A High-level Model to Leverage NoC-based Many-core Research

Iaçanã Ianiski Weber, Angelo Elias Dalzotto, Fernando Gehm Moraes
School of Technology, Pontifical Catholic University of Rio Grande do Sul – PUCRS – Porto Alegre, Brazil
iacana.weber@edu.pucrs.br, angelo.dalzotto@edu.pucrs.br, fernando.moraes@pucrs.br

***Abstract*—This work presents *Chronos-V*, a Many-Core System-on-Chip (MCSoC) that adopts abstract hardware modeling, executing the freeRTOS Operating System (OS) at each processing element (PE). The system architecture contains two regions: (i) General Purpose Processing Elements (GPPE), responsible for executing user applications; (ii) peripherals that provide IO capabilities or hardware acceleration to the system. The freeRTOS kernel provides scheduling for application tasks, with modules added to it providing a Message Passing Interface (MPI) and system management. The goal is to provide a platform with a parameterizable hardware model that allows software development and evaluation of system management techniques. Results evaluate the simulation speedup by comparing the abstract model simulation against an RTL simulation in systems with up to 100 PEs, and thermal management techniques added to the freeRTOS as an API.**

***Index Terms*—Many-Core System-on-Chip; High-Level Model; Heterogeneous Many-Core; Network-on-Chip.**

I. Introduction and Related Work

Recent advances in the industry, such as the ET-SoC-1 [1], emphasize that many-core system-on-chip (MCSoCs) is the predominant paradigm employed to meet high-performance computing required by Machine Learning applications or Cyber-Physical systems [2]. In this context, application testing and management of security, power consumption, and performance are an important issues in MCSoCs design [3].

In the current competitive scenario that demands low time-to-market for new products, the capability to parallelize hardware and software development is paramount. Therefore, a platform that allows developers to submit the code into a system model enables software iterations in the early design stages of a new product design. However, application testing to promote fast software development is not the only concern. MCSoCs design offer challenges in the management of system resources. System management includes actions to keep the system operating at safe conditions, such as controlling Dynamic Voltage and Frequency Scaling (DVFS) to maintain safe temperature and power, and deliver resources to applications, enabling them to meet their constraints (e.g., real-time deadlines). System management requires hardware and software modules to meet system and user constraints.

The *goal* of the present work is to propose a high-level simulation platform for MCSoCs research, called *Chronos-V*. This platform integrates an instruction-level model of RISC-V processors into a 2D-mesh Network on Chip (NoC) to create the MCSoC model. The literature [4]–[11] presents tools and abstract models for MCSoCs to enable software development and system validation before the final hardware design.

Many of those works provide platforms that allow hardware exploration. Lemaire et al. [5] introduced a modeling environment built around a SystemC-TLM kernel to explore application mapping and other hardware features at early design stages. Helmstetter et al. [6] proposed a platform to evaluate the hardware costs of a specific architecture according to application needs. The platform proposed by Duenha et al. [7] models heterogeneous systems providing energy consumption estimation. Lima et al. [10] proposed a platform to compare and select routing algorithms.

Furthermore, software development models seek simulation speed and not necessarily performance estimation. Madalozzo et al. [8] proposed a virtual platform that combines different architecture description languages and simulators to improve software productivity in many-core systems, providing fast software validation and debuggability. Cataldo et al. [9] built an abstract platform on GEM5 to model NoC-based MPSoCs and evaluate synchronization mechanisms of parallel applications.

This work provides a platform with a parameterizable hardware model that allows software development and hardware behavior testing while preserving temporal and spatial traffic distribution. The knowledge of the traffic behavior may be used to identify hotspots [12] or detect anomalies that may signalize attacks [13].

The *original* contributions of the present work include:

- a many-core platform with a state-of-the-art processor that uses an open-source Operating System (OS), allowing fast software evaluation in complex systems up to hundreds of processors;
- modeling that preserves the behavior at the instruction cycle level, both for the processors and the NoC;
- a set of communication and management Application Programming Interface (API), loosely coupled to the OS to enable their portability to other OSs;
- support for management techniques, such as the Observe-Decide-Act (ODA) loop [14], using the APIs and hardware monitors.

II. *Chronos-V* Model Overview

Figure 1 presents a general view of the *Chronos-V* many-core, organized in three layers.

The hardware layer corresponds to the physical components. Interrupt signals and MMRs (Memory-Mapped Registers) implement the interface between hardware and OS layers. The MCSoC is composed by Processing Elements (PEs) interconnected through a NoC. Each PE contains a RISC-V processor, a local memory, a network interface, a NoC router, and an instruction counter and it has an unique address that

978-1-6654-8128-1/22/$31.00 ©2022 IEEE

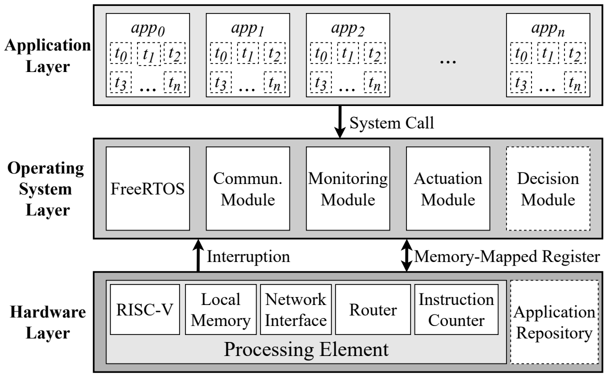

Fig. 1. *Chronos-V* system layers. Dotted borders represent centralized modules, whereas contiguous borders represent modules replicated throughout the system.

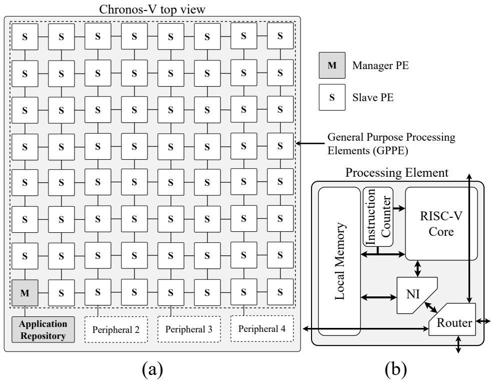

Fig. 2. *Chronos-V* many-core. The system has two main regions, (*a*) General Purpose Processing Elements (GPPE); (*b*) peripherals connected to the NoC border ports.

identify its position into the NoC. The Application Repository (App_{rep}) is a peripheral that provides application binary code to run in the system.

The application layer encompasses general-purpose applications (app_0, app_1, ..., app_n). Each application has one or more tasks (t_0, t_1, ..., t_n). Each task executes in a given PE, whose address is defined by a mapping heuristic. The communication paradigm adopted is the Message Passing Interface (MPI), which is available to applications tasks through system calls that trigger the OS layer.

The OS layer uses FreeRTOS (https://www.freertos.org/), an open-source real-time OS responsible for managing the tasks assigned to the PE. The FreeRTOS received four additional modules: Communication, Monitoring, Actuation, and Decision. The Communication Module serves its API to the Application Layer, implements the MPI message exchange, and implement drivers to communicate with the NoC. The remaining modules implement the ODA management functions.

III. *Chronos-V* Model

A. Hardware Layer

Many-cores may fall in two main classifications: homogeneous and heterogeneous [15]. *Chronos-V* fits in the second category because it contains two types of PEs: the General Purpose Processing Elements (GPPE) and peripherals. Figure 2(a) overviews the many-core components, highlighting the GPPE region that contains a set of homogeneous PEs, which execute general purpose applications.

Figure 2(b) details the general-purpose PE architecture. The many-core employs a RV32IM (32-bit RISC-V with multiply/divide extension) **core**. The core is connected to the local memory and to the **network interface** (NI) that has direct memory access (DMA) capability, allowing the processor to delegate data transfers between NoC and local memory. At the OS level, an API configures the NI for packet reception and packet transmission, reducing the processor overhead related to packet handling.

The **local memory** has two ports and stores code and instructions. This memory model was selected to reduce the system complexity and power consumption related to cache controllers and NoC traffic to supply cache lines. The NoC **router** has the following characteristics: wormhole packet switching, 2D-mesh topology, XY routing, round-robin arbitration, input buffering, and credit-based flow control.

The **instruction counter** module classifies and counts the number of instructions executed by the processor to enable power and temperature estimations. This module is included in the system when a given management strategy requires monitoring at the PE level. The OS periodically accesses the monitored information and transmits it to the management functions.

Peripherals provide specialized services for the system and applications (as hardware accelerators). The system requires at least one peripheral: the App_{rep}. The App_{rep} transmits to a manager PE (M_{PE}) requests for new applications to execute in the GPPE area. This M_{PE} runs the task mapping, sending back to the App_{rep} the address mapped for each task. In the sequence, the App_{rep} transmits the tasks binary code to the PEs in the GPPE area where they were mapped to. Note that the M_{PE} runs the same OS as the remaining PEs, being able of executing decision functions, such as application mapping, its the only difference.

B. Operating System Layer

The main component of the OS layer is the FreeRTOS kernel that allows tasks to be organized as a collection of independent threads. The kernel schedules these threads according to their priority. Kernel functionality is extended by the following *system modules*:

- **communication module** – enables communication among tasks mapped at different PEs;
- **monitoring**, **decision**, and **actuation modules** – manage the system using the ODA control method.

Note that general-purpose tasks have lower scheduling priority when compared to system modules.

Summarizing, the OS layer encompasses the FreeRTOS kernel that schedules the general-purpose tasks and the system modules. The proposed architecture aims to achieve a generic and modular OS layer, enabling designers to perform experiments related to system management by allowing to easily replace system modules according to their needs.

The **communication** module works as follows:

- When a PE receives a packet, this module reads this packet header to identify its *service*, i.e., the action the packet is requiring and provides the appropriate address to write the packet payload. A packet may contain user-level services, such as a request for data or data deliver, or OS services, such as start/stop a task, allocate a task, inform the end of execution of a given task, change the PE operating frequency, among others.

- This module configures MMRs and signalizes to the NI to inject the packet into the NoC. A packet transmission assumes that it is stored in the local memory. Therefore, after injecting the packet into the NoC, the NI signalizes to the communication module to release the memory reserved for that packet, opening space for new packets.

Figure 3 illustrates the communication flowing from a *producer* PE to a *consumer* PE. When the producer PE is ready to send a packet, the `SendRaw()` function configures the NI to inject the packet into the NoC. The consumer PE NI interrupts the processor upon the packet header reception. Then, the OS configures the memory address to write the incoming packet. After writing into the memory, the packet is available to the task waiting for its reception.

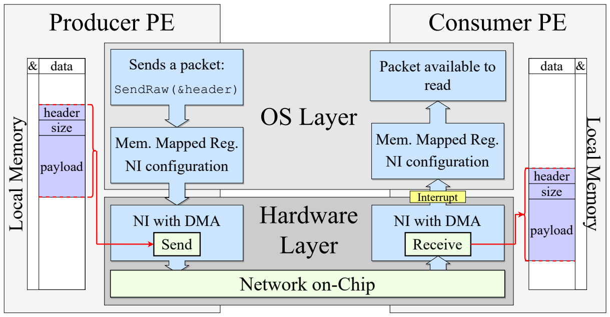

Fig. 3. Communication flowing from a producer PE to a consumer PE.

The **monitoring module** is triggered periodically to acquire local data to feed the decision module. The monitoring window period is defined at design time. In the current *Chronos-V* implementation, this module has two sensing sources. The first one is the *Instruction Counter*, which classifies and counts the amount of executed instructions in a given monitoring window. The second one is the *Router Counter* (embedded into the router), which counts the number of packets and flits traversing the router in a monitoring window. Using these counters, the monitoring module estimates the PE energy according to Equation 1. At the end of the monitoring window, the OS sends the computed energy to the M_{PE}.

$$\mathbf{Energy}_{PE_x} = \sum_{i=0}^{C-1}(ctI[i] \cdot E_c[i]) + ctM \cdot E_{mem} + ctF \cdot E_{flit} \quad (1)$$

Where:

- C: Number of categories (e.g., load, store, arithmetic, branch, jump, multiplication, and division) that the *instruction counter* classifies the executed instructions;
- $\sum_{i=0}^{C-1}(ctI[i] \cdot E[i])$: processor energy component, obtained by multiplying the number of executed instructions in a given category ($ctI[i]$) by the average energy to execute an instruction of the category ($E_c[i]$);
- $ctM \cdot E_{mem}$: memory energy component, corresponds to the number of executed load and store instructions (ctM) multiplied by the average energy cost to access the memory (E_{mem});
- $ctF \cdot E_{flit}$: router energy component, corresponds to the number of flits that traversed the router (ctF) multiplied by the average energy cost to transmit one flit (E_{flit}).

The **decision module** runs in the M_{PE}. This module receives raw monitoring data from every PE periodically according to the monitoring module window. The management heuristic uses the monitoring information to make decisions according to design goals. For example, suppose the decision module detects a temperature violation or a trend in temperature increase in a given PE. In this case, this module may send a message to the actuation module running at this PE to migrate a task or reduce the frequency (DVFS). The application mapping is also a function of the decision module.

The **actuation module** runs at each PE, including the M_{PE}. The actuation module executes actions generated by the decision module.

When a PE receives a task allocation packet, the actuation module allocates memory to the arriving task. It informs the task starting memory address to the NI, which writes into the memory the object code sent by the App_{rep}. After receiving the task code, the OS schedules the task to execute in the PE.

Figure 4 illustrates the task migration protocol. The decision module transmits a `migration request` packet (event 1) to the source PE (2). The source PE runs the task to be migrated until it reaches a safe migration state (3). Reaching this state, the OS stalls the task, sending a `migration acknowledge` (4) to the M_{PE}. Upon receiving this packet, the M_{PE} stalls all other tasks belonging to the same application (5-6), preventing packet exchanges with the task that will migrate. Next, the M_{PE} notifies the source PE to forward the task code and data to the target PE (7). When the task reaches the target PE (8), the OS recreates the task, but it remains blocked. The target PE (8) notifies the M_{PE} the successful migration through a `forward complete` packet (9). The final step is the application release through a `task resume` packet. This process creates an overhead in the application execution due to the task suspension to prevent losing packets during the migration.

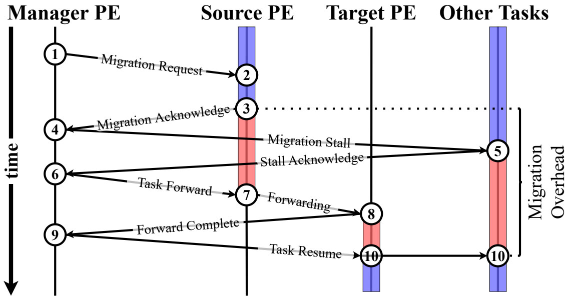

Fig. 4. Events diagram of task migration protocol implemented in the *Chronos-V* manycore.

Another actuation example supported by the platform is DVFS. As it is modeled at an abstract level (OVP), the change in frequency and voltage is simulated by changing the simulation *quantum* (detailed in Section IV). The energy computation considers different voltage-frequency pairs, depending on the processor state.

C. Application Layer

This layer corresponds to general-purpose applications. Each application contains a set of tasks. Those applications must be position-independent executables to allow multitasking without virtual memory management support. The applications, which are stored outside the MCSoC computing fabric, are deployed into the system through the App_{rep}. The App_{rep} communicates with the M_{PE}, that executes the task mapping protocol. After transmitting all tasks to the selected PEs, the M_{PE} releases the application execution. System calls make the interface between tasks and the OS layer, allowing, for example, inter-task communication.

Figure 5 presents the message passing protocol. Each message exchange requires two packets:

- `message request`, sent by the consumer task to the producer task, informing that the task is ready to receive a message. After sending this packet, the consumer task remains blocked, waiting for the message reception.
- `message delivery`, sent by the producer task after receiving the `message request`, contains the message payload. This packet is sent immediately after the `message request` if the message is already in the packet queue (Figure 5(a)). Otherwise, the OS registers the request (pending request), transmitting the message once the task generates it (Figure 5(b)).

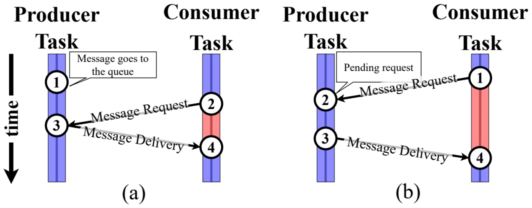

Fig. 5. Events diagram of message passing protocol implemented in the *Chronos-V* manycore.

When a task finishes its execution, the OS blocks its scheduling and waits for the consumption of packets stored in the packet queue. Next, the OS notifies the M_{PE} and releases the task memory. The M_{PE} upon receiving the "concluded" packet from all tasks of a given application, this application is considered finished. This notification is important to release data structures used by the decision module.

IV. Simulation Model

We adopted Open Virtual Platform (OVP) [16] to model and simulate the platform. The *Chronos-V* model preserves the temporal and spatial traffic distributions when compared to the physical implementation [17].

The platform simulation starts with a command line with five parameters: (*i*) simulation name, used to create a self-contained folder with the platform configuration, allowing reusing its configuration for different applications; (*ii, iii*) X and Y, corresponding to the system size; (*iv*) the management algorithm used in the simulation (if any); (*v*) the scenario file with the applications set to execute. It is possible to define for each application its injection time and if its runs periodically.

According to the provided parameters, the script responsible for creating the self-contained folder adjusts the platform source code and the hardware description. Next, every system component is compiled. The system components include the *NoC router*, the App_{rep}, the *NI*, and the virtual components that support the system simulation, like the *iterator* (discussed next).

The script creates a `.tcl` file with the system component instances and their interconnections that together forms the system *model*. In this file, each processor is defined and connected to every component that composes a PE, the NoC routers are connected to their respective neighbors, and the peripherals are connected to the NoC borders. The `.tcl` file is submitted the *iGen* [16] tool, that generates a *.c* file that models the system using the Open Platform (OP) API. Finally, it is compiled to generate the actual system model.

On top of that, there is the simulation manager, named *harness* [16], responsible for instantiating the system model and providing stimulus to it (it acts as a test bench). The simulation paradigm adopted by OVP is quantum-based with instruction accuracy. Definition 1 defines the *quantum* parameter.

Definition 1. Quantum – period that each processor executes. After all processors have executed their *quantum* period, the simulation advances the current time by a *quantum*, restarting at the first processor.

To compute the *quantum*, the number of cycles per instruction (CPI) is assumed equal to one. Equation 2 presents how to compute the *quantum* value.

$$quantum = Inst_Q / Core_{IPS} \tag{2}$$

Where: $Inst_Q$ amount of instructions executed during the *quantum* period; $Core_{IPS}$ number of instructions per second. As CPI=1, this parameter is equal to the processor frequency.

The synchronization between processors, i.e., the communication among them, occurs after all processors have executed their *quantum*. Thus, to avoid processors stalling while waiting for data, $Inst_Q$ must consider the trade-off between communication and computation. If $Inst_Q$ is too small, the simulation speed decreases due to the number of synchronization steps, while large $Inst_Q$ values stall processors waiting for data.

When all PEs have the same $Inst_Q$, we consider that the system is operating in a homogeneous frequency. As the *quantum* value is the same for all processors, it is possible to modify the processor frequency ($Core_{IPS}$) by modifying in the same proportion $Inst_Q$. For example, considering $Core_{IPS}$=1GHz and $Inst_Q$=10,000, the *quantum* is $1 \times 10^{-5} s$. If a processor needs to run at 500MHz, it is necessary to adjust its $Inst_Q$ to 5,000. This is the method to simulate the DVFS actuation.

Algorithm 1 presents a pseudo-code of the simulation loop defined in the *harness*. The simulation starts by releasing each processor (P_i) to simulate a *quantum* period (lines 3 to 5). The `simulate()` function returns $True$ if the simulated processor finishes its execution, otherwise $False$. The execution of those processes is parallelizable, and the `join()` (line 6) waits for every process completion. It is important to note that during the execution within the *quantum*, each processor only has access to data in its local memory and, consequently, packets that were delivered into the local memory by the NI.

The parallel behavior of components that do not rely purely on executed code, especially those depending on the communication through the NoC, was a challenge to model due to the absence of mechanisms that allowed the routers and processors to execute in parallel. To solve this problem we have developed a virtual peripheral called *iterator*, that is responsible for orchestrating routers after the execution of all processors.

The *iterator* behavior is expressed in the pseudo-code at lines 7 to 11. The execution calls the `iterate()` function for each router (R_j) in the system $model$ sequentially. Each "iteration" makes the router send one flit ahead, if available, to the next router in the packet route. When no flit can advance due to traffic or none is available to transmit, the function returns $True$. This loop is repeated until every router "iteration" returns $True$.

Algorithm 1: Simulation loop pseudo-code

```
Input: $model$, $Inst_Q[N]$, $quantum$
1  $SimulationTime \leftarrow 0$
2  repeat
3      foreach $P_i$ in $model$ do
4          $Finished_i$ = simulate($P_i$, $Inst_Q[i]$).start()
5      end
6      join()
7      repeat
8          foreach $R_j$ in $model$ do
9              $Finished_j$ = iterate($R_j$)
10         end
11     until $Finished_j = True \; \forall \; j$
12     $SimulationTime += quantum$
13 until $Finished_i = True \; \forall \; i$
```

Figure 6 exemplifies a 3x3 system with three packets being sent in the same *quantum*: 6 → 7; 0 → 7; 2 → 4. Each router is "iterated" sequentially, and, in the first turn the first flit of each packet advances one router (Fig. 6(a)). In the next turn, the R1 selects one of the two available packets to routing, following a Round Robin policy and forward the flit from 0 while blocking the flow from 2. Also, NI7 receives the packet from 6 and blocks the R7 local port. This way, in the next turn the flow 0 → 7 stops at the R7 south buffer and flow 2 → 4 stays blocked at the R1 east port (Fig. 6(b)). At this point, no flit can be forwarded and the simulation *quantum* ends by increasing the simulated time by one *quantum* (line 12). In the next *quantum*, the NI7 interrupt the processor to get a memory address to save the incoming packet, releasing the R7 local port.

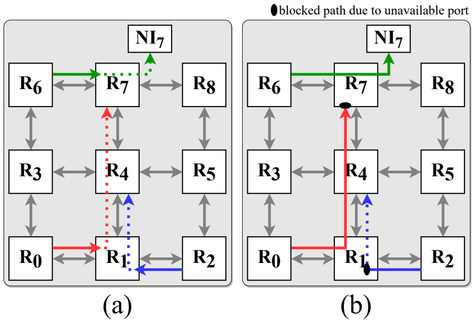

Fig. 6. Example of packets behavior after being sent in the same *quantum*.

Harness also provides assessing functions using the OP API, for example, the "instruction counter" module is a monitor (`opProcessorFetchMonitorAdd()`) that generates a callback in the *harness* every time the processor fetches a new instruction from memory. Inside the callback, the instruction is identified and categorized. After that, it updates the instruction counter amount in the PE memory (`opProcessorWrite()`), which is accessed periodically by the monitoring module to estimate the processor energy consumption.

V. Results

This Section evaluates the simulation speedup by comparing the proposed abstract model simulation against an RTL simulation. Next, we present thermal management techniques added to the FreeRTOS, to demonstrate that abstract models enable the development of management heuristics without the need to execute the RTL simulation. All simulations were executed in a workstation with a Xeon E-2246G@3.6 GHz processor (12 cores), 16GB DDR4 2666MHz dual-channel DRAM, with Ubuntu 20.04.

A. Simulation Effort

How fast is the many-core simulation using an abstract platform model? We compare the *Chronos-V* platform, running FreeRTOS, against a similar platform modeled at the RTL level (SystemC) – *Memphis*, running an in-house microkernel [18]. This experiment evaluates the simulation effort (min/s), i.e., the time required to simulate one second of the system model. The experimental included: (i) 9 system sizes, from 4 to 100 PEs; (ii) execution times ranging from 100ms to 1s (10 scenarios); (iii) system load equal to 50%. For example, a 64-PE system executes 32 tasks. The *Chronos-V* platform was evaluated for the 90 scenarios, while the *Memphis* simulated 46 scenarios due to the higher simulation time.

Figure 7 presents the average simulation effort to execute the proposed scenarios. The *Chronos-V* simulation varies from 2.36 min/s up to 92.46 min/s for system sizes ranging from 4 to 100 PEs, respectively. The Memphis requires 147.9 min/s up to 3,731.39 min/s for the same system sizes. The speedup decreased because in the quantum-based simulation all processors need to be simulated (multi-thread simulation, using up the 12 cores of the host machine) and synchronized at the end of this period (line 6 of Algorithm 1). In terms of absolute values, OVP required 1h30min while the RTL 62h13min to simulate a system with 100 PEs, running 50 tasks simultaneously.

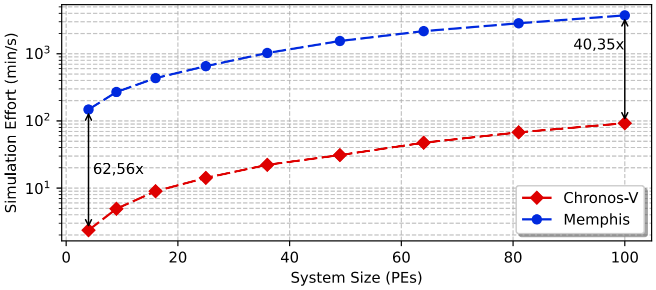

Fig. 7. The *Chronos-V* and Memphis simulation effort.

It is important to note that the platforms have different goals. Due to the clock-cycle accuracy, RTL simulation aims to validate the many-core hardware. The abstract model simulation seeks to validate the software development, both at the user and management levels. The *Chronos-V originality* is to offer to designers simulation speed, with an NoC behavior similar to the RTL model (paths taken and congestion effects). This feature allows designers to assess, for example: (i) the presence of hotspots and improve task mapping; (ii) study security-related techniques, such as detecting traffic anomalies that may indicate attacks; (iii) thermal management techniques.

B. Management Evaluation

This experiment aims to demonstrate the *Chronos-V* capability to provide an environment to test MCSoC management techniques. Experiments adopt an 8x8 MCSoC, running 36 tasks. Three scenarios are evaluated: (i) “no management”, i.e., tasks execute at the nominal voltage-frequency; (ii) chessboard patterning mapping [19] (PM), adopted to improve temperature distribution, but does not use thermal management; (iii) a Proportional, Integral, and Derivative Temperature Management (PIDTM) [20]. Figure 8 presents the system peak temperature and the system average temperature. The system temperature is calculated by a MatEx [21] module that uses the power estimations provided by monitoring (Equation 1).

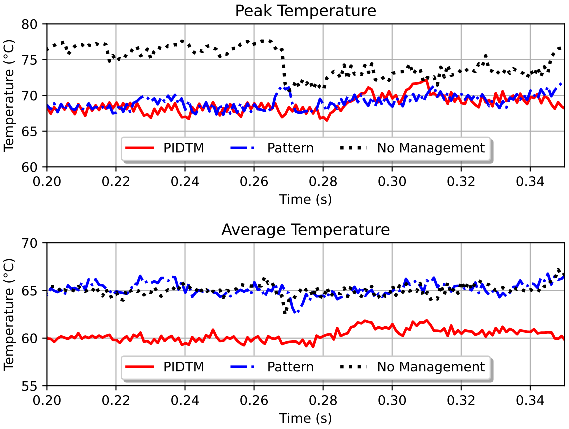

Fig. 8. Comparison between peak and average system temperature for three different management techniques.

As shown in the graphs, the absence of a management technique leads to higher peak temperatures and hotspots, which may affect reliability and lifetime [22]. PIDTM and PM present similar peak temperatures, showing that PM effectively reduces the peak temperature due to the temperature transfer reduction between neighbor PEs. However, a management technique relying on monitoring and actuation, like PIDTM, reduces the average temperature by 4.71° compared to PM. Our objective here is not to detail the management techniques but to demonstrate that the *Chronos-V* platform offers the possibility to develop and test software for large systems quickly.

VI. Conclusion

This paper presented the *Chronos-V* platform suitable for the research in management techniques and the development of parallel applications for MCSoCs with dozes of PEs. The hardware model adopted a quantum-based simulator to speed up the simulation, being two orders of magnitude faster when compared to clock cycle-accurate models. The software adopted an open-source OS, freeRTOS, extended with APIs to enable PE-PE communication and system management. We demonstrate the capability to evaluate the system management using thermal heuristics, added to the freeRTOS as an API. The proposed high-level model help designers leverage the research in the field of NoC-based many-core systems.

This platform paves the way for future works that include: (i) inclusion of reliability monitoring metrics into the system; (ii) perform studies of reliability-aware management techniques.

Acknowledgment

The authors would like to thank Imperas Software and Open Virtual Platforms for their support and access to their models and simulator. This work was financed in part by CAPES (Finance Code 001), and CNPq (grant 309605/2020-2).

References

[1] D. R. Ditzel and Esperanto team, “Accelerating ML Recommendation with over a Thousand RISC-V/Tensor Processors on Esperanto’s ET-SoC-1 Chip,” in *HCS*, 2021, pp. 1–23.

[2] M. Hassan, “Heterogeneous MPSoCs for Mixed-Criticality Systems: Challenges and Opportunities,” *IEEE Design & Test*, vol. 35, no. 4, pp. 47–55, 2018.

[3] P. Rahimi, A. K. Singh, X. Wang, and A. Prakash, “Trends and Challenges in Ensuring Security for Low-Power and High-Performance Embedded SoCs,” in *MCSoC*, 2021, pp. 226–233.

[4] R. Leupers, L. Eeckhout, G. Martin, F. Schirrmeister, N. P. Topham, and X. Chen, “Virtual Manycore platforms: Moving towards 100+ processor cores,” in *DATE*, 2011, pp. 715–720.

[5] R. Lemaire, S. Thuries, and F. Heitzmann, “A flexible modeling environment for a NoC-based multicore architecture,” in *HLDVT*, 2012, pp. 140–147.

[6] C. Helmstetter, S. Basset, R. Lemaire, F. Clermidy, P. Vivet, M. Langevin, C. Pilkington, P. G. Paulin, and D. Fuin, “A dynamic stream link for efficient data flow control in NoC based heterogeneous MPSoC,” in *ASP-DAC*, 2013, pp. 41–46.

[7] L. Duenha, M. Guedes, H. Almeida, M. Boy, and R. Azevedo, “MPSoCBench: A toolset for MPSoC system level evaluation,” in *SAMOS*, 2014, pp. 164–171.

[8] G. A. Madalozzo, M. Mandelli, L. Ost, and F. G. Moraes, “A platform-based design framework to boost many-core software development,” in *ICECS*, 2015, pp. 320–323.

[9] R. Cataldo, R. Fernandes, K. J. M. Martin, J. Sepúlveda, A. A. Susin, C. A. M. Marcon, and J. Diguet, “Subutai: distributed synchronization primitives in NoC interfaces for legacy parallel-applications,” in *DAC*, 2018, pp. 83:1–83:6.

[10] G. L. Lima, N. de Farias Traversi, D. F. Adamatti, G. P. Dimuro, C. Meinhardt, E. W. Brião, and O. M. Mendizabal, “Exploring MAS to a High Level Abstration NoC Simulation Environment,” in *ICECS*, 2018, pp. 365–368.

[11] Y. M. Qureshi, W. A. Simon, M. Zapater, D. Atienza, and K. Olcoz, “Gem5-X: A Gem5-Based System Level Simulation Framework to Optimize Many-Core Platforms,” in *SpringSimu*, 2019, pp. 1–12.

[12] M. F. Reza, D. Zhao, and M. A. Bayoumi, “Power- Thermal Aware Balanced Task-Resource Co-Allocation in Heterogeneous Many CPU-GPU Cores NoC in Dark Silicon Era,” in *SOCC*, 2018, pp. 260–265.

[13] G. F. Junior., J. Rodrigues, L. F. Carvalho, J. Al-Muhtadi, and M. L. P. Jr, “A comprehensive survey on network anomaly detection,” *Telecommun. Syst*, vol. 70, no. 3, pp. 447–489, 2019.

[14] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal, “A generalized software framework for accurate and efficient management of performance goals,” in *EMSOFT*, 2013, pp. 1–10.

[15] P. Chakraborty, B. N. Swamy, and P. R. Panda, “Manycore processor architectures,” in *Many-Core Computing: Hardware and Software*, B. M. Al-Hashimi and G. V. Merrett, Eds. The Institution of Engineering and Technology, 2019, ch. 17, pp. 419–448.

[16] “Open Virtual Platforms - the source of Fast Processor Models & Platforms,” 2022. [Online]. Available: www.ovpworld.org

[17] G. Lopes, I. I. Weber, C. A. M. Marcon, and F. G. Moraes, “Chronos: An Abstract NoC-based Manycore with Preserved Temporal and Spatial Traffic Distribution,” in *LASCAS*, 2021, pp. 1–4.

[18] M. Ruaro, L. L. Caimi, V. Fochi, and F. G. Moraes, “Memphis: a Framework for Heterogeneous many-core SoCs Generation and Validation,” *Design Automation for Embedded Systems*, vol. 23, no. 3-4, pp. 103–122, 2019.

[19] X. Wang, A. K. Singh, and S. Wen, “Exploiting dark cores for performance optimization via patterning for many-core chips in the dark silicon era,” in *NOCS*, 2018, pp. 1–8.

[20] A. L. da Silva, A. Martins, , and F. G. Moraes, “Mapping and Migration Strategies for Thermal Management in Many-Core Systems,” in *SBCCI*, 2020, pp. 1–6.

[21] S. Pagani, H. Khdr, W. Munawar, J. Chen, M. Shafique, M. Li, and J. Henkel, “MatEx: Efficient transient and peak temperature computation for compact thermal models,” in *DATE*, 2015, pp. 1515–1520.

[22] M.-H. Haghbayan, A. Miele, Z. Zouv, H. Tenhunen, and J. Plosila, “Thermal-cycling-aware dynamic reliability management in many-core system-on-chip,” in *DATE*. IEEE, 2020, pp. 1229–1234.