A Data Extraction and Debugging Framework for Large-Scale MPSoCs

Marcelo Ruaro, Henrique Chamorra, Felipe Rubin, Alexandre Amory, Fernando G. Moraes
PUCRS University, Computer Science Department, Porto Alegre, Brazil
{marcelo.ruaro, henrique.chamorra, felipe.rubin}@acad.pucrs.br, {alexandre.amory, fernando.moraes}@pucrs.br

Abstract—**Although it is possible to design and manufacture MPSoCs with hundreds of processors, there is still a gap in the ability to debug hardware, software, and applications for such chips. Current state-of-the-art works related to MPSoC debugging suffer from poor integration, scalability in data storage, and simple graphical data representation. This work proposes a modular debugging framework to aid the software development for large-scale MPSoCs. The framework contains two parts: back-end and front-end. The back-end contains a generic model for extraction and storage of communication and computation data. The front-end comprises a graphical debugging toolset, which provides intuitive debugging tools for the communication and computation data events. A key point of the proposal is to be generic and applicable to different MPSoC architectures due to the decoupling of the back-end and the front-end components of the framework. A data extraction layer abstracts the MPSoC architecture by defining a standard structure to database insertion. The framework is validated in an MPSoC platform described in SystemC RTL (clock-cycle accurate), addressing MPSoC sizes of up to 400 processors.**

Keywords— MPSoC; Many-Core; CAD tools; Debugging Tools; Frameworks

I. INTRODUCTION

Powerful CAD tools aid the VLSI design of MPSoCs, reporting low-level design estimations. The drawback of these simulators includes considerable simulation time and limited software debugging. High-level debugger abstracts low levels details, enabling to concentrate the effort to validate system behaviors as parallel software execution or communication protocols.

This paper presents a modular verification framework where it is possible to debug high-level computation and communication events in large-scale MPSoCs concurrently. The *first contribution* is a generic data extraction layer, named DEL, which collects computation and communication events from the simulated MPSoC and stores such data into a database. The process of data extraction can be integrated with RTL, TLM, or virtual platforms descriptions. Such flexibility comes from an abstraction of the target MPSoC architecture implemented by DEL, which defines a generic data extraction method combined with a standard database insertion. The *second contribution* is a graphical debugging toolset which explores the debugging database generated by DEL and creates several GUI used for high-level communication and computation debugging. The data extraction (*back-end*) is decoupled from the graphical debugging tools (*front-end*), enabling the development of other custom front-end debuggers (graphical or not).

The *originality* of this proposal is a generic and scalable approach for data collection combined with a graphical toolset for debugging. This proposal can be used jointly with state-of-the-art debugging methods acting as a complementary debugging approach.

II. RELATED WORKS

Debugging methods for MPSoCs and NoCs have gained increased attention due to the increased processor density, which makes the debugging process more complex. Table I presents recent works related to NoC and MPSoC debugging.

TABLE I – RELATED WORKS RELATED TO NOC AND MPSOC DEBUGGING.

Work	Data Extraction	Target Debugging	MPSoC description	Scalability concern	DB	GUI
Mur. [1] 2014	event inside cores	Parallel Software Concurrency	Virtual Platform	No, 4 cores	No	No
Geo. [2] 2014	software API	Software Errors	Virtual Platform	Yes, 32 cores	No	Yes
Wen. [3] 2012	core events	Parallel software data race	Virtual Platform	Yes, 64 cores	No	No
Cue. [4] 2012	breakpoints	Multimedia App. periodic conflicts	RTL	No, 3 cores	N/A	Yes
Roj. [5] 2011	SW instrumentation	Generic Observation	RTL	No, 3 cores	No	No
Hed. [6] 2011	virtual HW events	Parallel software data race	Virtual Platform	Yes, 16 cores	No	No
Nei. [7] 2012	NI	NoC transactions and data race	RTL	No	No	No
Fri. [8] 2014	at runtime by a host unit	Visualize core logs	RTL (FPGA)	Yes, 45 cores	No	No
Alh. [9] 2010	router links	NoC link usage statistics	RTL (FPGA)	Yes, 16 cores	No	Yes
Mol. [10] 2010	router links	NoC usage statistics	RTL	No, 9 cores	No	Yes
This work	**router links, CPU events**	**Communication / Computation**	**RTL and Virtual**	**Yes, 400 cores**	**Yes**	**Yes**

Data extraction (2nd column of Table I) addresses how the debugging method collects the data to be used in the debugging process. Some works extract data from the cores, enabling the debug of parallel applications [1]-[6]. Others works apply a communication-based data extraction by extracting data from the NoC links [9][10] or the NI (Network Interface) [7] to debug the communication events. The proposed work extracts the data from the router's links covering communication debug, and extracts CPU events to cover computation debug. Such approach provides a broad view of the system resources. Non-intrusive monitors extract these computation and communication events.

The *target of debugging* (3rd column) corresponds to the focus of the debugging. Three main debugging methods are identified in the literature: (*i*) parallel software debuggers; (*ii*) NoC debuggers; (*iii*) FPGA emulation. Most of the works adopt parallel software debug. Other works focus on improving the debug over FPGA implementations [8][9], and the works [9][10] only debug the NoC structure (NoC debuggers). The proposed method mixes the debug of computation and communication resources.

The MPSoC description (4th column), is related to abstraction level adopted to model the MPSoC. Authors [1][2][3][6] adopt virtual-based platform descriptions. Such choice enables to speed up the simulation time to direct all debugging efforts to improve the software development. Other platforms are designed at the RTL level, with some proposals including FPGA emulation [8][9]. As the proposed framework adopts a generic data extraction, the designer implements the DEL according to the platform model. Therefore, both HDL models or virtual models are supported.

The scalability column (5th column) evaluates whether the debugging methods can handle large-scale data sets. The proposed framework leverages scalability by adopting a database (DB – 6th column of Table I) to provide an efficient solution (described in the Results section) to access structured data with SQL queries.

The graphical representation (GUI) of the system events (7th column) eases and reduces the time spent debugging. Cueva et al. [4] propose a simple GUI to observe the periodicity conflict between applications. Alhonen et al. [9] and Moller et al. [10] propose graphical tools to generate NoC statistic. All the aforementioned works adopt simple graphical interfaces. The proposed framework presents several GUIs allowing a fast interpretation of the monitored system events.

III. DATA EXTRACTION MODEL

Figure 1 details the proposed *data extraction model*. It contains the Data Extraction Layer (DEL) and the database. The upper part of Figure 1 corresponds to a standard simulation environment. It assumes a generic NoC-based MPSoC, with a set of PEs interconnected to a NoC. The software part contains an OS and a set of applications to execute at each PE of the system.

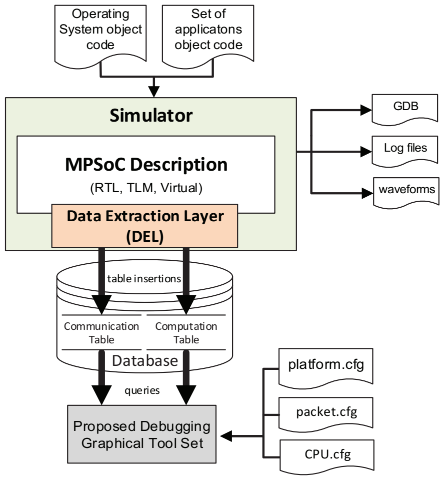


Figure 1– Overview of the proposed debugging data extraction method.

A simulator receives the hardware and software parts of the system, simulating it using some hardware description abstraction. A set of analysis may be executed according to the adopted simulator, as waveforms, log files, and GDB. This work follows a different approach by proposing a generic DEL to extract communication and computation events and save them into a database following a standard structure (subsection 3.1). The front-end debugging tools communicate with the database, according to a management protocol detailed in subsection 3.2.

A. DEL – Data Extraction Layer

The DEL collects data generated by the simulation and inserts them into database tables (e.g. latency, throughput, selected functions). *Communication events* are represented by packets arriving in the NoC router's input port. *Computation events* are specific software addresses executed into CPU. Such events enable to achieve a holistic view of the platform functional behavior. The DEL can be seen as the interface between the MPSoC platform and a front-end debugger (graphical or not), decoupling both parts of the system.

To achieve non-intrusiveness over the application execution, the DEL is implemented as a part of the hardware platform. This part is used only during the simulation at design-time, and it is removed for synthesis or prototyping. The DEL operation comprises two parts: *monitoring* and *database insertion*. The monitoring is in charge for extract the system data. For communication data, sniffers are implemented inside the router instance monitoring all input ports. The computation data extraction is implemented in a similar fashion, sniffing the CPU instruction address.

The database insertions are done directly from the hardware in the same process of the monitoring. The DEL varies according to description level of the target platform. An RTL SystemC simulation enables database insertions directly from the SystemC code. A VHDL simulation can use a SystemC wrapper to execute the same task. The implementation of the DEL is a responsibility of the designer, once that it knows the platform details.

The data created by the DEL must follow a standard format to enable the insertion of the monitored events into the debugging database. The database implements two different tables. One for storing communication and the other for computation events.

B. Database Management

The database is modeled in SQL. The communication and computation tables are created at the beginning of a new simulation. Figure 2 shows the sequence diagram for a new debugging scenario detailing the database management protocol necessary to the interaction between the Simulator, DEL, database, and a generic front-end debugger.

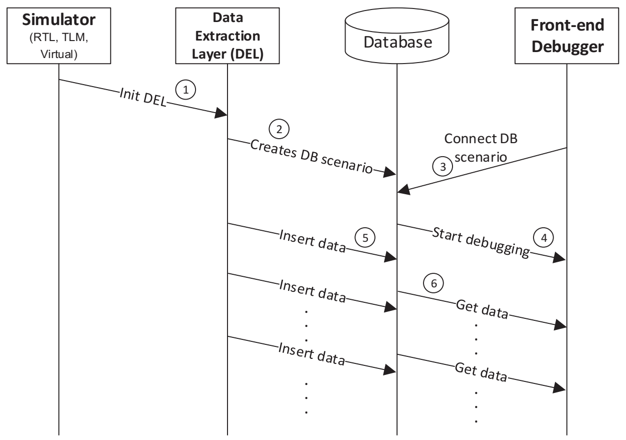


Figure 2 – Sequence diagram for database management.

At step *1*, the simulator initializes the DEL; at step *2*, DEL connects to the database server and creates a new database scenario for that simulation. A backup of previous databases is also created at this step. At step *3*, the front-end debugger can start a new debugging session, establishing a new connection with the database. The database connection of the DEL, at step *2*, and the connection of the debugger, at step *3*, are performed using a 5-tuple {*hostname, remote_port, scenario_name, user, password*}. After the connection of the debugger with the database in step *3*, the debugging can start, in step *4*. The DEL extracts the platform communication and computation events and inserts into the database (step *5*). The front-end debugger can read such information, at step *6*, by requesting information from the database using SQL queries.

IV. Graphical Debugging Toolset

The proposed graphical debugging toolset (referenced from this point as *debugger*) works by reading communication and computation events stored in the database and converting such data into graphical information that can be used by the system designer to debug and to improve the design of MPSoC hardware and software components (OS and applications).

The focus of the debugger is to provide high-level awareness of the system status to the designer, enabling the analysis of computation and communication events. Figure 1 details the debugger inputs. The debugger receives configuration files at the initialization and performs queries over the communication and computation database tables.

A. Configuration Files

The debugger uses three configuration files: *platform*, *services*, *cpu*. These files can be generated automatically during the system compilation phase.

The *platform* file configures the debugger by providing parameters about the MPSoC architecture (e.g. size of the system) and the application task set. The set of tasks to be executed in the MPSoC are listed as a tuple {*task name, identifier*}.

The *service* file contains the services supported by the platform. A *service* identifies the function of a given packet, and a *protocol* is defined by a set of *services*. For example, the message exchange protocol requires two *services*: MESSAGE_REQUEST and MESSAGE_DELIVERY. Each line of the *service* file contains a tuple {*service name, identifier*}.

The *cpu* file describes the CPU addresses to be monitored. These addresses are monitored by the DEL, which generates computation events to informs that the addresses were executed. Examples of observation points are the addresses of the scheduler, system calls, task execution, interruption handler.

B. Main View

The main view shows an overview of the MPSoC architecture. By using this view, the system designer can debug the *communication* behavior as NoC routing and link utilization to validate system management protocols or task communication messages, and supervise parallel communications.

Figure 3 illustrates the view for a 4x4 MPSoC with 2x2 clusters. The green PEs represent cluster managers, the orange PE represent a global manager, the blue PEs execute user applications. Each PE of this view contains the input channel utilization of each router port. Those values represent the percentage of the channel bandwidth usage, computed for a fixed time window (parameterizable in the tool).

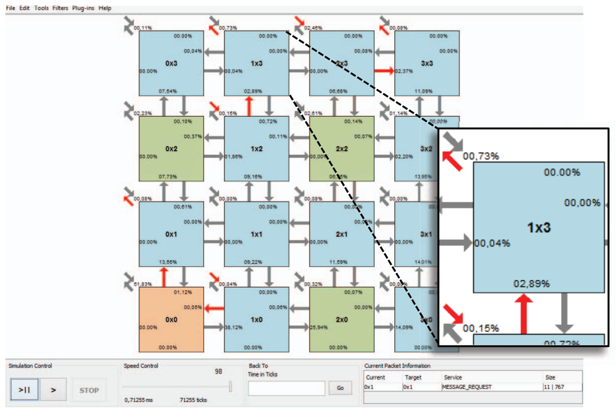
The zoom in PE 1x3 (Figure 3) details the channel utilization. It is possible to observe that the south port of the PE router has a channel utilization equal to 2.89% on this particular time window.

Each packet traveling into the NoC is displayed with a red arrow according to the packet advances to the next PE. The packet traces are colored in red. Figure 3 shows three packets traveling in the NoC at the time 71,225 (observed into *Speed Control* panel): 1x0 to 0x1, 1x2 to 1x3, and 2x3 to 3x3.

C. Mapping View

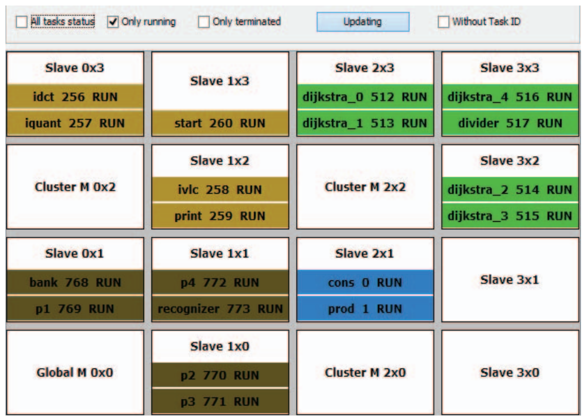
Figure 4 presents the mapping view addressing *computation* debugging. With this view, the designer can to validate task mapping algorithms, view the occupation of the PEs, and observe the task execution status.

Each application contains a set of communicating tasks. Tasks belonging to the same application have the same color. Each task is displayed according to its ID detailed in the *platform* file. The designer can choose to see all task status, only the running tasks, or only the terminated tasks (tasks that already finished its execution). Tasks are displayed dynamically as they are mapped.



Current	Target	Service	Size
0x1	0x1	MESSAGE_REQUEST	11 767

Figure 3 – Throughput and Communication Event View.



☐ All tasks status ☑ Only running ☐ Only terminated Updating ☐ Without Task ID

Slave 0x3 idct 256 RUN iquant 257 RUN	Slave 1x3 start 260 RUN	Slave 2x3 dijkstra_0 512 RUN dijkstra_1 513 RUN	Slave 3x3 dijkstra_4 516 RUN divider 517 RUN
Cluster M 0x2	Slave 1x2 ivlc 258 RUN print 259 RUN	Cluster M 2x2	Slave 3x2 dijkstra_2 514 RUN dijkstra_3 515 RUN
Slave 0x1 bank 768 RUN p1 769 RUN	Slave 1x1 p4 772 RUN recognizer 773 RUN	Slave 2x1 cons 0 RUN prod 1 RUN	Slave 3x1
Global M 0x0	Slave 1x0 p2 770 RUN p3 771 RUN	Cluster M 2x0	Slave 3x0

Figure 4 – Mapping view for a scenario with 4 applications, each one represented by a different color.

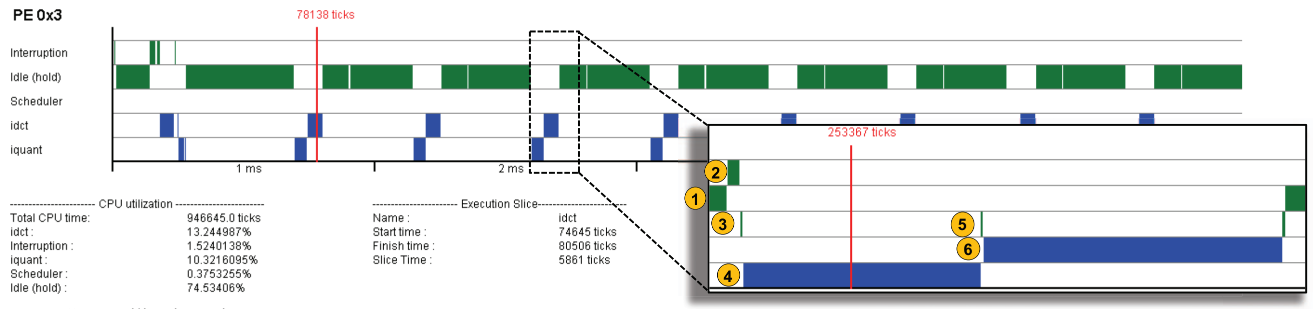
D. CPU Utilization View

The CPU utilization view addresses *computation* debugging. It enables the designer to verify the CPU use by different software parts over the time. The logged CPU events are addresses related to OS functions and the execution of a task. The addresses are described in the *cpu* file.

Figure 5 shows an example of the CPU utilization view for the PE 0x3. The y-axis corresponds to the monitored software events, and the x-axis the simulation time. This view also shows at the bottom left corner CPU utilization statistics, as the total CPU simulation time, and the percentage of CPU utilization for each OS functions and each task executed by the CPU (in this example *idtc* and *iquant*). With this view, the designer can validate scheduling algorithms by verifying if a given task meets its constraints, evaluate the processing load to validate task mapping algorithms, correlate the processor events with communication protocols. Additionally, this view enables to debug OS or software bugs.

V. Results

This Section presents results using an MPSoC platform modeled in SystemC RTL, with clock-cycle accuracy [11]. Applications (as DTW, MPEG, Dijkstra) and OS are described in C language.



```
---------------------- CPU utilization ----------------------
Total CPU time:          946645.0 ticks
idct :                   13.244987%
Interruption :           1.5240138%
iquant :                 10.3216095%
Scheduler :              0.3753255%
Idle (hold) :            74.53406%

---------------------- Execution Slice----------------------
Name :                   idct
Start time :             74645 ticks
Finish time :            80506 ticks
Slice Time :             5861 ticks
```

Figure 5 – CPU Utilization View.

Figure 6 presents the debugging of a large-scale MPSoC – 256 PEs, organized in a 16x16 mesh, with 4x4 clusters. The example is a TASK_ALLOCATION service, correlating the debugging of computation and communication events. The main view enables the user to observe a TASK_ALLOCATION packet leaving PE 0x0 (label 1 in Figure 6) and arriving at PE 6x4 (2) at time 429,138 ticks (clock cycles). This packet carries a task object code. When the target PE (6x4) receives the packet, the mapping view shows the task *p3* allocated at this PE (3). The CPU view enables to observe that the OS of the target PE was in the idle state. At time 429,252 the OS executes the interruption handler (4), to consume the packet received at the local port of the router, with the task code (*p3*). Next, the scheduler selects the task *p3* and starts its execution (5).

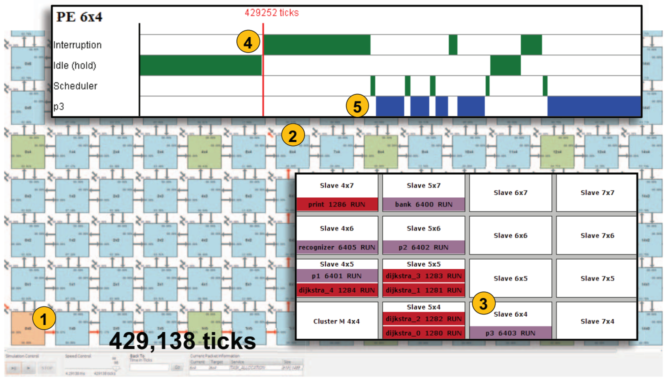


Figure 6– Case-study debugging an MPSoC with 256 PEs.

Table II summarizes results related to simulation time and data storage, obtained from a SystemC RTL simulation, comparing the use of the database with an approach using log files (events stored in text files). The simulation time is penalized in 15% due to the operations to manage the database. On the other side, the required space to store the events reduces 26%. Executing an application in a 20x20 MPSoC for 10 seconds would require a data storage space of ~7 GB using log files, and ~5 GB using the database approach.

TABLE II – RESULTS RELATED TO SIMULATION TIME AND DATA STORAGE, FOR 100 MS OF SIMULATION (DB: DATABASE).

System size	Simulation time (sec)			Data Storage (MB)		
	DB	log files	DB/log	DB	log files	DB/log
8x8	3,131	2,740	1.14	9.41	12.39	0.76
10x10	15,546	8,446	1.84	13.34	17.55	0.76
12x12	11,299	8,261	1.37	18.41	24.60	0.75
14x14	9,354	11,470	0.82	25.13	33.98	0.74
16x16	19,660	17,880	1.10	31.96	43.71	0.73
18x18	16,993	17,146	0.99	41.78	57.29	0.73
20x20	26,552	25,886	1.03	50.87	69.98	0.73

These results advance two key points related to debugging large-scale MPSoCs. The first one is the *graphical representation* of events generated during the simulation. It requires intuitive tools for accelerate debugging. The proposed framework provides fundamentals GUI that enable the designer to verify the system operation quickly. The second point is *scalability*, once large-scale MPSoCs request flexible debugging approaches, as databases to handle large data sets.

VI. CONCLUSION AND FUTURE WORKS

This work proposes a generic data extraction model combined with a graphical debugging toolset based on debugging data stored in a database. The presented results, with MPSoC sizes up to 400 PEs, show a flexible and scalable data extraction model that can be easily integrated into an MPSoC platform. Additionally, this work proposed a graphical debugging toolset that provides an intuitive debugging environment. By combining these two contributions, this work showed a flexible debugging framework that fits with modern large-scale MPSoCs. The proposed framework is open source, and it may be obtained by contacting the Authors. Extensions to this work include the integration of energy estimation, and a real-time deadline debugger to graphically observe deadlines for each real-time application.

ACKNOWLEDGMENTS

The Author Fernando Moraes is supported by CNPq - projects 472126/2013-0 and 302625/2012-7, and FAPERGS - project 2242- 2551/14-8.

REFERENCES

[1] Murillo, L. G.; et al. *Automatic detection of concurrency bugs through event ordering constraints*. In: DATE, 2014.
[2] Georgiev, K.; Martin, V. M. *MPSoC Zoom Debugging: A Deterministic Record-Partial Replay Approach*. In: EUC, 2014, pp. 73-80.
[3] Wen, C. N.; et al. *NUDA: A Non-Uniform Debugging Architecture and Nonintrusive Race Detection for Many-Core Systems*. IEEE Transactions on Computers, 2012 vol. 61(2), pp. 199-212.
[4] Cueva, P. L.; et al. *Debugging embedded multimedia application traces through periodic pattern mining*. In: EMSOFT, 2012, pp. 13-22.
[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 analyze Multiprocessor Systems-on-Chip software*. In: RSP, 2011.
[7] Neishaburi, M. H.; Zilic, Z. *An enhanced debug-aware network interface for Network-on-Chip*. In: ISQED, 2012, pp. 709-716.
[8] Friederich, S.;Heisswolf, J; Becker, J. *Hardware/software debugging of large scale many-core architectures*. In: SBCCI, 2014, pp. 1-7.
[9] Alhonen A.; et al. *A scalable, non-interfering, synthesizable Network-on-chip monitor*. In: NORCHIP, 2010.
[10] Möller, L.; et al. *Graphical interface for debugging RTL Networks-on-Chip*. In: BEC, 2010, pp. 181-184.
[11] Carara, E.; Oliveira, R.; Calazans, N.; Moraes, F. "*HeMPS - a Framework for NoC-based MPSoC Generation*". In: ISCAS, 2009.