RESEARCH ARTICLE OPEN ACCESS

Building A Model for Multi-Core Architecture Using Systemc

Mouhsen Ibrahim [1], Mohammad Sobeih [2]
Department of Operating Systems and Computer Networks [1], Tishreen University, Latakia
Department of Operating Systems and Computer Networks [2], Tishreen University, Latakia

ABSTRACT
Modern parallel programming models help programmers to easily convert serial programs into parallel ones, by creating a set of tasks and their dependencies to define execution order of them. To increase the efficiency of running these multi task programs, Hardware-Accelerators were created to take the lift of the Operating System kernel from scheduling tasks to worker processing cores, to be able to create these accelerators, we need to have a model of a multi-core architecture where we can define a big number of cores, and simulate executing several tasks on these cores with help from the hardware accelerators.
In this paper, we present a model of a multi-core architecture using SystemC, which can be used to evaluate the Task-Management Hardware-Accelerators using simulation and tune their parameters accordingly.
Keywords :— Multi-core architecture, SystemC, Parallel Programming

I. INTRODUCTION

To effectively use available cores in a multi-core architecture several task-based programming models have been proposed to enable programmers to write parallel applications that generate multiple tasks to be executed on worker cores.

StarSs is one of these programming models, it enables programmers to write annotations around the code which will be executed in parallel, however relying on software to schedule these tasks to worker cores will create a bottleneck in performance as the software will not be able to co-op with the ever increasing number of tasks and cores.

Nexus [1] was created for the StarSs parallel programming model and was enhanced later [2]–[4]

Also we have TaskGenX [5] which focuses mainly on accelerating the process of adding new tasks to the dependency graph in order to be sent to be executed on ready worker cores once all their dependencies are fulfilled.

Anyway to be able to design and test Nexus, we need a model of a multi-core architecture so we can run it inside it, a model was created using SystemC and evaluated [6] so we decided to use SystemC to create our own model that can be easily used to evaluate and test Nexus.

The previous model [6] lacks an advanced memory system and also cannot be used effectively to run tasks with arbitrary number of inputs, outputs and execution cycles, so here we decided to put these two consideration in top priority and create a model that has advanced memory architecture and can be easily used to run tasks with many inputs, outputs and different execution cycles.

Section 2 shows some related work, section 3 gives an introduction to SystemC and its architecture, section 4 describes the multi-core architecture modelled here using SystemC, section 5 describes the SystemC modules used to create the architecture here and their interfaces, section 6 shows several test benches for the SystemC modules and evaluates their efficiency in executing multiple independent tasks in the architecture, section 7 provides concluding results.

II. RELATED WORK

HORNET [7] is a parallel, cycle-level multicore simulator based on an wormhole router network-on-chip architecture. The parallel simulation engine offers cycle-accurate as well as periodic synchronization, while preserving functional accuracy. This permits tradeoffs between perfect timing accuracy and high speed with very good accuracy. Most hardware parameters are configurable, including memory hierarchy, interconnect geometry, bandwidth, and crossbar dimensions. A highly parameterized table-based NoC design allows a variety of routing and virtual channel allocation algorithms out of the box.

The COTSon team at HP labs describes [8] a methodology to efficiently simulate a Chip Multiprocessor of hundreds or thousands of cores using a full system simulator. They consider an idealized architecture with a perfect memory hierarchy, i.e., without any interconnect, caches nor distribution of memory banks. Their experiments show that the simulator can scale up to 1024 cores with an average simulation speed overhead of only 30% with respect to the single-core simulation.

As an example of heterogeneous multi-core architecture we find ARM big.LITTLE which combines in order execution cores with low power consumption and high performance out of order execution cores [9], to achieve the best utilization there was a model to evaluate performance and power consumption [10] using gem5 and McPAT simulators.

Also another model of heterogeneous cores was created to be used in edge computing, this architecture was evaluated using FPGA to get better unbiased results [11] also a parallel programming model was created for this architecture.

In [6] a SystemC model is presented to prove the applicability of SystemC to simulate many-core architecture, it models a system of P cores then simulate the execution of matrix multiplication. The simulation of the model allows analyzing the results regarding the number of transfers and the number of clock cycles required to complete each transaction. A theoretical model of the algorithm execution time is used to evaluate the precision of the system-level simulator. Simulation results indicate that the simulation models are quite precise and simulation times of a few minutes are possible for systems with a hundred of cores.

Our work presents an improved version of the last SystemC simulator, it uses multiple modules to represent each component of a multi-core architecture such as an execution unit, a core unit, memory access unit and a board unit for simulating a different number of cores. Also we presented a complete memory architecture using NUMA, this is considered essential in any multi-core architecture because the memory access becomes a performance bottle neck with increasing number of cores so we had to use NUMA to avoid this.

We opted to use SystemC because it makes it easy to simulate hardware and software component together where we can create a model for the multi-core architecture and at the same time create another one for the Hardware-Accelerator we want to test so integrating the Hardware-Accelerator model is very easy with the multi-core architecture as they are both written in SystemC.

III. SYSTEMC INTRODUCTION

SystemC is a system design language that has evolved in response to a pervasive need for a language that improves overall productivity for designers of electronic systems [6], it enables designers to create models of their systems, evaluate and verify these models in software before they are synthesized to create an even more accurate model of the systems.

SystemC offers flexibility and the ability to favor performance or accuracy above each other, each SystemC process can be allowed to take as much as clock cycles we want but the real hardware can do X amount of work in Y clock cycles, this is the task of the synthesizer to determine the number of clock cycles required to do a task, however in SystemC we are given the ability to change this number as we want to favor performance over accuracy if we want.

SystemC consists of a collection of C++ classes which implement extra data types for hardware communication and has classes which are used to model hardware units with inputs, outputs and processing methods inside them, the SystemC scheduler handles the execution of all methods as separate threads running at the same time, however SystemC does not use operating system threads to do its task and is considered single thread programs.

The main building blocks of SystemC are modules which are declared using the SC_MODULE macro, a module has a group of input and output ports used to send and receive data to/from other modules through communication channels.

Also, modules have processes which do the actual job for each module these processes might read inputs from input ports, process these inputs and write outputs to output ports, there are three types of processes: Methods, Threads and Clocked threads.

Methods are functions that get executed once each time its trigger is activated (A change in input value).

Threads these are executed all the time and are activated using a specific trigger, the thread continues execution until it reaches a wait call, where it waits until the trigger is active again.

Clocked Threads are exactly just like threads but they can only be activated using clock cycles which makes them the best in simulating hardware components.

IV. MULT-CORE ARCHITECTURE

Here we will explain the main components of the architecture created in this paper.

We created two models one that uses shared memory space called UMA (Uniform Memory Access) and the other segments memory space among all available cores called NUMA (Non-Uniform Memory Access), first we will explain common components between the two

There are 3 common components:

1. Execution unit: each core consists of one execution unit which is used to execute tasks, we did not include any extra details in here because we only need to model the task execution time, we do not care what kind of instructions a task is executing since we already have task execution time in number CPU cycles.

2. The core unit: This unit describes a single CPU core which has a buffer for buffering tasks before they are sent to execution unit, it receives tasks from the System Board or the Nexus hardware accelerator, sends them to be executed in the execution unit and finally notifies whatever sent them of their completion.

3. The Board unit: This unit describes a System Board which contains a predefined number of cores, it receives tasks from the OS and send them to an idle core, currently the board unit does not contain a hardware accelerator for dependency management and resolution, it will be added later.

That is all for common components now we move to each one of the models:

The first Model UMA

This model uses a shared memory space between all cores, access to main memory is coordinated using a memory controller, here we describe this new unit

The Memory Access Unit this unit is used to organize access to the System Memory, it has a number of inputs and outputs that is equal to the number of cores, when a core wants to read a memory location it signals this unit using its own input, the memory unit reads each input in round-robin algorithm once it finds an active one it signals the right core that it can use memory now using its own output after that the core can access memory and once finished this unit continues to check other cores' inputs, memory read/write is modelled using a wait for a few cycles, this parameter can be changed accordingly.

Figure 1 shows the above described model

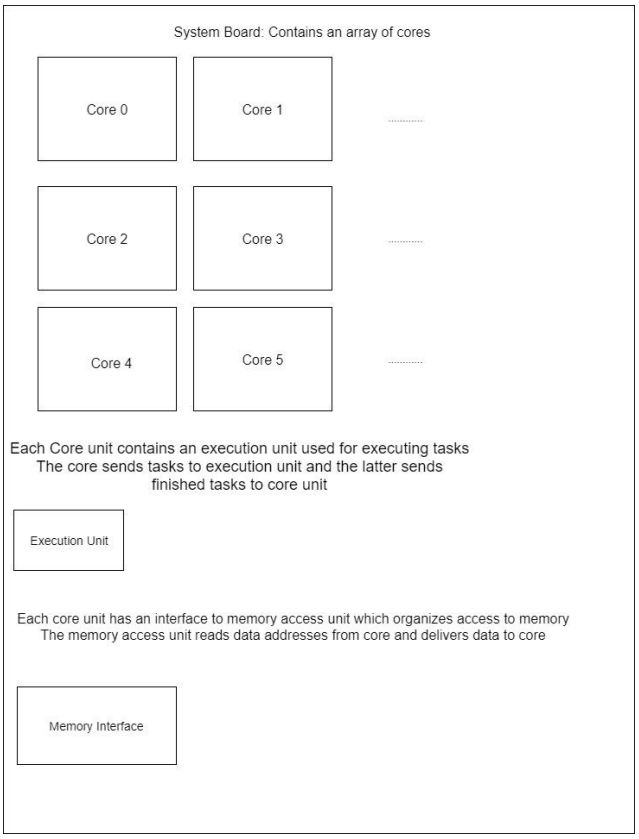

Figure 1 first model of multi-core architecture

The second model NUMA

In this model memory is partitioned to segments and each segment is linked directly with a number of cores, this consists what is called a NUMA node, cores within a single node can access its own memory segment without any competition from other cores, when the core needs to access a memory address from another segment the memory bus is used, also there is a cache memory for each core and another cache memory for each NUMA node.

The following figure shows the NUMA architecture

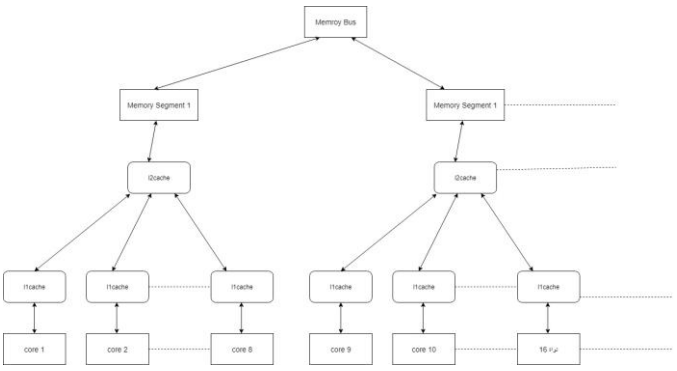

Figure 2 second model of multi-core architecture

The architecture described here is hierarchical, and has four components:

1. Level 1 cache memory l1cache: This represents the first memory level, when any core wants to read a memory location it asks the address from l1cache first, if it was found it is returned immediately to the core in only two clock cycles and if not, it is asked from l2cache.
2. Level 2 cache memory l2cache: It represents the second level of memory, it is shared between the cores in each NUMA node, access to this memory is coordinated the same way it was in UMA model, we chose to use 8 cores for each NUMA node because practical results show that using 8 cores does not cause much memory contention.
3. Memory Segment: It represents the third and last level of memory hierarchy, this segment contains a portion of address space between min_addr and max_addr, it can be accessed directly from l2cache, if the address was not available here then it is sent to the memory bus which brings the data from the right memory segment.
4. Memory Bus: This is not part of memory hierarchy, it just links memory segments together so we can fetch any memory address form any NUMA node, it coordinates access to each memory segment from other memory segments.

V. SYSTEMC MODEL OF THE ARCHITECTURE

Two models were built using SystemC for each one of the previous multi-core architecture models, these two models have 3 modules in common:

1- Execute Module: This module represents the Task Execution Unit in a single processing core, it reads data from the core unit using ready/valid protocol, when it is ready to receive data it enables the rdy output and when the core unit send data it enables the data in valid pin then the execute module reads data and once finished it enables the data in finished pin to signal end of reading data to the core module, all other modules follow the same protocol for reading and writing data.

2- Core Module: This module represents a single processing core, it has a buffer for incoming tasks, this buffer has a size of 2 by default which means it buffers only one task while executing another one, it receives tasks from the system board and sends them to execution unit, it stops reading new tasks once buffer is full and continues to read data when the buffer is not full. This module has a slightly different design in the previous two models, in UMA model it accesses memory using memory access module and has to wait until it is given right to access memory, but in the NUMA model it can access l1cache directly which can return data immediately of it is available in local cache or ask for the data according to the memory hierarchy.

3- System Module: In this module we have a group of cores that can be changed as needed, all received tasks from the OS are buffered before they are sent to an idle core or to a hardware accelerator such as Nexus that can send tasks to idle cores later. In this module we have a single memory access module in the UMA model but in NUMA model we have a hierarchy of memory modules that represent l1cache, l2cache, memory segment and memory bus.

Now we will describe the different modules in each model

UMA model

Here we have memory access module which organizes access to main memory from all cores, it only allows one core at a time to access memory which increases waiting time when more cores are added, it uses round robin algorithm to give access so in each round all cores are allowed to access if they need.

NUMA Model

In this model we have these additional modules:

1- Level 1 cache module l1cache: Every core has its own Level 1 cache memory, in this memory the most used memory addresses are stored for very fast access, the core can access this cache directly without competition from any other core and it has a well defined interface to send memory addresses and read/write data to them, if the memory address is not available here it is automatically requested from Level 2 cache.

2- Level 2 cache module l2cache: Every NUMA node has a level 2 cache memory that is bigger that l1cache, this memory is shared between all cores in the NUMA node (The number is 8 by default). These cores compete to access it and if the address was not found here it is asked from the memory segment for this NUMA node.

3- Memory segment: Evert NUMA node has its own memory segment that holds part of the global address space, cores in the NUMA node compete with each other only to access this segment, if a memory address is not found here the request is finally sent to the memory bus.

4- Memory Bus: This bus is connected with all memory segment in the board, it is used to access any segment to get the right address data from it, only memory segments for each NUMA node compete with each other to access the bus.

VI. RESULTS

We did experiments using both models to compare them and determine the best one of them to be used later when evaluating the previously mentioned hardware accelerators, we used 3 sets of independent tasks as follows:

1. A set that consists of 10,000 tasks with one input per task, all inputs exist in a single memory segment.
2. A set that consists of 10,000 tasks with each task having 4 to 6 inputs, all inputs exist in the same memory segment.
3. A set of 10,000 tasks with one input per task but these inputs are distributed among all memory segments.

The models can be used to simulate a larger number of tasks but simulation time would increase also the results are very similar.

The following table shows the parameters for the two models

Table 1 Models' parameters

Parameter	Value	Model
Cores Number	10	Both
Buffering Depth	2	Both
Memory delay	10	UMA
Number of buffered tasks in board	100	Both
L1cache delay	2	NUMA
L2cache delay	4	NUMA
Memory segment delay	6	NUMA
Memory Bus delay	4	NUMA

The values for Memory delay in NUMA were used to reflect increasing delay as we go higher in memory hierarchy, we can get better values using a synthesizer. Other values can be changed as we need especially cores number which is the most important parameter.

First, we will show results for the first task set using both models.

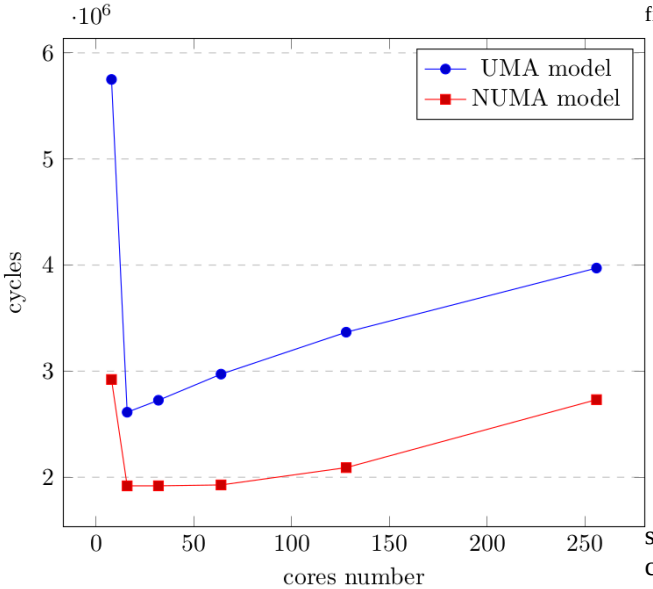

Figure 1 total execution cycles using first set of tasks

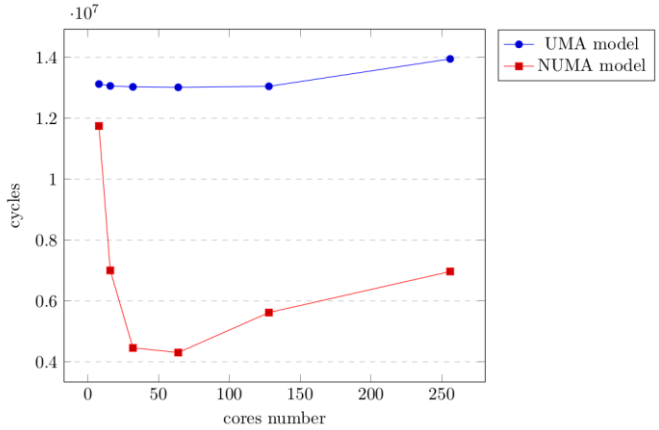

Figure 2 memory access cycles using first set of tasks

The previous results show a slight enhancement in execution cycles using the NUMA model and a big enhancement in memory access time, however with increasing number of cores the total cycles increase again which is not acceptable.

We will show results using second set of tasks with more inputs than the first one.

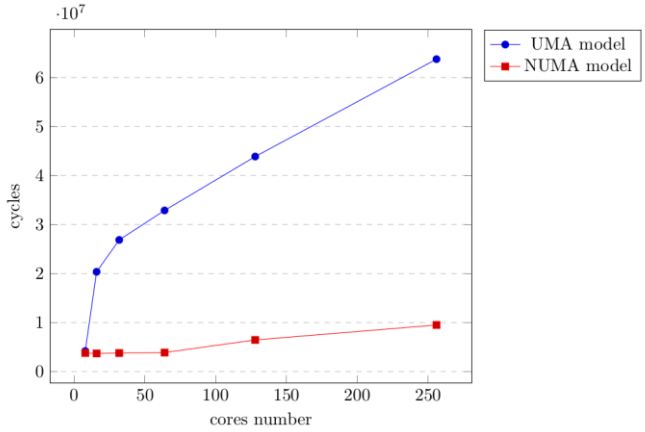

figure 3 total execution cycles using second set of tasks

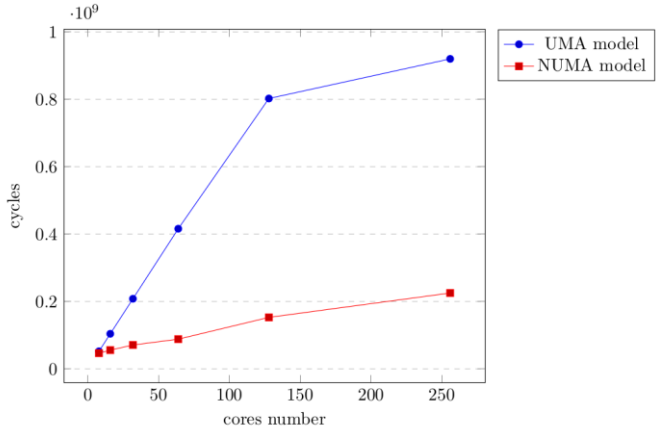

Figure 4 total memory cycles using second set of tasks

Here the results are much like the previous ones, NUMA shows some enhancements compared to UMA but when more cores are used, we still have the problem of more execution cycles to get the tasks done.

The last set of tasks has one input per task but the inputs are distributed among all memory segments not just one segment as in the first set of tasks, here we also used an enhanced scheduling algorithm to decrease the total cycles needed for execution, the following figures show our results.

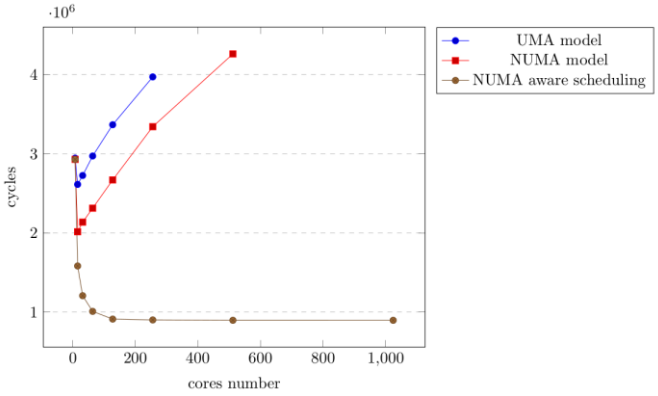

Figure 5 total execution cycles using third set of tasks

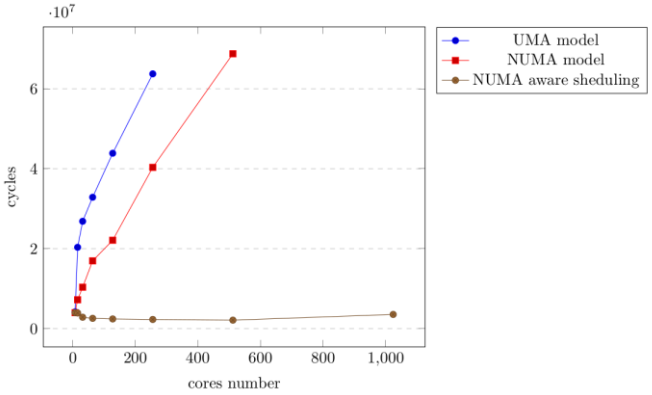

Figure 6 total memory cycles using third set of tasks

Here in these figures we see a significant improvement using the new NUMA aware scheduling algorithm, this algorithm schedules tasks to the cores which are near its input arguments which would decrease the time needed to fetch task inputs and also decreases the total execution time, with this algorithm we can increase the number of cores as much as we can without negatively affecting execution time.

VII. CONCLUSIONS

Here we offered two models for multi-core architecture, the first one uses UMA architecture and shows bad scalability when more cores are added, the second one uses NUMA architecture and offers very good scalability when NUMA aware scheduling algorithm is used, these two models can be easily used to evaluate hardware accelerators with many task sets and different dependency patterns and execution cycles with a variable number of inputs and outputs.

ACKNOWLEDGMENT

The authors wish to acknowledge the Faculty of Information Engineering in Tishreen university for their support of this research.

REFERENCES

[1] C. Meenderinck and B. Juurlink, “A case for hardware task management support for the StarSS programming model,” *Proc. - 13th Euromicro Conf. Digit. Syst. Des. Archit. Methods Tools, DSD 2010*, pp. 347–354, 2010.

[2] T. Dallou, B. Juurlink, and C. Meenderinck, “Improving the scalability and capabilities of the nexus hardware task management system,” *1st {International} {Workshop} {Future} {Architectural} {Support} {Prallel} {Programming}*, vol. 5, pp. 442–445, 2011.

[3] T. Dallou, A. Elhossini, and B. Juurlink, “FPGA-Based Prototype of Nexus++ Task Manager,” *6th Work. Many-Task Comput. Clouds, Grids, Supercomput.*, 2013.

[4] T. Dallou, N. Engelhardt, A. Elhossini, and B. Juurlink, “Nexus#: A Distributed Hardware Task Manager for Task-Based Programming Models,” *Proc. - 2015 IEEE 29th Int. Parallel Distrib. Process. Symp. IPDPS 2015*, pp. 1129–1138, 2015.

[5] K. Chronaki, M. Casas, M. Moreto, J. Bosch, and R. M. Badia, “TaskGenX: A hardware-software proposal for accelerating task parallelism,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 10876 LNCS, pp. 389–409, 2018.

[6] A. R. Silva, W. José, H. Neto, and M. Véstias, “Modeling and Simulation of a Many-core Architecture Using SystemC,” *Procedia Technol.*, vol. 17, pp. 146–153, 2014.

[7] M. Lis *et al.*, “Scalable, accurate multicore simulation in the 1000-core era,” *ISPASS 2011 - IEEE Int. Symp. Perform. Anal. Syst. Softw.*, pp. 175–185, 2011.

[8] M. Monchiero, J. H. Ahn, A. Falcón, D. Ortega, and P. Faraboschi, “How to simulate 1000 cores,” *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 2, p. 10, 2009.

[9] a R. M. Cortex-a and P. Greenhalgh, “big . LITTLE Processing with,” no. September 2011, pp. 1–8, 2012.

[10] A. Butko *et al.*, “Full-System Simulation of big . LITTLE Multicore Architecture for Performance and Energy Exploration To cite this version : HAL Id : lirmm-01418745 Full-System Simulation of big . LITTLE Multicore Architecture for Performance and Energy Exploration,” 2016.

[11] A. Gamatie, G. Devic, G. Sassatelli, S. Bernabovi, P. Naudin, and M. Chapman, “Towards Energy-Efficient Heterogeneous Multicore Architectures for Edge Computing,” *IEEE Access*, vol. 7, pp. 49474–49491, 2019.