of a state is left to the developer.

  In our tool, we do not use states.

- Event : Un event is a punctual phenomenon that happens at a given time (timestamp).

  In our tool, events represent the input operations, as well as message receptions.

- Variable : A variable follows the evolution of numerical values. It may represent the value of a measured parameter or contain a calculated value.

  In our tool, variables are not used.

- Link : A link represent an interaction between two containers, characterized with a start and an end timestamps. It is typically used for modeling communications.

In our tool, links represent causal relations. On one hand, they are used to show the successive accesses of tasks to shared synchronization structures. On the other hand, they show the establishment of network connection by relating the calls to `tcpConnect` and `tcpAccept`.

An example of visualization is shown on Figure **??**. 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.

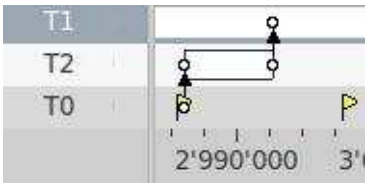


Figure 14: A fragment of trace visualization

5 Validation

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

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) [**?**]. 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.

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 [**?**] 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 **??**). 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.

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.

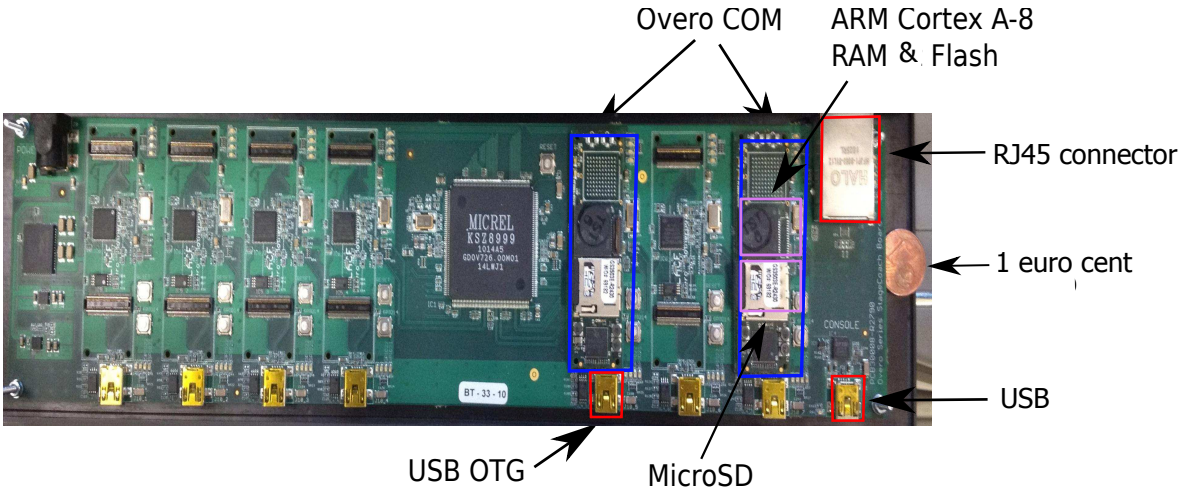


Figure 15: Stagecoach board with two Overo FE COM nodes

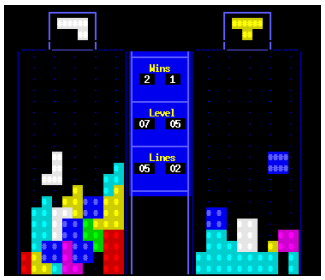


Figure 16: Two Player Tetris.

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 reexecuted several times the Tetris application to obtain a reference trace containing the error (cf. Figure **??**).

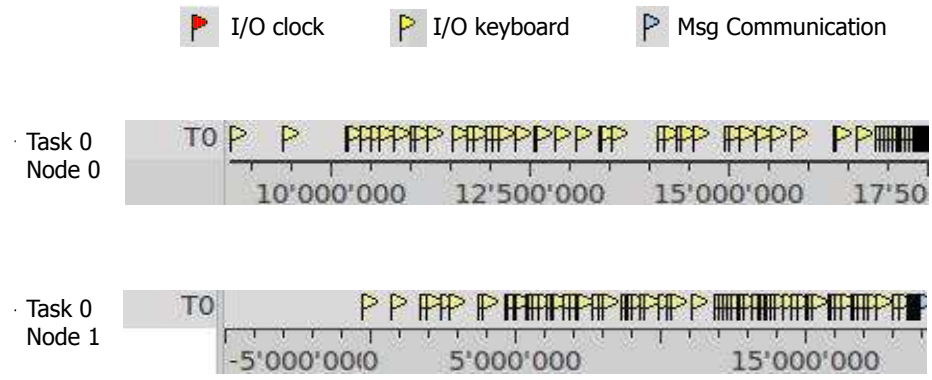


Figure 17: Visualization of the Tetris Reference Trace

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 **??**). 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.

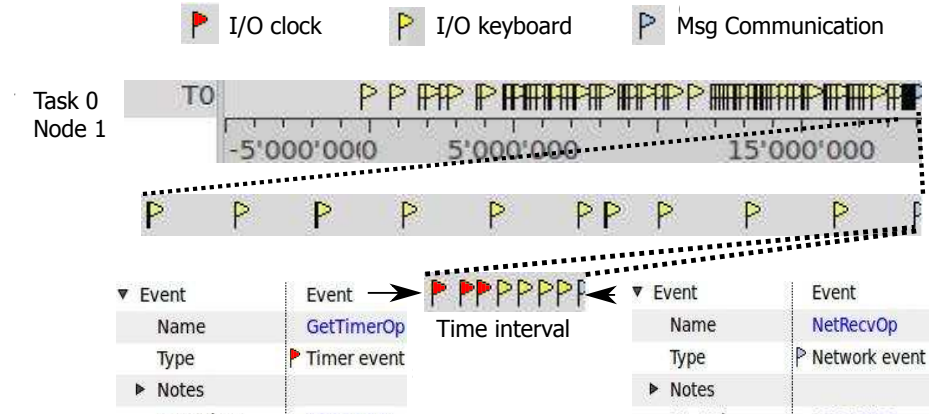


Figure 18: Time Interval Selection

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 **??**).

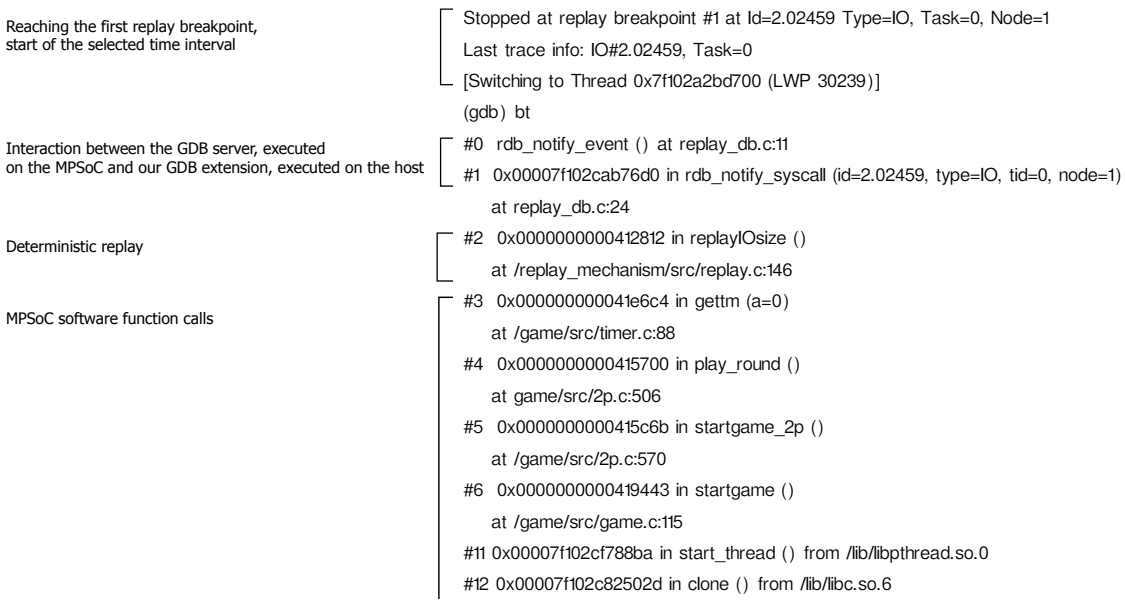


Figure 19: Partial Debugging of the MPSoC Tetris Application

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 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.

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 [**?**, **?**].

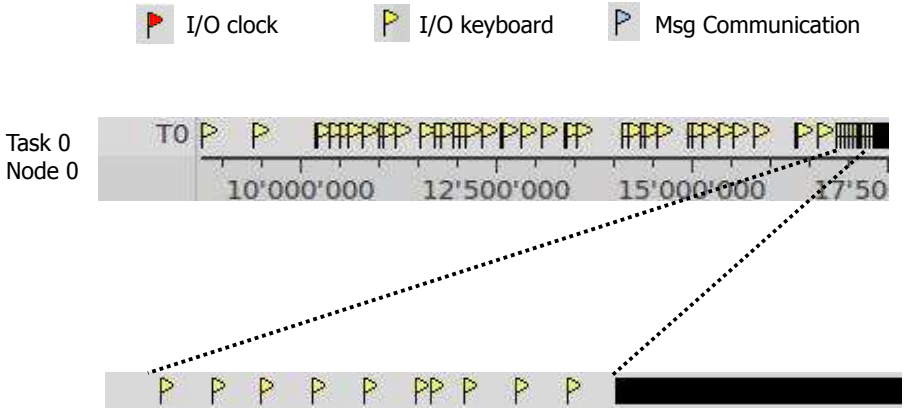


Figure 20: Considering a Different Node and a Different Time Interval

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. The architecture is shown on Figure **??**.

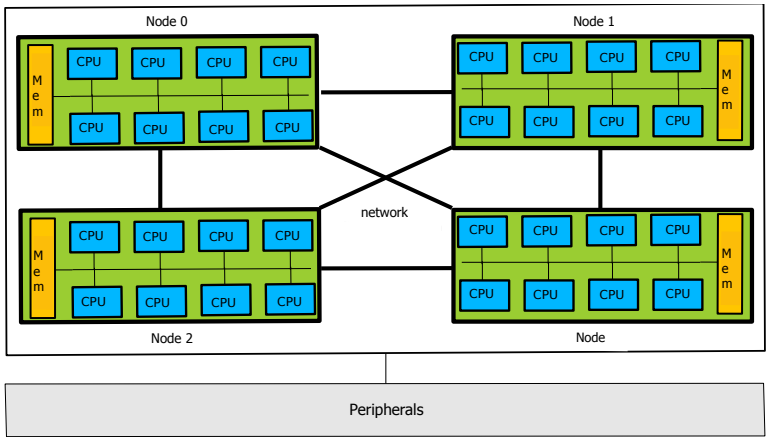


Figure 21: NUMA Architecture

We have used the NUMA resources to represent the MPSoC architectural elements in the following manner:

- MPSoC Nodes. MPSoC nodes and cores are mapped to the NUMA nodes and processors respectively. To do so, we represent the hardware and the software of an MPSoC node by a Linux process. However, the Linux process does not occupy all the resources of the corresponding NUMA node. Part of these resources are reserved for the debugging needs (cf. Figure **??**).

- The memory of an MPSoC node is the memory of the corresponding NUMA node. As the NUMA memory is shared by all NUMA processors, to conform to MPSoC constraints, we use the `libnuma` library to delimit the accessed memory region. Moreover, MPSoC nodes are configured to use small memory partitions (4GB) i.e they do not use the total available NUMA memory (32GB).
- MPSoC Peripherals. We use the NUMA peripherals as MPSoC peripherals. However, in NUMA machines all nodes can access the peripherals. To be more realistic, we have confined the peripheral access to a single MPSoC (respectively NUMA) node.
- MPSoC Trace Port. To trace the operations on an MPSoC node with ReDSoC, we need access to a trace port. To implement this feature, we have chosen to use a distinct memory partition for trace recording (4GB).

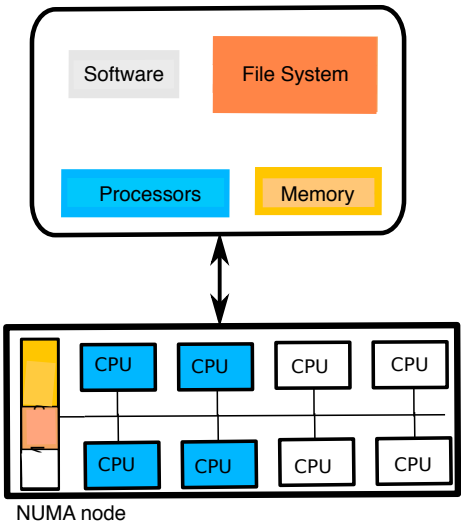


Figure 22: MPSoC Node

In the final experimental setup (Figure **??**), 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* [**?**] 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 [**?**] and FFSERVER [**?**] components. FFSERVER is a video server, receiving video flows through different protocols (e.g. RTP [**?**] or

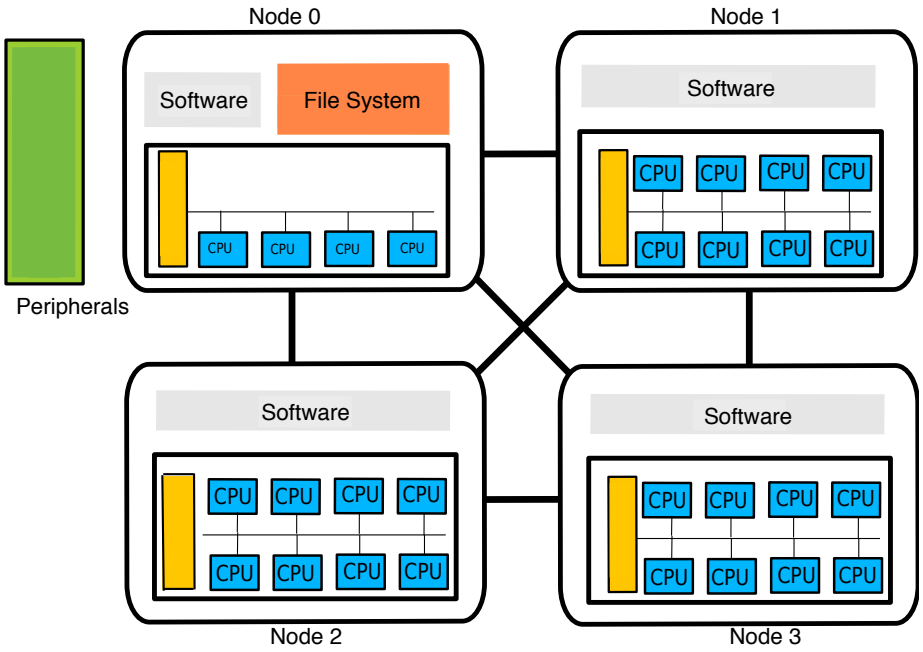


Figure 23: MPSoC Architecture

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 **??**). We have reengineered the code to redirect all Linux function calls to calls to our MPSoC API.

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 **??**. 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.

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

Figure 24: Video Mosaic Application

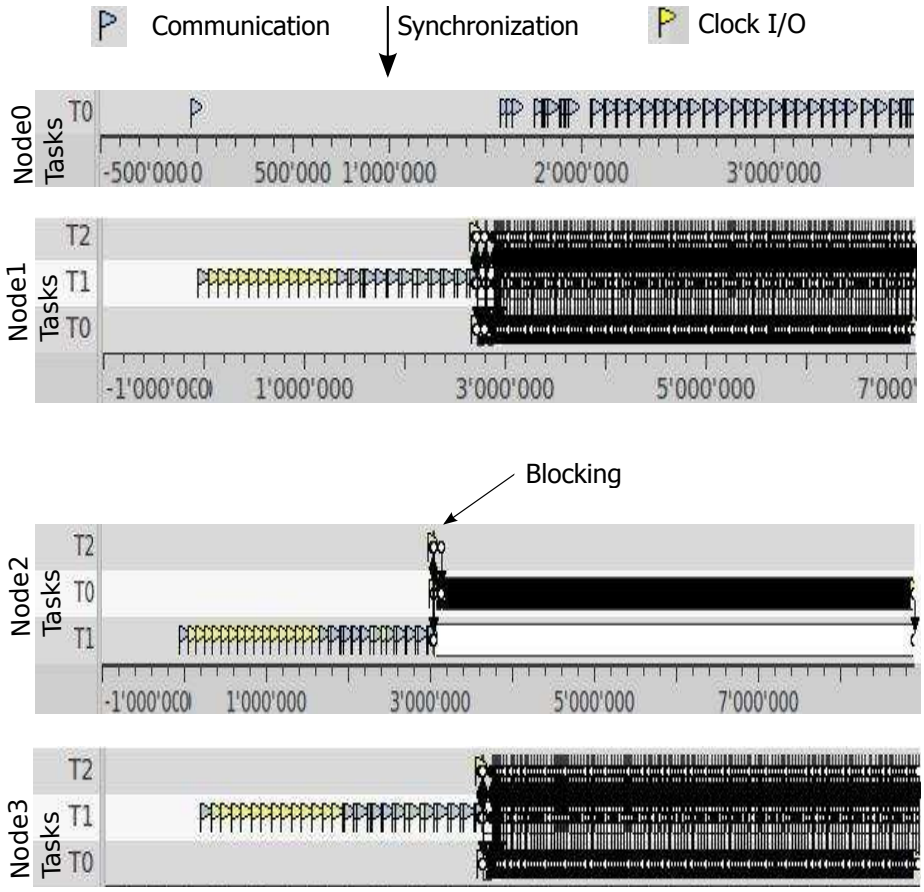


Figure 25: Visualization of Captured Traces.

Software	Native Time(s)	Reference Time(s)	Overhead(%)	Trace Volume(KB)	Number of 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

embedded and the NUMA platforms and have used the following measures:

- *Native 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.
- *Reference Execution Time*: 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.
- *Overhead*: 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 **??**.

In all use cases, the intrusion is very low 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 in 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.

A first observation is that, 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 to 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, another observation is 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, as we focus on a restrained type of events to record, in all cases the number of entries is rather small. In the MJPEG case, for example, due to the more intensive use of synchronization, the number of entries 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 example, 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 investigate to accelerate the process is to consider the creation of application snapshots allowing the deterministic replay to start in the middle of an application execution.

6 Conclusion and Perspectives

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".

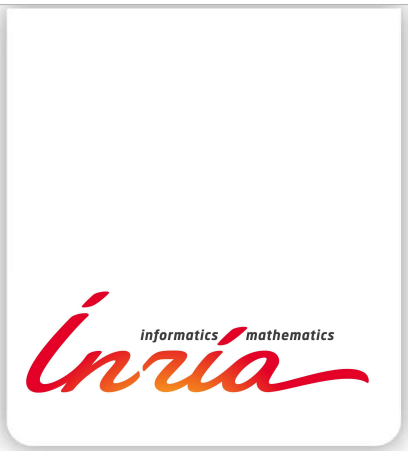
Intrusion is a major issue in record-replay systems and the usual answer is to provide ad-hoc solutions for minimizing the execution overhead. However, modeling and formally estimating the cost of a given tracing/replaying technique will allow for cost predictions and will greatly facilitate the choice 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 [**?**] 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.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

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

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399