Tool-set for NoC-based MPSoC Debugging – A Protocol View Perspective

Marcelo Ruaro[1], Everton A. Carara[2], Fernando G. Moraes[1]
[1] *PUCRS University, Computer Science Department, Porto Alegre, Brazil*
[2] *UFSM – DELC – Av. Roraima 1000 – Santa Maria – 97105-900 – Brazil*
marcelo.ruaro@acad.pucrs.br, carara@ufsm.br, fernando.moraes@pucrs.br

***Abstract*—Software development becomes an important issue in today's MPSoC design. Due to the inherent non-deterministic behavior of MPSoCs, they are prone to concurrency bugs. Debugging tools for MPSoC may be grouped in the following classes: simulators, parallel software development environments, NoC debuggers. An important gap is observed concerning a complete NoC-based MPSoC: tools to inspect the traffic exchanged between processing elements in a higher abstraction level, and not simply as raw data. This is the goal of the paper: propose a *new class* of debugging tools, able to trace the messages exchanged between PEs, enabling debugging at the protocol level. Examples of protocols include communication between tasks, mapping heuristics, monitoring schemes for QoS, among others. The paper presents the proposed debug framework, as well as a task migration protocol as case study.**

***Key-words*: MPSoC, NoC, debug environments, protocols.**

I. Introduction and Related Work

Challenges in MPSoC software development comprise [1]: (*i*) inter-processor communication; (*ii*) protocol stacks definition; (*iii*) OS porting and analysis; (*iv*) exploration of better programming models to simplify parallel programming; among others.

Due to the inherent non-deterministic behavior of MPSoCs, they are prone to concurrency bugs. During the software development, the support of debugging tools is mandatory to find such bugs, deadlocks and bottlenecks. Given a composition of subsystems running different portions of an application, communicating with each other in a variety of ways, using a number of different ISA processors and running tasks written in multiple programming languages using multiple programming models, the task of understanding the system behavior to track down the root cause of a problem, poses a big challenge to concurrent debugging tools [2].

Debugging tools for NoC-based MPSoCs may be grouped in the three classes. *Simulators* allow the use of waveforms, assertions, code coverage, etc. Such simulators enable accurate hardware verification at a cost of a high simulation time, besides offering limited support to software debugging. *Parallel software development environments* [3][4][5][6] have been proposed for the debug of applications, abstracting the hardware through virtual platform. Applications can be verified independently of the target hardware platform. *NoC debuggers* enable to evaluate the NoC performance from synthetic traffic generators or application traces [7]. The focus is on packet exchanging (raw data).

Murillo et al. [3] propose a debugger framework that detects software concurrency bugs dynamically. The Authors use an assertion language, named BPDLang. The framework can be configured to address different MPSoCs and low-level APIs. The adoption of an assertion language targets the debug of specific parts of the software, not being a general-purpose debug tool.

Murillo et al. [4] propose a component-based debugger architecture. The proposed debugger architecture is divided into components that communicate through well-defined interfaces, and an event-based intermediate representation. A set of high-level event monitors is used to collect events (debug data), ensuring scalability. A virtual platform was used to validate the proposal. The Authors show that the added infrastructure efficiently generated debug events, with an acceptable simulation time overhead (170%). However, details on how to use the events to debug the system are not presented.

Rojas [5] et al. propose a component-based approach to observe the MPSoC behavior. The method relies in 4 steps: instrumentation of the target system, raw data collection, treatment of the collected data, and data storage. The system instrumentation is done in C language, with the *EMBera* software library. The proposed approach resents small intrusiveness, increasing the memory use by 2.9%, and the average execution time by 1%.

Hedde and Pétrot [6] present a method to trace software events, enabling a fine-grain analysis of software running in MPSoCs. The method uses simulation together with instrumentation of component models. The SoCLib framework (virtual platform) is used in the experiments, with 1 to 16 PEs, focusing on data race detection. The Authors manually inserted software errors in the threads of a MJPEG decoder to validate the approach, resulting in correct data race detection.

Möller et al. [7] present a GUI to debug NoCs through monitors, named *Scopes*. Seven *scopes* are implemented, enabling to monitor the communication inside the NoC, the power consumption, the buffer usage, hotspot regions and the activity in input/output router ports. The work only focuses in the NoC traffic (raw data), does not addressing a real MPSoC traffic.

Due to the huge design space to explore, each class has specific goals: *simulators* target hardware development; *software environments* evaluate the correct execution of parallel applications, allowing the detection of concurrency bugs (e.g. deadlocks and race conditions). NoC debuggers explore NoC architectural parameters. However, an important *gap* is observed concerning a complete NoC-based MPSoC: tools to inspect the traffic exchanged between processing elements (PEs) in a higher abstraction level, and not simply as raw data. The meaning of data exchanged in a multiprocessed system is crucial to enable the events' sequencing during the debug process.

This is the *goal* of the paper: *propose a new class of debugging tools*, able to trace the messages exchanged between

PEs, enabling debugging at the protocol level. The implemented environment of such debug class acts as *Wireshark*, a free and open-source packet analyzer used for network analysis, and development of software and communication protocols.

The motivation to propose this new debug class is to enable fast evaluation of communication APIs, mapping heuristics, monitoring schemes, QoS services, NoC routing algorithms. In summary, make available to MPSoC designers a way to inspect the NoC traffic generated by the developed software at a higher level, keeping the confidence of the RTL simulation. The implemented environment presents the following features: (*i*) *generality*: it is possible to use it with different NoC-based MPSoCs; (*ii*) *non-intrusiveness*: the added code to trace packets does not interfere with the applications and network performance; (*iii*) *simulator independent*: the traffic trace may be obtained from any simulator; (*iv*) *configurability*: the environment may be easily configured for different MPSoC parameters.

The proposed debugging method relies in two steps: (*i*) generation of a compact traffic trace with all packets transmitted through the NoC (Section III); (*ii*) a set of tools to analyze the traffic trace (Section IV).

II. MPSoC Architecture

An MPSoC contains a set of PEs interconnected by a communication infrastructure. This work assumes NoCs as the communication infrastructure, due to its scalability and performance compared to buses. The following features, commonly found in MPSoCs [8], are adopted: (*i*) each PE contains at least one processor with a private memory; (*ii*) a 2D-mesh NoC topology; (*iii*) applications modeled as task graphs; (*iv*) communication model is message passing (MPI-like); (*v*) a multi-task operating system (OS) runs at each PE; (*vi*) a mapping function assigns tasks to PEs.

The NoC-based MPSoC is assumed modeled at the RTL-level (using e.g. SystemC, VHDL or Verilog), and should be simulated with a clock-cycle accuracy. Such requirement enables to accurately compute throughput, execution time of applications, and the traffic distribution in the NoC links.

The debug environment requires two files to describe an MPSoC (Figure 1), one with the platform configuration, and the other one with the set of available services. The latter is used to add meaning to the raw data traffic, enabling the debug at protocol level.

Figure 1(a) presents the platform configuration file. The *router_addressing* is related to how the routers are numbered, being possible to use a Hamiltonian sequence [9], or XY coordinates. The subsequent parameters configure the MPSoC dimensions, and if a clustering management is used [10] the cluster configuration. The clock period is used to compute the throughput. After the initial parameters, the set of tasks to be executed in the MPSoC are listed as a tuple [*task name, identifier*].

An MPSoC works by exchanging messages between PEs. Such messages correspond to specific protocols, such as communication between tasks (MPI-like behavior), or management functions, as task mapping or monitoring. Figure 1(b) presents the service description file, which contains all supported services by the MPSoC platform. The first lines of the file contain a tuple [*service name, identifier*]. The two last lines contain reserved words (starting with '$'), used to identify the services related to task mapping and end of task execution (bold in Figure 1(b)). With such information, it becomes possible to debug the traffic with the service name, instead its identifier stored in the packets.

router_addressing	hamiltonian
mpsoc_x	6
mpsoc_y	6
cluster_x	3
cluster_y	3
manager_position_x	0
manager_position_y	0
global_manager_cluster	0
clock_period_ns	10
BEGIN_task_name_relation	
...	
FIR	774
iquant	257
ivlc	258
idct	260
...	
END_task_name_relation	

(a) platform configuration file

MESSAGE_REQUEST	10
MESSAGE_DELIVERY	20
TASK_ALLOCATION	**40**
TASK_TERMINATED	**70**
TASK_DEALLOCATED	80
LOAN_PROCESSOR_RELEASE	90
DEBUG_MESSAGE	100
...	
...	
UPDATE_CS_CONTROL	280
QOS_REQUEST_SERVICE	290
MONITORING_PACKAGE	300
MIGRATION_CODE	320
MIGRATION_TCB_MEMORY	330
$TASK_ALLOCATION_SERVICE	**40**
$TASK_TERMINATED_SERVICE	**70**

(b) service description file

Figure 1 – Files used to describe the MPSoC platform.

Therefore, an important feature of the proposed environment is *generality*, being possible to use it with different MPSoCs types, by modifying the configuration files.

III. Packet Logger

A second important feature of the debugging environment is its *non-intrusiveness*. A packet logger, specific to each MPSoC implementation, is instantiated at each NoC router, reporting in a compact format the relevant data for all received packets. It acts like a network sniffer, keeping track of all packets entering the router. Each line of the generated log file contains the information of 1 packet, with the parameters:

$$\mathbf{tick}\,|\,\mathbf{router}_{cur}\,|\,\mathbf{service}\,|\,\mathbf{size}\,|\,\mathbf{bandwidth}\,|\,\mathbf{port}_{cur}\,|\,\mathbf{tgt}\,|\,\mathbf{task}_{ID}$$

where:

- *tick*: time in clock cycles related to the reception of the packet header in a router port;
- $router_{cur}$: current (local) router address;
- *service*: packet function (integer);
- *size*: packet size, in flits;
- *bandwidth*: number of clock cycles required to receive the packet, i.e., the amount of clock cycles the link is reserved for the packet;
- $port_{cur}$: port from which the packet entered the $router_{cur}$;
- *tgt*: address of router to which the packet must be sent;
- $task_{ID}$: optional parameter, corresponding to the task identifier number, required when the packet contains the task allocation or task finish services.

The *service* parameter corresponds to the packet function. In MPSoCs there are data and management packets. Data packets correspond to the messages exchanged between tasks at the application level. Management packets are treated at the OS level, and correspond to messages such as start task mapping, task migration, processor initialization, among other services.

A given *service* is part of a given *protocol*. For example, when a new application starts, there is a set of packets exchanged between a manager PE and PEs that will receive the application tasks.

The generated report makes possible to debug the MPSoC at the *protocol* level step by step. The protocol messages are

identified inside the traffic log file by means of the services description file - Figure 2(b). The simulation of a set of applications running simultaneously in an MPSoC requires millions of clock cycles, and dozens of signals to be monitored. Debugging with waveforms is unfeasible. With the proposed debugging environment, the designer can monitor the NoC traffic, inspecting for example only specific services corresponding to the protocol under development.

The result of the *packet logger* is a unique file, which is written by all *packet logger* instances. The *packet logger* is instantiated at each NoC router, corresponding to a SystemC module with less than 80 lines.

IV. Debug Environment

Figure 2 presents the simulation and debugging flow. The MPSoC simulation requires an RTL description, and object code of the OS and tasks. The result of the simulation can be verified with waveforms (unfeasible for large MPSoCs), or in an efficient way using the proposed environment. The debug environment requires 3 inputs: (*i*) the traffic log file created by the packet logger; (*ii*) the platform configuration file; (*iii*) the service description file. Simulators as ModelSim (Mentor) and NCLaunch (Cadence) generate the traffic log only at the end of the simulation. The SystemC simulator generates the log file during the simulation, enabling to use the debugging tools concurrently with the simulator. In this way, the debugging tools acts as a GUI for the SystemC simulator.

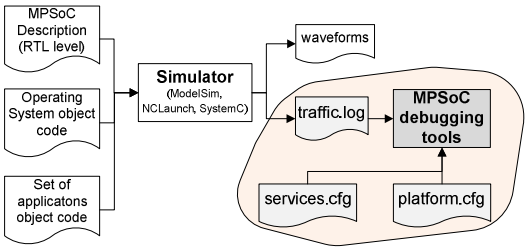

Figure 2 – Simulation and debugging flow.

The debug environment contains three main *views*: (*i*) throughput view; (*ii*) load view; (*iii*) mapping view.

A. Throughput View

Figure 3 illustrates the *throughput view* GUI, for a 4x4 MPSoC instance. The debugging environment assumes a 2D-mesh topology. Modifying the GUI can easily circumvent this restriction. The debug environment supports single and duplicated physical channels. There is no restriction regarding the number of virtual channels.

This view enables the designer to select the speed to display the packets, and supports breakpoints at specific services. Each packet read from the traffic log file highlights the link being used (red arrows from PE 0 to PE 13). When the packet reaches its destination, the source-target path returns to gray. Besides inspect packets from specific protocols with breakpoints, this view displays the throughput at each link, as highlighted in Figure 3. A parameterizable window interval counts the number of received flits at each input port (*bandwidth* field of the traffic log). A detailed report is created, with the throughput per link per time window.

Therefore, such view helps the designer to validate protocols, to measure the load induced by specific services/protocols, to identify hotspot regions, and to validate routing algorithms (the path taken by packets is highlighted).

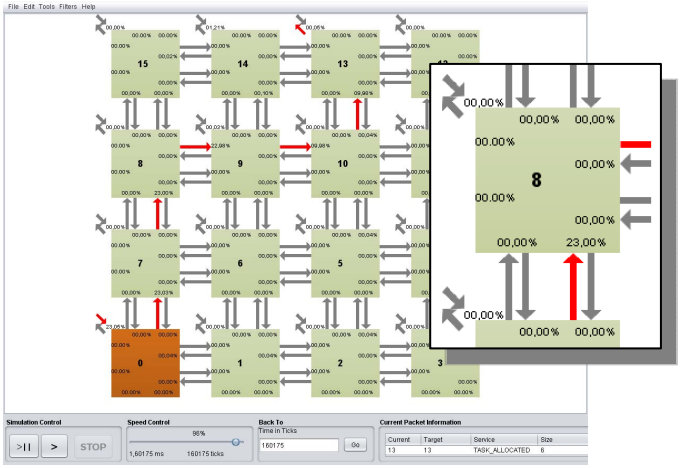

Figure 3 – *Throughput view* GUI, for a 4x4 MPSoC, with duplicated physical channels. PE0 is the manager PE. Router 8 is highlighted, with an average link use of 23% in one south port.

B. Load View

The goal of this view is twofold. First, it enables to evaluate *globally* the percentage of packets for a given service at each router. For example, it is expected that most packets related to task mapping have their source at the manager processor. Secondly, it enables the evaluation at the *router* level. Figure 4 illustrates the forwarding percentage of data packet at the router level. For example, 92.8% of packets traversing PE 14 are data packets. It is possible to inspect in the "throughput view" the percentage of such packets addressed to the processor (local port).

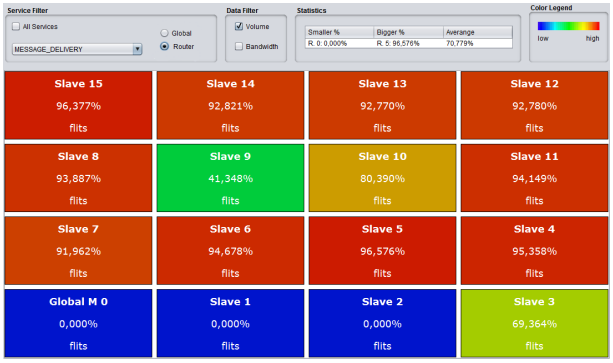

Figure 4 – Load view for the message delivery service (data packets), evaluating the percentage of received packets at each router.

Thus, this view enables to evaluate the impact of a given service in the traffic generated by the execution of the applications. For example, if the MPSoC designer is implementing a given monitoring protocol to ensure QoS, it is possible to measure the impact of such protocol in overall traffic.

C. Mapping View

This view enables the designer to monitor how applications are dynamically mapped in the MPSoC. Figure 5 illustrates this view. The DTW (dynamic time warping) application (in light green) uses PEs 10, 12 and 13. This view also shows the task status, i.e., running (RUN) or terminated (TER). Hence, this view is an important tool to evaluate mapping heuristics [11].

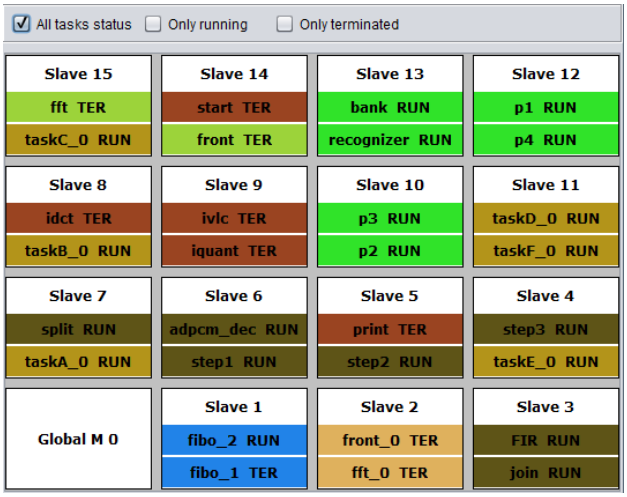

Figure 5 – Mapping view, for a scenario with 7 applications. The OS running in the slave PEs enable the execution of two simultaneous tasks. The Global PE manages the MPSoC.

V. Case Study

A case study is presented to show the use of the proposed debug environment. The protocol under evaluation is the task migration. The example corresponds to the migration of task t_i running in PE 8 to PE 4, in the HeMPS platform [12], simulated with clock-cycle accuracy. The manager PE sends a "*task migration request*" to PE 8, with the new t_i position. The OS running in PE 8 (OS_8) sends only the text area (instructions) to PE 4, without interrupting t_i – service "*migrate code*". When the OS_8 verifies that t_i may be effectively migrated, the data area is transferred to PE 4 – service "*migrate data*".

To debug this protocol, the user defines breakpoints for the 3 services used in the protocol. Once defined the breakpoints, the debug starts. When a given packet is read from the *log* file, the debug stops, and in main view (Figure 6 left) the user capture the time the packet was injected into the NoC.

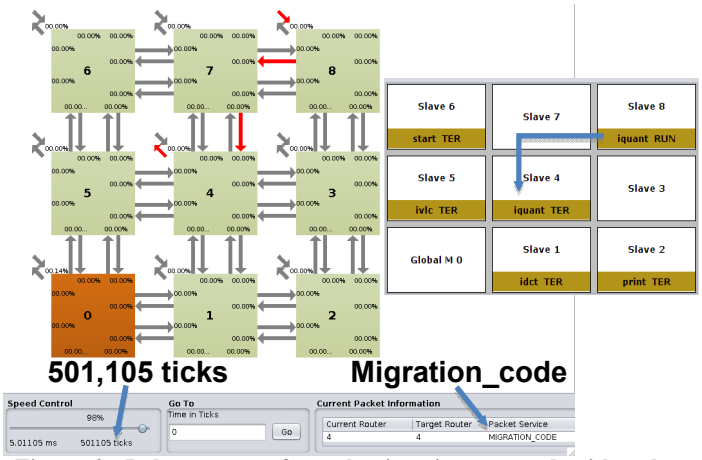

Figure 6 – Debug process for task migration protocol, with task iquant (MJPEG application) migrated from PE 8 to PE 4.

The delay of each service of the task migration, in clock cycles (*cc*), for the header flit, corresponds to:

- 1: *task migration request*: **500,084** to 500,104 *cc*, 6-flits packet;
- 2: *migrate code*: 501,095 to 501,105 *cc*, 758-flits packet;
- 3: 16,518 *cc* – time the task execute during the code migration;
- 4: *migrate data*: 517,613 to **517,623** *cc*, 1,066-flits packet.

Thus, the total time to execute the task migration is 17,539 *cc* plus 1,066 *cc* (corresponding to the last packet size). It is also possible to debug the path taken by packets, the links usage percentage during migration, and to inspect the task mapping before and after migration (Figure 6 right), with t_i corresponding to task *iquant*.

It is important to mention that the protocol evaluation was carried out with no software instrumentalization, as most works in the state-of-the-art require.

VI. Conclusions

The presented environment abstracts the MPSoC ISA processors, i.e., to debug a given MPSoC instance it is not necessary to known which processor architecture is being used, neither software details. In addition, it is compliant with the features required by a debugger framework: generality, non-intrusiveness, configurability (supports MPSoCs of any dimensions). The current implementation adopts a clock-cycle accuracy providing correspondence with waveform simulation. It is possible to use abstract MPSoCs models (untimed or TLM), and generate a sequence of events. The main loss using abstract MPSoC models is the throughput accuracy.

The proposed open-source environment does not replace existing debug tools. As shown, the "views" of the debug environment help designers to evaluate the NoC traffic according to the traffic generated by the software running on real MPSoC PEs and executing real applications. Such debug level is the *original* contribution of the paper.

Works in the path to extend the debug environment include: (*i*) a latency and power views; (*ii*) protocol coherence verification; (*iii*) CPU logger, enabling to capture OS events.

Acknowledgment

The author Fernando Moraes acknowledges the support granted by CNPQ, processes 472126/2013-0 and 302625/2012-7; and CAPES process 708/11.

References

[1] Gray, I.; Audsley, N. ***Challenges in Software Development for Multicore System-on-Chip Development***. In: RSP, 2012, pp. 115-121.

[2] Martin, G.; ***Overview of the MPSoC Design Challenge***. In: DAC, 2006, 6p.

[3] Murillo, L.; et al. ***Debugging Concurrent MPSoC Software with Bug Pattern Descriptions***. In: S4D, 2011, 4p.

[4] Murillo, L.; et al. ***Scalable and retargetable debugger architecture for heterogeneous MPSoCs***. In: S4D, 2012, 6p.

[5] Prada-Rojas, C.; et al. ***A Generic Component-Based Approach to MPSoC Observation***. In: EUC, 2011, pp. 261-267.

[6] Hedde, D.; Petrot, F. ***A non intrusive simulation-based trace system to analyse Multiprocessor Systems-on-Chip software***. In: RSP, 2011, 6p.

[7] Mo□ller, L.; et al. ***Graphical interface for debugging RTL Networks-on-Chip***. In: BEC, 2010, pp. 181-184.

[8] Garibotti, R., et al. ***Simultaneous Multithreading Support in Embedded Distributed Memory MPSoCs***. In: DAC, 2013, 6p.

[9] Lin, X.; McKinley, P. K.; Ni, L. M. ***Deadlock-free Multicast Wormhole Routing in 2-D Mesh Multicomputers***. IEEE Trans. on Parallel and Distributed Systems, v.5(8), 1994, pp. 793-804.

[10] Castilhos, G.; Mandelli, M.; Madalozzo, G.; Moraes, F.G. ***Distributed Resource Management in NoC-Based MPSoCs with Dynamic Cluster Sizes.*** In: ISVLSI, 2013, pp. 153-158..

[11] Singh, A.; et al. ***Mapping on Multi/Many-Core Systems: Survey of Current and Emerging Trends***. In: DAC, 2013, 10p.

[12] Carara, E.A.; de Oliveira, R.; Calazans, N.; Moraes, F.G. ***HeMPS - a framework for NoC-based MPSoC generation***. In: ISCAS, pp. 1345-1348, 2009.