ManyGUI: A Graphical Tool to Accelerate Many-core Debugging Through Communication, Memory, and Energy Profiling

Marcelo Ruaro, Kevin J. M. Martin

ManyGUI: A Graphical Tool to Accelerate Many-core Debugging Through Communication, Memory, and Energy Profiling

Marcelo Ruaro
marcelo.ruaro@univ-ubs.fr
Univ. Bretagne-Sud, Lab-STICC, UMR CNRS 6285
Lorient, France

Kevin J. M. Martin
kevin.martin@univ-ubs.fr
Univ. Bretagne-Sud, Lab-STICC, UMR CNRS 6285
Lorient, France

ABSTRACT

The debugging and validation of the many-core design is a complex task due to numerous events happening in the system simultaneously. Current state-of-the-art many-cores are strongly based on waveforms and log files to validate their behavior during simulation. Our hypothesis is that, as happened with ASIC development, a Graphical User Interface (GUI) can significantly accelerate the many-core development. To sustain that, we propose an open-source GUI tool called *ManyGUI* for many-core debugging. *ManyGUI* is organized in a framework that collects and classifies high-level events during simulation related to computation (executed CPU instructions), memory, and communication (NoC packets). Such events are shown graphically to the developer through a set of intuitive and practical frames. We evaluate *ManyGUI* in a silicon-proven state-of-the-art open-source many-core called OpenPiton, which uses RISC-V 64-bits CPU, 3 NoCs, and a distributed/shared cache memory organization. Results show that *ManyGUI* allows the developer to rapidly obtain a comprehensive view of the many-core behavior in terms of communication statistics (packets paths, link utilization), memory statistics (memory access, miss rate), and energy (CPU, memory, and NoC).

CCS CONCEPTS

• **Hardware → Simulation and emulation**; • **Computer systems organization → Multicore architectures**.

KEYWORDS

Many-core, Validation, Graphical, Energy, Memory, Network-on-Chip

Author version

This document is the author version of the paper "ManyGUI: A Graphical Tool to Accelerate Many-core Debugging Through Communication, Memory, and Energy Profiling" by *Marcelo Ruaro, Kevin J. M. Martin*, accepted for publication in RAPIDO'22.

ACM Reference Format:
Marcelo Ruaro and Kevin J. M. Martin. 2022. ManyGUI: A Graphical Tool to Accelerate Many-core Debugging Through Communication, Memory, and Energy Profiling. In *System Engineering for constrained embedded systems (DroneSE and RAPIDO '22), June 21, 2022, Budapest, Hungary.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3522784.3522791

1 INTRODUCTION

Many-cores are systems-on-chip that reach outstanding processing power by assuming tiles implemented as a processor, memory, or a specific accelerator [4]. The tiles are connected through a scalable on-chip technology called Network-on-Chips (NoCs) [11]. Many-cores quickly evolve from research prototypes to mature designs used in industry today[1]. The next generation of many-core, powered by the advances in semiconductor technologies and novel power management techniques, is expected to reach hundreds either thousands of tiles. While NoCs facilitate the placement of new tiles on the chip, the complexity of such systems due to the abundance of components, events, protocols, among others, pose significant challenges to debugging them during the design phase. Specifically, there is an important lack of tools that allow the designer to rapidly understand what is happening in the system regarding communication events (packet level), memory statistics, and energy consumption.

A study of the state-of-the-art of current many-core platforms leads to the observation that they are heavily based on log files and waveforms. We see that such methods can be significantly extended by using Graphical User Interfaces (GUI). The introduction of GUI in the ASIC design allowed a rapid and intuitive chip development. As happened on ASIC design, we support that GUI advantages can also be exploited to accelerate many-core functional development, assuming a higher level observation strategy that aims to reduce the time lost analyzing big waveforms and understanding large log files. Figure 1 position our idea. It shows functional development steps of many-cores typically found in the literature [8, 11, 13, 15]. The figure can be observed from the bottom to the top. At first, the developers work in designing the hardware model, for instance, using RLT-based languages like VHDL and Verilog. The software design comprises the Operating System (OS) and application development.

After having the software and hardware design, the developers team usually creates scripts that automatically resize the model before compilation according to design-time features, such as the number of cores, memory sizes, and NoC flit width. The scripts generate essential software and hardware files, as "include files," which will drive the automatic generation of hardware and predefined pragmas in software. Once the final design was generated, the flow advances to the simulation phases, using commercial or open-source simulator tools, which compile the hardware in a given description level and simulate its behavior based on the software.

The next step is the debugging and validation, where the developer will analyze if everything is working as expected at a logical level[2]. Traditionally, simulators provide waveforms, which allow a very detailed (gate-level) representation of the system. The design can also print some values at important checkpoints, which are displayed in log files. If the hardware is not modeled at the RTL

[1] www.kalrayinc.com/products/mppa-technology
[2] the physical validation is out of the scope of this work

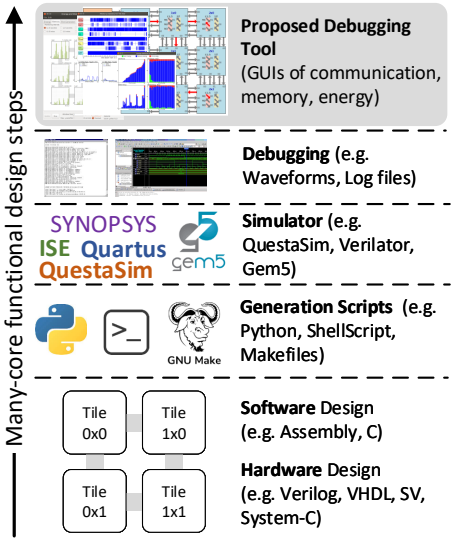


Figure 1: Overview of many-core functional design steps. This work proposes a high-level GUI-based debugging step, extending waveforms and log files.

level (as in the Gem5 simulator [13]), the simulator can only output log files.

The motivation for this work is that current debugging methods based on waveform and log files are slow and prone to interpretation errors due to the fatigue passed to the developer to follow the massive information volume that many-core generates while simulating. The goal of this work is to go one step higher in the debugging and validation process by proposing a GUI tool called *ManyGUI* that, based on high-level events collected at computation, memory, and communication level, can depict the communication flow, memory statistics, and energy consumption. *ManyGUI* allows a rapid and comprehensive understanding of the many-core behavior and, consequently, the application running on it. This work has the following contributions:

(1) An overview of recent many-cores platforms and how they implement the debugging during simulations;
(2) A framework to collect and classify events of computation (executed CPU instructions), memory, and communication (NoC packets) during the many-core simulation, assuming a generic interface which can be easily adapted to different many-core designs;
(3) A set of graphical frames which show, during the simulation, communication statistics (packets paths, link utilization), memory statistics (memory access, miss rate), and energy (CPU, memory, and NoC);
(4) A case study of *ManyGUI* in a state-of-the-art open-source many-core called OpenPiton [2].

2 OVERVIEW OF MANY-CORE PLATFORMS AND DEBUGGING

This section presents contribution (1). Table 1 addresses the main characteristics of the state-of-the-art many-core design platforms in literature. The 1st column shows the name of the platform, its reference work, and the year. Focusing on an updated classification, we consider only recent works, dating not more than 5 years ago (from 2017 on). Older platforms classifications can be found in [1, 7, 14]. The 2nd column addresses the language used to model the platforms. It is possible to see that most of the listed platforms use Verilog or System-Verilog (SV) while very few rely on VHDL and System-C. One of the advantages of using Verilog and SV is its C-like style and its easy integration with open-source simulators as Verilator[3], which promises to speed-up the simulation by converting the model to multithreaded C++. The 3rd column classifies works by detailing how the computation is implemented. Noticeably, RISC-V was received considerable attention from research, resulting in the majority of the works, which either adopt RISC-V cores exclusively [2, 6, 7] or use it in an on-chip heterogeneous fashion [4, 8, 15]. Another observed aspect is that many-cores are increasingly adopting accelerators to reach energy efficiency in complex applications, specifically to support machine learning [4, 12, 13, 15].

The 4th column classifies works by their memory organization. Few architectures adopt purely distributed system [6, 11, 14]. Most systems use a hybrid shared memory system, which includes a private memory (L1 or scratchpad) with a logically shared but physically distributed last level shared memory [1, 2, 7, 12, 15]. Such hybrid systems are suitable to many-core since a centralized shared memory is not scalable for a high number of tiles [1]. Some works adopt cache coherence protocols, as the distributed directory-based protocol [2, 12]. Other works [8, 9, 15], argue that cache coherence protocols are not scalable for many-cores due to their high cost in terms of synchronization overhead and energy consumption observed, specifically, for streaming data-flow applications [15]. Instead, the alternative is to rely upon software-managed scratchpad memory close to each CPU, with the communication among CPUs initialized by software [1, 7, 8, 14, 15]. Some designs cluster the cores and assign a shared memory for each cluster [1, 7, 8, 13]. Such an approach has benefits since it preserves data locality inside the cluster, speeding intra-cluster communication. The side effect is the heterogeneous latency (jitter) caused to access off-cluster data and the task mapping complexity for applications that do not fit in one cluster.

The 5th column classifies works by communication implementation. NoCs are interesting solutions to build scalable and high throughput interconnections [1, 2, 4, 6–8, 11, 12, 14, 15]. Some platforms adopt hybrid approaches (bus + NoC), specifically, those that adopt clustering [1, 7, 8]. Thus, a bus implements the communication intra-cluster, while a NoC implements the communication inter-cluster. The 6th column details if the work is open-source. Finally, the last column details if works adopt some GUI-based method. As can be observed, GUI-based debugging is the main gap found in the state-of-the-art. In [11], the authors use a GUI to generate the many-core according to some properties and also to collect NoC-centered statistics during simulation. In [14], the authors use a GUI to assist platform debugging during simulation. They focus on CPU scheduling and, as in [11], mostly into communication statistics. This work advances the state-of-the-art of

[3] www.veripool.org/verilator/

Table 1: State-of-the-art (from 2017) many-core platforms.

Name, Work, Year	Language	Computation	Memory	Communication	Open	GUI Debug
ProNoC, [11], 2017	SV	Generic (not the focus)	Distributed/core	Parameterizable VC-based NoC (inter-core)	✓	To generation (NoC-centered)
CoreVA-MPSoC, [1], 2017	N.A.	CoreVA	L1/core + L2 slice/Cluster	Bus (intra-cluster) + 1x NoC (inter-cluster)	-	-
HERO, [8], 2018	SV	RISC-V + ARM	L1/Cluster + L2/System	Bus (intra-cluster) + 1x NoC (inter-cluster)	✓	-
RVNoC, [6], 2018	Verilog	RISC-V	Distributed/core	Parameterizable VC-based NoC	✓	-
Celerity, [4], 2018	Verilog	RISC-V + NN Accelerator	L1 cache/core	1x NoC	✓	-
OpenPiton, [2], 2019	Verilog, SV	RISC-V	L1,L1.5/core + L2 slice/core	3x NoCs	✓	-
Memphis, [14], 2019	VHDL, System-C	MIPS-I	Distributed/core	1x NoC	✓	NoC + CPU scheduling
Kamaleldin et al., [7], 2020	N.A.	RISC-V	L1/core + L2/cluster	Bus (intra-cluster) + NoC (inter-cluster)	-	-
BlackParrot, [12], 2020	SV	BlackParrot Core + Heterogeneous tiles	L1/core + L2 slice/core	1x NoC	✓	-
Savas, [15], 2020	Verilog + Chisel	RISC-V + Accelerator	L1/core + Shared L2 slice/core	1x NoC	✓	-
Gem5-X, [13], 2021	C++	ARMv8 + Accelerators	L1/core + L2/Cluster + L3/System	Bus	✓	-

GUI debugging for many-core by addressing not only communication but memory statistics and energy consumption. As could be observed from previous columns, systems are increasingly complex, and memory is a key player in driving the chip's performance and energy efficiency. Therefore, easily understanding the memory statistics and the energy consumption is fundamental in the design of next-generation many-cores.

3 MANY-CORE GUI DEBUGGING

This section presents the contributions (2) and (3). Figure 2 presents an overview of *ManyGUI* debugging framework. It includes the steps of event extraction, classification, and representation. The event extraction is built inside the hardware by monitoring instruction events from the CPU, memory events from the memory subsystem, and NoC events from the NoC router. These events are inserted into a database during the simulation. In parallel, the *ManyGUI* desktop application (implemented in Java), read from this database, classifies, and displays the event graphically. *ManyGUI* desktop application is composed of two parts: (1) a back-end part, implemented by an event classification engine (**ECE**); (2) a front-end part, implemented by the Event Graphical Interface (**EGI**). The back-end part reads from the database and creates a logical structure of the events in the memory of the *ManyGUI* program. The front-end part displays graphically and intuitively the events to the developer in a set of frames. The following subsection will enter in details of each part.

3.1 Event Monitoring

Figure 2 shows an example of a many-core with 4 tiles. To support event monitoring, we implemented inside each tile three types of event monitors: Instruction Counter Monitor (ICM), Memory Event Monitor (MEM), and Packet Counter Monitor (PCM). The role of these monitors is to extract events from the hardware model during the simulation and insert them into a database structure.

The monitors implementation consists of non-synthesizable code inside key hardware points, which are generic to any modern many-core: CPU (ICM), memory (MEM), and NoC router (PCM). Inside CPU, the ICM extracts *computation* events by monitoring the executed instruction. Each ICM event consists of a timestamp in clock cycles, an instruction ID, and a class ID representing its instruction class (e.g., branch, ALU, multiplication, and floating-point). Inside

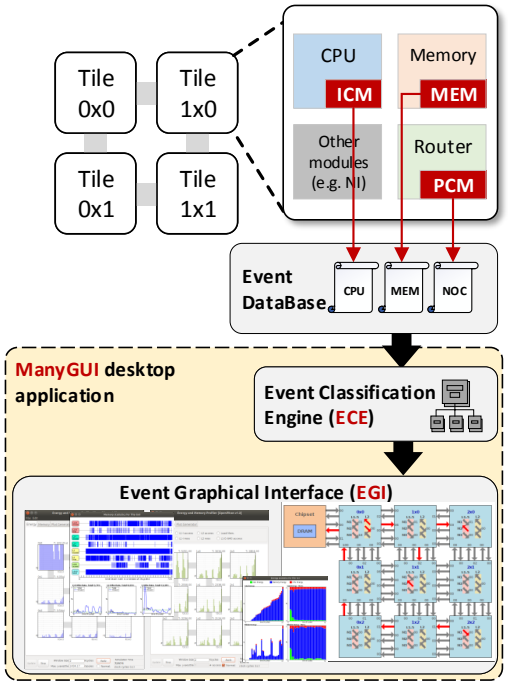


Figure 2: Proposed *ManyGUI* debugging framework for event extraction, classification, and representation. It includes hardware-implemented monitors: ICM = Instruction Counter Monitor, MEM = Memory Event Monitor, PCM = Packet Counter Monitor. The monitored data are stored in an Event DataBase in the host desktop computer. The *ManyGUI* desktop application reads from the database, classifies the events by the Event Classification Engine (ECE), and shows them by the Event Graphical Interface (EGI).

the memory, the MEM extracts *memory* events. Memory events vary according to the many-core memory organization. Cache memories are currently common implementations, and one of the *ManyGUI* goals is to be suitable to caches. Therefore, memory events types include L1 access, L1 miss, L2 access, L2 miss, and DRAM accesses. Each MEM event has a timestamp and an ID representing the class.

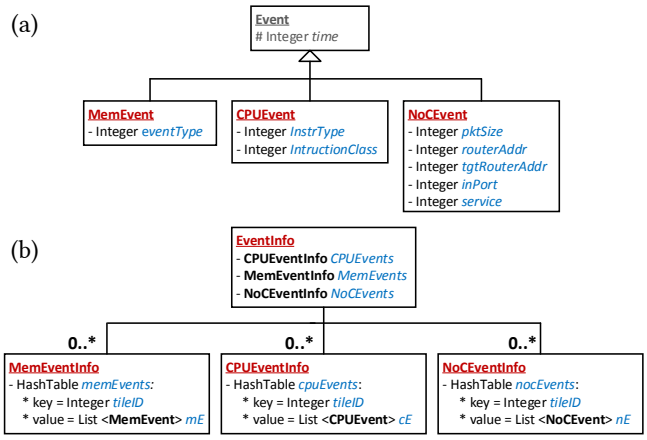


Figure 3: Class diagram of the Event Classification Engine (ECE). (a) Classes that model the 3 collected events. (b) Classes that store all the events read from the database.

Finally, inside the NoC routers, the PCM extracts *communication* events. The PCM generates one new event each time a packet enters in a router by a given input port. The PCM comprises a timestamp, packet size, current router address, destination router address, input port, and packet service.

An important functional property of the monitors is that they register the events in three respective databases shown in Figure 2. Such a database can be implemented as a structured database (as Structured Query Language (SQL)) or simply by using text files on the host OS directory structure.

3.2 Event Classification Engine (ECE)

The Event Classification Engine (ECE) aims to read the events from the database and compute statistics at runtime. It was designed to provide a simplified set of data structures that the *ManyGUI* front-end can easily access. We use object-oriented implementation to classify the data. Figure 3 shows two main class diagrams implemented by ECE. For clarity, we omit the functions of these classes and keep only the attributes and UML relationships among them.

Figure 3(a) shows the class diagram of the basic classes to represent events. It consists of 3 classes (*MemEvent*, *CPUEvent*, *NoCEvent*) which store one single event from database. These 3 classes inherit from the *Event* abstract class, which has the attribute *time*. The *MemEvent* class has one attribute that stores the type of memory event (e.g.: L1 miss, L1 access, L2 miss). The *CPUEvent* class stores the additional attributes of instruction type (e.g.: add, sub, jal), and instruction class (e.g.: ALU, branch, multiplications). Finally, the *NoCEvent* class stores all attributes required to one packet (size, current router address, target router address, input port, and service). Figure 3(b) shows the class diagram that stores all events from the database. At runtime, the ECE pools the database during a fixed interval of time (200 milliseconds), gets all new incoming events within this time, and classifies them by storing in a high-level class structure. To implement such a structure, we design a class named *EventInfo*. It is the top-level class that stores all events of the system. It is composed of 3 other classes: *MemEventInfo*, *CPUEventInfo*, and *NoCEventInfo*. Those 3 classes implement a similar behavior which can be explained as a single one. Each one has a hash table, where keys represent the tile IDs and values represent a list of events for that tile. This hash table allows to speed up the access to events by the EGI since it rapidly extracts event associated to a given tile due to its average access time complexity of $O(1)$ [3]. These 3 classes also implement functions to collect statistics from the events, which include: cache miss rate, NoC links throughput, CPU energy, memory energy, and NoC energy.

The cache miss rate is calculated by dividing the number of cache misses by the cache access. The NoC link throughput is calculated by dividing the total time the router was busy handling packets by the total simulated time. The energy estimation is performed by assuming a previous hardware characterization of the many-core (out of the scope of this work). In this work we adopted the characterization provided in [10] for memory and NoC, and in [16] for CPU. Based on such energy characterization, the developer can extract the following information: (1) energy at *computation* level: the energy per CPU instruction class; (2) energy at *memory* level: the energy per memory miss and hit, and (3) energy at *communication* level: the energy to transmit one-hop flit into the NoC. These energy values are informed to the tool using a configuration file which is loaded when *ManyGUI* is initialized (section 3.4).

In summary, the goal of ECE is to provide a structured event classification to compute statistics about the events. This structure is clustered in the class *EnergyInfo*, which is the input of the front-end EGI.

3.3 Event Graphical Interface (EGI)

The Event Graphical Interface (EGI) is composed of a set of frames that brings information about the system's communication, memory, and energy. Figure 4 shows the principal frames of *ManyGUI* . The next subsections explain the functional properties of the frames. The explanation of the data content is left to section 4.

3.3.1 Communication. The communication is the initial frame of the tool and is called when *ManyGUI* is opened. Figure 4(a) shows an example for a many-core with 9 tiles (3x3 dimension). This frame aims to show the interconnection among cores, the packets traversing the NoC in a given time (red arrows), and the link utilization of each router input port. The frame contains several configuration panels at bottom. In the panel detailed by marker (1), the user can control the displaying of the packet events (start, stop, show next packet). When a new NoC event is read by the tool, it colors in red the arrow between the current router and the previous router. By keeping the arrows colored until the packet reaches the destination router, it is possible to show the full path of the packet, as zoomed by marker (5), which shows a packet generated by tile 0x0 and that arrived at tile 0x2, crossing the router of tile 0x1. Marker (2) shows the speed control panel, in which the velocity of displaying the NoC event can be controlled. Marker (3) allows backing the packet displaying to a specific period in the simulation time. Finally, marker (4) allows for seeing the packet service, which carries the meaning of that packet. In this example, the packet is a `DATA_ACK`, representing an acknowledgment message from a high-level cache to a low-level cache.

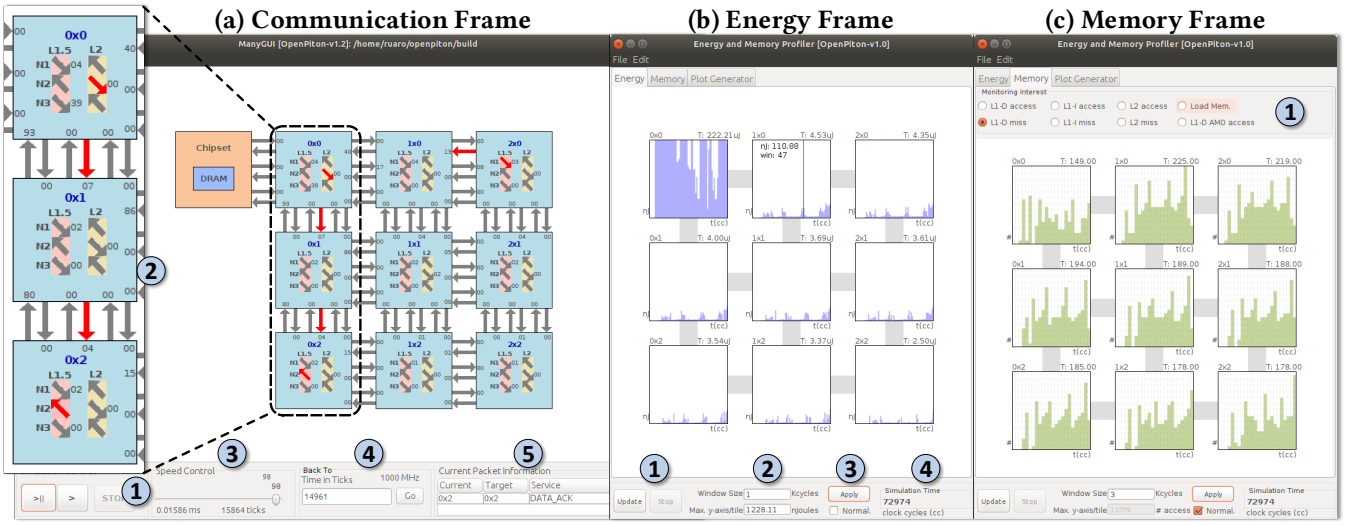


Figure 4: *ManyGUI* main frames.

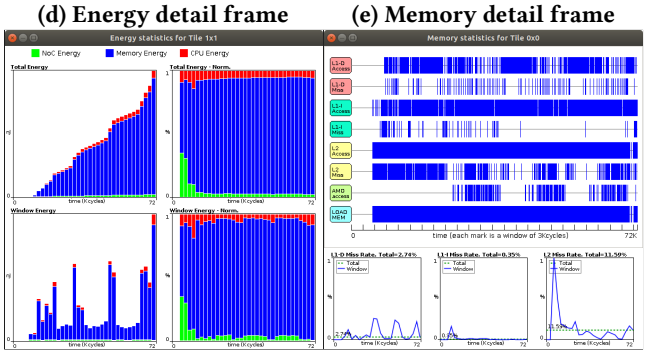


Figure 5: *ManyGUI* detailed frames.

The debug possibilities of this frame include the validation of the routing algorithms, communication load distribution, link utilization, and packet content. Additionally, we observe that this frame is helpful when the simulation hangs due to a software error inside a given tile. Using the tool, the developer can quickly identify the tile responsible for this bug by observing that it stopped consuming packets, which leads to a cascade effect in the NoC. This awareness is more difficult (in terms of time) to be reached just using log files or waveform since the developer needs to trace log by log (or signal by signal) along the time to find the actual source of the hang.

3.3.2 Energy. From the communication frame, the user can open the energy frame depicted in Figure 4(b). The goal of this frame is to detail the energy consumption for each tile. The energy is shown in the form of a plot for each tile in a 2D-mesh layout. The energy measurement is broken in monitoring windows, so each tile bar at the x-axis represents the energy accumulated on one monitoring window. The y-axis of each tile represents the energy in nJ. The frame contains some configuration panels at the bottom. In the panel detailed by marker (1), the user can enable or disable the update of new events during simulation. In the panel of marker (2), the user can configure the size of the window. In this example, it was configured to the default size, which is 1 Kcycles. The panel of marker (3) allows the user to choose if the displayed values inside each tile must be normalized to the worst value among all tiles or according to a specified value. This option is helpful to allow a quick and fair comparison among the energy consumed by all tiles. However, sometimes one tile can consume much more energy than the other ones, which will hide the energy bars of the low-energy tiles. In such a case, the user can specify an absolute value, which will be used as a reference to set the maximum value of the y-axis for all plots. Finally, the panel of marker (4) shows the current time of the simulation in clock cycles. Another debug property of this frame is to show the tile energy in a detailed frame (by clicking over the tile), shown in Figure 5(a). There, the developer can observe 4 plots related to energy: the total accumulated energy (top-left) and its normalized version (top-right), and the energy per window (bottom-left) and its normalized version (bottom-right). Each energy bar of those plots is broken in the energy of CPU, memory, and NoC, allowing the user to profile quickly which many-core subsystem is spending more energy at a given time.

The debug possibilities of the energy frame include: achieving a comprehensive view of energy for all tiles, detecting tiles that are energy-hungry, observing the total energy and energy per window over time, comparing the energy consumption among different many-core subsystems (CPU, NoC, memory).

3.3.3 Memory. Figure 4(c) shows the memory frame. Its layout is similar to the energy frame, especially regarding the configuration panels at the bottom. This frame includes an upper panel highlighted by marker (1), which selects which memory statistics must be displayed, e.g., L1-D access, L1-D miss, L2 access. Thus, the user can rapidly see the statistics of different memory properties. The y-axis of each tile represents the number of events (e.g., number of L1-D miss), and the x-axis represents the monitoring window. Similar to the energy frame, this frame allows to click over the tile

to achieve a detailed memory statistic frame. This action generates the frame depicted in Figure 5(b). There, the developer can observe 4 plots: at the top, one big plot shows the memory event scheduling. This plot is proper to compare different memory events and see how they can influence each other and impact tile's energy consumption. At the bottom, there are 3 plots showing the miss rate for L1-D (left), L1-I (center), and L2 (right). Each of these plots also brings the total miss rate achieved at that given simulation time (represented by the horizontal dashed line).

The debug possibilities of the memory frame include: achieving a comprehensive view of memory behavior for all tiles, detecting undesired memory behavior in a given tile, crossing different memory parameters with the energy frame, observing the memory event scheduling and the miss rate of different cache levels.

3.3.4 Plot Generator. An additional frame shown in Figure 6 consists of a plot generator, which, based on a set of filters, allows the developer to generate in a vector format any specific plots shown in the energy and memory frames. To implement this, we integrate the tool with Matplotlib[4] (python-based script to generate plots). To generate a new plot, the developer must filter which parameters must be considered (e.g., L1-D access of tile 1x0, the energy of tile 0x0). Once the generate command is pressed, the vector version of the plot will be shown in a separated frame. This plot can be saved (e.g., in PDF) for future use in presentations or reports.

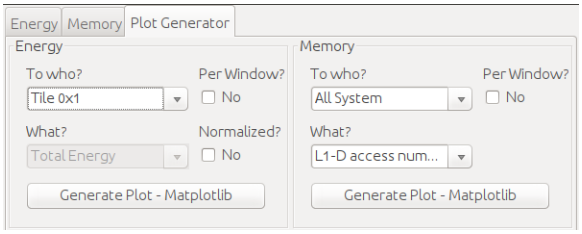


Figure 6: Plot generator frame.

3.4 Configuration file

At initialization, *ManyGUI* demands a configuration file as input. Table 2 describes the parameters of the configuration file. From left to right, the columns detail the name, the unit, the numeral format, and the description. This file allows the developer to configure important parameters, such as the XY dimension, the router that implements off-chip connection, frequency, flit width, NoC number, the monitoring window used by energy and memory frames, and the energy for events of instructions, memory, and NoC.

4 CASE STUDY: OPENPITON MANY-CORE

OpenPiton is a state-of-the-art silicon-proven many-core developed by Princeton University [2, 10]. Figure 7 presents an overview of OpenPiton. Tiles are organized in a 2D-mesh topology, with routers of tile 0x0 connected to off-chip peripherals. Each tile has a CPU (with L1 D and I caches), one L1.5 cache (a replica of L1 cache), and a slice of L2 cache, modeled as a distributed/shared memory. The

[4] www.matplotlib.org

Table 2: Configuration file description.

Name	Unit	Form.	Description
X Dimension	Number of tiles	$\mathbb{N}$	Horizontal length of many-core
Y Dimension	Number of tiles	$\mathbb{N}$	Vertical length of many-core
Offchip XY addr	NoC XY address	$\mathbb{N}$x$\mathbb{N}$	Address of the router connected to the offchip interface
NoC number	NoC	$\mathbb{N}$	Amount of NoC or sub-NoCs
Flit size	Bits	$\mathbb{N}$	Size of one NoC flit
Frequency	MHz	$\mathbb{R}_{>0}$	Many-core running frequency
Monitor Window	Clock cycles	$\mathbb{N}$	Size of one monitor window
EnergyInstru<ALU>	pJ	$\mathbb{R}_{>0}$	Energy per ALU instruction
EnergyInstru<Mult>	pJ	$\mathbb{R}_{>0}$	Energy per Branch instructions
EnergyInstru<...>	pJ	$\mathbb{R}_{>0}$	...
EnergyMem<L1 miss>	pJ	$\mathbb{R}_{>0}$	Energy per l1 miss
EnergyMem<L1 hit>	pJ	$\mathbb{R}_{>0}$	Energy per l1 hit
EnergyMem <...>	pJ	$\mathbb{R}_{>0}$	...
EnergyNoC <Flit>	pJ	$\mathbb{R}_{>0}$	Energy per flit hop

cache implements the directory-based MESI coherence protocol. The system has 3 packet-switching NoCs with credit-based flow control and XY routing algorithm. The 3 NoCs are required to keep the coherence protocol free of deadlocks. Figure 7 shows the connection between the memory (L2 and L1.5 cache) and the local ports of routers of each NoC.

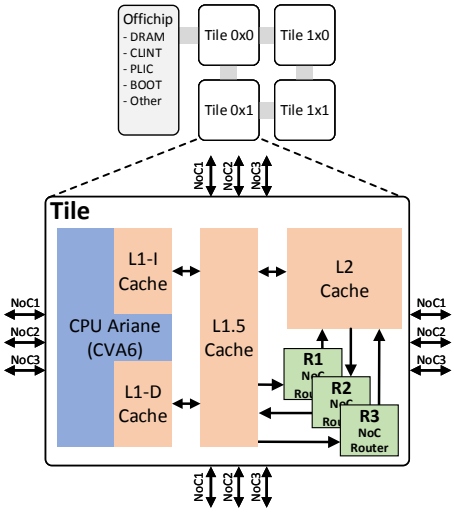
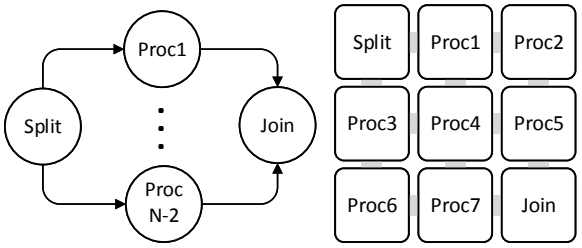


Figure 7: Hardware overview of OpenPiton many-core.

4.1 Experimental Setup

Intending to address an easy-to-follow debugging experiment, we build the OpenPiton many-core with 9 tiles (3x3 dimension) and run on this system a synthetic application shown in Figure 8(a). The application consists of 3 classes of tasks. A single *Split* task receives input data and splits it among several parallel tasks, called *Proc*, which process the data slice in parallel. A *Join* task receives the processed data gathering them to compose the final result. This parallel application pattern is commonly found in image and video processing applications [5]. The application runs during 3 iterations. The size of the input data is 26.25 KB, meaning that each *Proc* task handles 3.75 KB at each iteration. The *Proc* cost function has a complexity of $O(n)$, where n is equal to 960 (3.75 KB / 32 bits).

Figure 8(b) shows the task mapping of the application in OpenPiton. Each task runs in one tile in a bare-metal fashion (without OS). The application communicates using semaphore operations of *up()* and *down()* to produce and consume data, respectively. Due to the absence of OS, the semaphore implementation spins the lock until it is released by the communicating task pair, then it advances to modify the semaphore counter. This behavior is important since it will stress the cache memory, as will be evaluated in the next subsection.



(a) Task graph **(b) Task mapping**

Figure 8: Synthetic application.

4.2 Results

The goal of the results is not to enter in the merit of the application performance but evaluate the capabilities of *ManyGUI* to provide a rapid understanding of the many-core in terms of communication, memory usage, and energy consumption. For the sake of space, we reuse the frames of Figure 4.

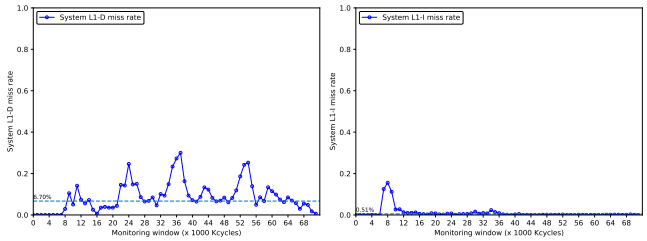
The first basic information that can be achieved, is the simulation time, shown in all frames. It can be observed that the application took 72,974 cycles to finish its execution (marker (4) of Figure 4(b)).

4.2.1 Communication. In communication frame, each tile was graphically modeled to represent the inputs and outputs of each NoC router. As OpenPiton has 3 NoCs, each tile has 3 sets of inputs and outputs that represent packets entering and exiting it. The inner draw of the tile represents the local interface between the memory with routers (which can be better understood jointly to the Figure 4). A descend arrow means a connection from a given memory module (L1.5 or L2) to the router. The ascendant arrow means the opposite, the connection from the router to the memory module.

The zoomed panel shown in marker (5) of Figure 4(a) points that L2 of tile 0x0 is sending a packet by NoC2 to the L1.5 of tile 0x2 at time 15,864 (marker 2). Another interesting behavior is link utilization. Observing the marker (5) it is possible to see that the left-most NoC input link (NoC1) of tile 0x0 is saturated, with 93% of utilization. This behavior is expected because the application fits entirely inside the L2 cache of tile 0x0, which stores the data and semaphores locks used for synchronization among all tasks. This makes all the cores communicate with L2 of tile 0x0. As the communication on NoC uses XY algorithm, the south input port receives most of the packets since 6 in 9 (66%) tiles use this port to communicate with tile 0x0. Looking to the east input port of tile 0x0, we can see a link utilization of 40%. This usage is lower since only 2 tiles (22%) use this link to communicate.

4.2.2 Memory. By observing the L1-D miss plots of Figure 4(c), it is possible to identify (after a warm-up period) 3 increasing patterns in the miss along the time. A similar pattern is found when selecting the L1-D access (not shown in the figure). These memory peaks occur because at each new application iteration (3 in total), the *Split* task overwrites new data in the shared buffers used by the *Proc* task to access the data. This invalidates the current data, increasing the L1-D misses. This behavior is endorsed by observing the detailed memory frame of Figure 5(b), specifically, the bottom-left plot, which shows the L1-D miss rate at tile 0x0. Clearly, 3 peaks can be observed in the miss rate, representing the 3 iterations of the application. In this same frame, it is also possible to observe that the L1-I miss rate (bottom-center plot) is higher during the warm-up period (up to 16 Kcycles) when the task's code is loaded inside the L1-I cache.

By using the plot generator frame, it is possible to look at the statistics closer. Additionally, it is possible to collect system-level metrics. As an example, Figure 9(a) and (b) shows the system miss rate for L1-D and L1-I, respectively. As expected, the L1-D miss rate (after warm-up) shows the same 3 patterns already explained. The horizontal line represents the average miss rate, which reached 6.7% in this case. In the L1-I miss rate, a peak occurs only during the warm-up, with the miss rate remaining below 0.1% after that. The average LI-D miss rate is 0.51%.



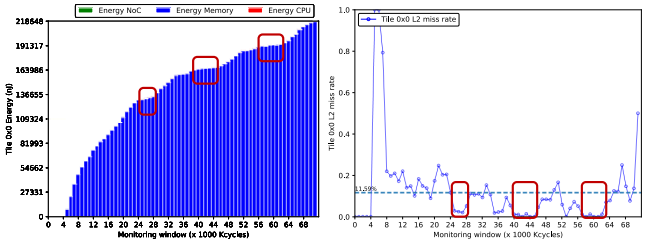
(a) L1-D miss rate **(b) LI-D miss rate**

Figure 9: System statistics.

4.2.3 Energy. The energy frame combined with the memory frame can lead to powerful analysis. First, lets focus on the energy consumption of a tile running a *Proc* task (tile 1x1), shown by Figure 4(e). The normalized total energy of the tile (top-right plot) shows that the NoC has a low influence on the total energy after the warm-up period, representing on average less than 4% of the tile's energy. This observation is in accordance with what was observed in the characterization of OpenPiton in silicon [10], which pointed out the low impact of NoC in the chip's energy consumption. Additionally, the most important player in energy consumption is the memory (> 88% after warm-up), which in this case is highly used since the semaphores implement busy-wait spinlock.

Looking into the all tiles energy plots in Figure 4(b), we can see that the 3 peaks observed in the memory frame are correlated with the energy consumption of all tiles, except for tile 0x0, which presents the largest energy consumption among all. This energy consumption is directly impacted due to the high L2 usage of tile 0x0 previously explained. As the energy of miss and hit increases according to the highest cache level [10], the L2 energy dominates

the energy consumption of tile 0x0. To perform a detailed analysis, we use the plot generator frame to extract the accumulated energy of tile 0x0 in Figure 10(a) and the L2 miss rate in Figure 10(b) (the red rectangles where manually inserted). It is possible to see in Figure 10(b) that there are 3 periods where the energy remains practically constant. Such points correlate precisely with the 3 L2 miss rate plateaus of Figure 10(a), caused due to the computing period of the *Proc* tasks. Such computation moments are outside of the critical region, and therefore do not push the L2 cache to keep the lock synchronized among tiles. As a last observation, Figure 10(a) also allows for achieving the total energy consumption of tile 0x0, which was 218,648 nJ (maximum value of y-axis).



(a) Accumulated energy **(b) L2 miss rate**

Figure 10: Tile 0x0 statistics.

5 CONCLUSION

This work has investigated that state-of-the-art open-source platforms have an important gap in debugability of high-level events, leading the developer to meticulous tasks of debugging and correlating log files and waveform to understand what is happening during the simulation. We proposed *ManyGUI* , a graphical tool to debug many-cores, composed of a data extraction and classification framework connected to a set of graphical frames that allows to represent the many-core behavior during simulation in an intuitive way. The choice to focus in high-level events (instructions, memory, and NoC packets), allows to build a comprehensive profile of many-core in terms of communication events, memory events and energy consumption. We implemented the support of *ManyGUI* to state-of-the-art silicon-proven many-core called OpenPiton. The experimental results show that *ManyGUI* can profile correctly NoC, cache memory, and energy consumption, presenting values already endorsed in silicon energy analysis. *ManyGUI* is an open-source tool accessible to download in github.com/Nooman-LabSTICC/manyGUI. Since OpenPiton is also a open-source tool, we believe that the combination of both will contribute to accelerate many-core research. Future works include to add thermal estimation and collect events at the software level, allowing to correlate a given software region with the statistics of memory and energy. In such a way, the developer could improve the software design accordingly to the presented statistics.

ACKNOWLEDGMENTS

This work is supported by the Agence Nationale de la Recherche under Grant No.: ANR-17-CE24-0018.

REFERENCES

[1] Johannes Ax, Gregor Sievers, Julian Daberkow, Martin Flasskamp, Marten Vohrmann, Thorsten Jungeblut, Wayne Kelly, Mario Porrmann, and Ulrich Rückert. 2018. CoreVA-MPSoC: A Many-Core Architecture with Tightly Coupled Shared and Local Data Memories. *Transactions on Parallel and Distributed Systems* 29, 5 (2018), 1030–1043. https://doi.org/10.1109/TPDS.2017.2785799

[2] Jonathan Balkind, Katie Lim, Fei Gao, Jinzheng Tu, David Wentzlaff, Michael Schaffner, Florian Zaruba, and Luca Benini. 2019. OpenPiton+ Ariane: The First Open-Source, SMP Linux-booting RISC-V System Scaling From One to Many Cores. In *Workshop on Computer Architecture Research with RISC-V (CARRV-)*. ACM, USA, 1–6.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press, Cambridge, Massachusetts.

[4] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawai, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, Aporva Amarnath, Bandhav Veluri, Paul Gao, Anuj Rao, Gai Liu, Rajesh K. Gupta, Zhiru Zhang, Ronald Dreslinski, Christopher Batten, and Michael Bedford Taylor. 2018. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. *IEEE Micro* 38, 2 (2018), 30–41. https://doi.org/10.1109/MM.2018.022071133

[5] Karol Desnos, Maxime Pelcat, Jean-François Nezan, and Slaheddine Aridhi. 2016. Distributed Memory Allocation Technique for Synchronous Dataflow Graphs. In *International Workshop on Signal Processing Systems*. IEEE, USA, 45–50. https://doi.org/10.1109/SiPS.2016.16

[6] M. A. Elmohr, A. S. Eissa, M. Ibrahim, M. Khamis, S. El-Ashry, A. Shalaby, M. AbdElsalam, and M. W. El-Kharashi. 2018. RVNoC: A Framework for Generating RISC-V NoC-Based MPSoC. In *PDP*. IEEE, UK, 617–621. https://doi.org/10.1109/PDP2018.2018.00103

[7] Ahmed Kamaleldin, Salma Hesham, and Diana Göhringer. 2020. Towards a Modular RISC-V Based Many-Core Architecture for FPGA Accelerators. *IEEE Access* 8 (2020), 148812–148826. https://doi.org/10.1109/ACCESS.2020.3015706

[8] Andreas Kurth, Alessandro Capotondi, Pirmin Vogel, Luca Benini, and Andrea Marongiu. 2018. HERO: An Open-Source Research Platform for HW/SW Exploration of Heterogeneous Manycore Systems. In *Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems* (Limassol, Cyprus) *(ANDARE '18)*. Association for Computing Machinery, New York, NY, USA, Article 5, 6 pages. https://doi.org/10.1145/3295816.3295821

[9] Andreas Kurth, Wolfgang Ronninger, Thomas Benz, Matheus Cavalcante, Fabian Schuiki, Florian Zaruba, and Luca Benini. 2021. An Open-Source Platform for High-Performance Non-Coherent On-Chip Communication. *Transactions on Computers* NA (2021), 1–1. https://doi.org/10.1109/TC.2021.3107726

[10] Michael McKeown, Alexey Lavrov, Mohammad Shahrad, Paul J. Jackson, Yaosheng Fu, Jonathan Balkind, Tri M. Nguyen, Katie Lim, Yanqi Zhou, and David Wentzlaff. 2018. Power and Energy Characterization of an Open Source 25-Core Manycore Processor. In *International Symposium on High Performance Computer Architecture*. IEEE, Austria, 762–775. https://doi.org/10.1109/HPCA.2018.00070

[11] A. Monemi, J.W. Tang, M. Palesi, and M.N. Marsono. 2017. ProNoC: A low latency network-on-chip based many-core system-on-chip prototyping platform. *Microprocessors and Microsystems* 54 (2017), 60–74. https://doi.org/10.1016/j.micpro.2017.08.007

[12] Daniel Petrisko, Farzam Gilani, Mark Wyse, Dai Cheol Jung, Scott Davidson, Paul Gao, Chun Zhao, Zahra Azad, Sadullah Canakci, Bandhav Veluri, Tavio Guarino, Ajay Joshi, Mark Oskin, and Michael Bedford Taylor. 2020. BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs. *IEEE Micro* 40, 4 (2020), 93–102. https://doi.org/10.1109/MM.2020.2996145

[13] Yasir Mahmood Qureshi, William Andrew Simon, Marina Zapater, Katzalin Olcoz, and David Atienza. 2021. Gem5-X: A Many-Core Heterogeneous Simulation Platform for Architectural Exploration and Optimization. *ACM Trans. Archit. Code Optim.* 18, 4, Article 44 (July 2021), 27 pages. https://doi.org/10.1145/3461662

[14] Marcelo Ruaro, Luciano L. Caimi, Vinicius Fochi, and Fernando G. Moraes. 2019. Memphis: a framework for heterogeneous many-core SoCs generation and validation. *Design Automation for Embedded Systems* 23, 3-4 (2019), 103–122. https://doi.org/10.1007/s10617-019-09223-4

[15] Süleyman Savas, Zain Ul-Abdin, and Tomas Nordström. 2020. A framework to generate domain-specific manycore architectures from dataflow programs. *Microprocessors and Microsystems* 72 (2020), 102908. https://doi.org/10.1016/j.micpro.2019.102908

[16] Florian Zaruba and Luca Benini. 2019. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-Nm FDSOI Technology. *IEEE Trans. Very Large Scale Integr. Syst.* 27, 11 (nov 2019), 2629–2640. https://doi.org/10.1109/TVLSI.2019.2926114