System-Level Prototyping Framework for Heterogeneous Multi-Core Architecture applied to Biological Sequence Analysis

Nuno Roma
Instituto Superior Técnico / INESC-ID
Rua Alves Redol, 9 - 1000-029 Lisboa, Portugal
Email: Nuno.Roma@inesc-id.pt

Pedro Magalhães
Instituto Superior Técnico / INESC-ID
Rua Alves Redol, 9 - 1000-029 Lisboa, Portugal

Abstract—**An event-driven prototyping and simulation framework to support the design and early development stages of an heterogeneous multi-core processing architecture is presented in this manuscript. The main focus of this parallel structure is to efficiently execute a set of widely used bio-informatics algorithms for DNA sequences alignment and processing. The conceived framework was entirely developed using the SystemC description language and allows a full parametrization of the prototyped multi-core architecture, such as the amount of computing nodes, the effective alignment throughput of each node, and the capacity and access time of the memory devices. The presented experimental results demonstrate that the conceived framework provides the system designer with a very useful preliminary evaluation of the prototyped architecture. In particular, the included evaluation demonstrates the relation between the number and performance of the computing nodes and the resulting alignment performance gain (speedup), as well as the inherent bus contention losses in the shared resources (bus and shared memory).**

I. Introduction

With the latest developments in computer architecture, modern computational systems are often composed by multiple processors, memories and dedicated hardware structures (accelerators) integrated either in embedded systems or even in System on Chip (SoC) devices. However, as the complexity of these systems increases and the design time is shortened due to market demands, it is extremely important to thoroughly prototype these systems before their design achieves the manufacturing process. Furthermore, several difficulties usually arise in the pre-design stage, when the system's characteristics need to be specified before it is actually implemented. Not rarely, the system specification may be incomplete and inconsistent, and often there is not any feasible way to verify the correctness of such specification [1], [2].

All these facts have pushed the usage of hardware description languages (e.g. VHDL and Verilog) and of system description languages (e.g. SystemC) [3] a lot further, in order to make them applicable to system design and behavioral simulation. With such development frameworks, the designers are immediately able to benefit from some of the advantages traditionally offered by programming languages. Firstly, they provide the control and data abstraction layers that are necessary to develop compact and efficient system descriptions, allowing to divide the project into pieces and to separate them in a logical way, in order to allow each member of the development team to concentrate on each module design. Secondly, they allow to effectively and accurately simulate and prototype complex systems containing both hardware and software components. Finally, they usually provide the designers with the same developing environment that it is usually offered by state of the art programming frameworks and tools, thus significantly improving the efficiency of the design stage [3].

In particular, the SystemC description language was built based on standard C++, by extending the language with specific class libraries, and by providing an event-driven simulation kernel in C++. With this environment, the designer is able to simulate concurrent processes using plain C++ syntax. Moreover, SystemC processes can communicate in a simulated real-time environment, by using signals of all data types offered by C++ and some additional ones offered by the SystemC library, as well as those that are user defined. In certain aspects, SystemC deliberately mimics VHDL and Verilog hardware description languages, but is more aptly described as a system-level modeling language.

By taking advantage of these computer-aided design tools, several processing structures have already been extensively simulated and prototyped by using this language specification [4], [5], allowing a preliminary identification and an early correction of design flaws in the prototyped designs. In particular, it was already demonstrated the viability of using this design approach to accurately prototype complex interconnection structures to sustain the implementation of high-performance parallel processing architectures [6].

In this scope, an integrated simulation and prototyping framework targeting the design of an optimized heterogeneous multi-core architecture applied to biological sequence analysis is presented in this paper. When compared with current general purpose Central Processing Units (CPUs), this dedicated and heterogeneous parallel architecture will allow to significantly accelerate the execution of a broad set of highly demanded algorithms for biological and genome analysis, such as Deoxyribonucleic acid (DNA) alignment and mapping.

II. Prototyping and Simulation Framework

The design and definition of a dedicated parallel platform usually comprises several distinct and correlated phases:

Phase 1- Initial specification of the parallel processing structure, comprising the enumeration of the several elements (computing nodes, memory devices, buses, etc.) and their interconnection topology;

Phase 2- Preliminary evaluation of the requirements of the several elements and identification of possible contention points, targeting a given scalability potential;

Phase 3- Definition of the internal architecture of the several processing nodes, memories and bus infrastructures, according to the requisites evaluated in *Phase 2*;

Phase 4- Formal parallel programming of the targeted set of algorithms in the conceived parallel structure;

Phase 5- Final evaluation of the conceived system.

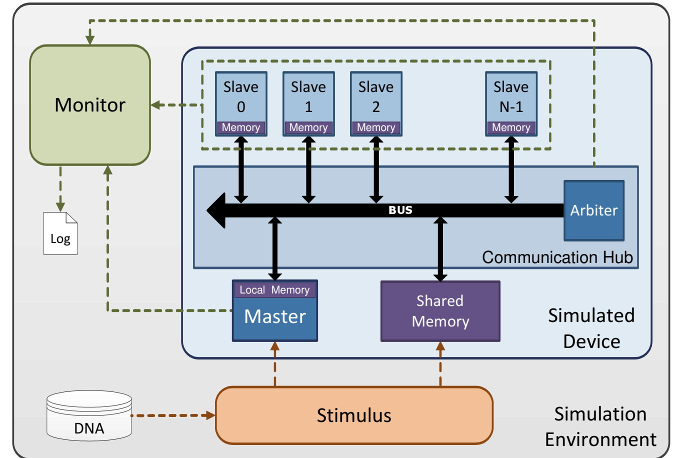

Fig. 1. Simulation framework block diagram.

Accordingly, the main goal of the presented framework is to provide an integrated prototyping and simulation environment to support the evaluation procedures corresponding to *Phase 2* of this design cycle, targeting a specific heterogeneous multi-core processing structure for bio-informatics applications. In particular, considering the co-existence of several processing nodes that will have to communicate and cooperate with each other, the main aim of this simulator is to prototype and evaluate a broad set of characteristics in a very early development stage. This will influence the final processor design, in order to achieve the desired performance and offered scalability potential in what concerns the number of instantiated processing nodes. Among this set of evaluated characteristics, it is worth mentioning:

- Number of computing nodes;
- Minimum processing throughput to be offered by the several computing nodes;
- Data-transfer throughput of the shared memory device;
- Arbitration policy to access the shared resources.

With the gathered information, it will be possible to prosecute with the formal definition of the several elements of the multi-core structure, namely, the selection of the internal architecture of the computational nodes, the definition of the communication structures, the selection of the most appropriate memory devices, etc.

Due to the higher abstraction level that is considered in this developing stage, the implementation of such a prototyping framework follows a pure transactional level approach, where the conducted simulations provide feedback about the main events that concurrently take place inside the processing units, memories, buses, etc. Fig. 1 illustrates the several components that integrate the developed system. Two co-existing parts can be identified. The first one is the *simulation environment*, which contains the simulated device and the modules that are responsible to support the programmer during and at the end of the simulation (monitor and stimulus). The second one is the *simulated device*, which incorporates the actual processing units, memory devices and communication infrastructure. All of them are meant to be configured and simulated.

A. Simulation Environment

The simulation environment is composed by the *stimulus* and the *monitor* modules. These will be used by the programmer in order to: *i)* provide the data to be processed by the *simulated device*, and *ii)* monitor its activity. The simulated device can be seen as an independent module, that is inserted into the simulation framework for the specific purpose of testing and evaluation.

Stimulus Module - The stimulus module is the starter of the system. When the operation of the simulator is initiated, the stimulus module is the only one that is awake, except for the shared memory that is always active and only reacts to write and read operations. Its job is focused on providing the system with test data.

Monitor Module - The monitor is in charge of tracing the status of the simulated device. The computing devices/nodes involved in this operation are the *master* and *slave* nodes. Every time a module changes its status, an event is sent to the monitor and recorded in a log file. The status of each module changes depending on the operation that is going through. In particular, the simulated processing nodes can be in one of the following five states:

- Yield status;
- Read status;
- Write status;
- Executing status;
- Communication waiting status.

B. Simulated Device

The simulated device is composed of one master processor, a variable number of slave processors, one shared memory, one communication hub to interconnect all the modules, and a mailbox message exchange system connecting all the nodes. In the following, it will be presented a brief description of each device.

Master - The master node is the manager of the simulated device. It sends commands to the slave nodes and they answer back with the corresponding results. This way, all the data that is produced in the device is a consequence of a set of commands issued by the master node.

In order to improve the efficiency of the communication infrastructure, the master node is always aware of the commands that were sent to each slave. When a command is sent to a slave, this one is considered as busy until an answer message arrives, with the results of the ordered command. Hence, the slaves will receive new commands only when they are not busy and as long as there is still some data to be processed.

Slave - The slave nodes are in charge of most of the computational demanding job carried out by this prototyped multi-core system. Nevertheless, their job is characterized by a passive position, since they only start working after receiving commands from the master node.

Memory - Besides the scratchpad local memory that is available in each of the slaves and master nodes, the simulated device also has a built-in shared global memory, which is used for data transfer between the master and the slaves. This memory can be regarded as a DNA data repository, where the sequences are stored before being transferred to the slave devices' local memory, where they will be subsequently processed.

Communication Hub - The communication hub is the entity that makes data transfers possible within the simulated device. It is the bridge for data and command messages traffic between the master, the slaves and the shared memory. Master and slave nodes use it as an interface to access the shared memory and to send messages between each other. Its internal implementation (single/multiple-bus paths, crossbar switch, Network on Chip (NoC), etc.) can be adjusted to the target application needs.

Mailboxes - While the data under processing is transferred from one processor to another via the shared memory, the command messages are sent and received through an individual mailbox that is attached to each processing node. Such messages usually represent commands that the master node sends to the slaves. Therefore, these messages tend to be short and contain only important parameters that are used by the receiving processor to execute a specific operation. This way, command messages often transport address and size parameters, so that the receiving processor can subsequently read the intended data from the shared memory and execute the related tasks from the implemented algorithm.

A final note is worth mentioning concerning the addressing space that is seen by each processing node of this multi-core system. Every node (master or slave) is able to address the system's shared memory, its own internal local memory and its own mailbox. The shared memory, with an exclusive access policy defined by an arbiter, is mapped within the first half of the addressing space and is usually used to exchange data between nodes. Consequently, these addresses are seen by all the computing nodes of the system. The local scratchpad memory of each device, used during the execution of the intended processing algorithm, is addressed starting from the second half of the addressing map. Finally, the last positions of the addressing space are used to access the local mailbox, required to transfer command messages. Fig. 2 shows the addressing space that is made available to each node. The dimension of the addressing space that is considered by this prototyping framework is highly configurable, depending on the requisites of the considered simulation and evaluation.

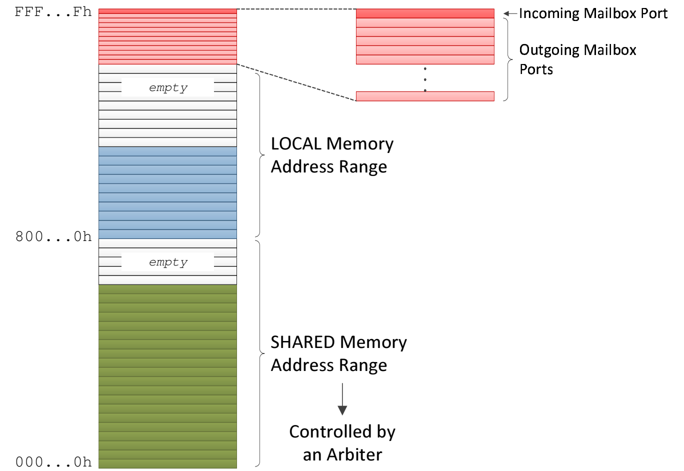

Fig. 2. Considered addressing space in each processing node.

III. Framework Class Structure

The whole prototyping framework was implemented using SystemC description language, adopting a programming style that considered a careful hierarchy of classes and methods, and by extensively exploiting inheritance mechanisms among the several modules of the system.

In particular, the *stimulus module* and the master and slave nodes of the *simulated device module* perform common actions in the framework, such as reading/writing data from/to the shared memory through the communication hub or sending/receiving command messages to other nodes. Their main cycle is constantly checking for new commands and reacts accordingly. As a consequence, the structure of their source code is similar and was implemented with a common class: `ModuleStruct`. Its source code is divided into four parts:

Ports - It includes three types of ports: *i)* a clock input port, used to keep the `main_action` cycle of the module running; *ii)* a communication interface, used to read and write data; and *iii)* a monitor port, representing the channel used to keep the monitor up with the status of the module.

Constructor - The constructor of the module incorporates the main settings that are established during modules initialization. Some examples of such settings are the size and access time of its internal memory, a message counter of the received messages (used for log purposes) and the capacity of each node's mailbox.

Methods - The `main_action` is the procedure that is constantly active within the module, checking for tasks that

need to be attended. The `msgRecv_process` method is fired every time the module receives a command message. The `check_incoming_message` is run inside the main cycle, to check whether there are command messages that have not yet been fulfilled. The `msgSend_process` method takes the responsibility for sending command messages. The access to the local/shared memory device is implemented with the `read_data` and `write_data` methods. Finally, the `log_status` method is responsible for reporting to the monitor the status of the module whenever there is some change.

Class variables - Among the set of variables declared within each module, there is an internal local memory, represented by `mem`, which is characterized by the parameters `mem_size` and `mem_delay`, regarding to its size and access time, respectively. The `my_id` variable specifies the id of the module. The mailbox of each node is implemented through an `sc_fifo` data structure, provided by SystemC. The first and last addresses of the internal memory are defined by `LOCAL_MEM_ADDRESS` and `LOCAL_MEM_ADDRESS_RANGE`, respectively.

Another important method that is present in both the master and slave nodes is the one where the user of the prototyping platform will be able to introduce the actual program of the processing algorithm that should be executed by each of these nodes. Such program, implemented in C/C++ programming language, will invoke the several provided methods to: *i)* read/write from/to the local and shared memory devices; *ii)* send/receive command messages through the respective mailbox. Besides these programs, the user should also supply a parameter that represents the processing time that models the execution of each step of the implemented algorithm. The value assigned to such parameter is tightly related to the type of architecture (and corresponding throughput) being modeled for the master and slave nodes.

IV. Modules Implementation

A. Stimulus

The stimulus first mission is to fill the shared memory with the data-set to be processed. This procedure is done via three different stages: *i)* reads the data from the input file; *ii)* writes the data in the shared memory; and *iii)* fills the address value where the data-set was saved in memory in the master's local structure `data`, together with an unique identifier. This information will be used by the master to locate in the memory the data-set to be processed. This process is repeated for every data instance that is read from the input file(s).

The process of starting the simulation of the simulated device is done via 2 steps: the first step is to wake up the slaves and monitor modules, and the second step is to wake up the master module, by unlocking its `main_action` method.

B. Monitor

The monitor structure is completely based on dedicated interfaces to the modules of the simulated device. All modules, except the shared memory, use these monitor's interfaces, so that the entire activity of the system is tracked down.

A `start` port is used only once, when the stimulus wakes up the monitor. This event triggers the `activate` thread, so that it can start receiving log events from the simulated device by using its `log` method. Through this channel, the simulated device's modules are able to report the monitor its status changes at the specific time that they occur.

In addition to registering the status tracking in a log file, the monitor also stores all the log data in a local array (`OP_TABLE`). This table stores the identification, status, start/end time-stamps and duration of every status that each module has been through the simulation. This way, the programmer may easily perform statistical calculations during and/or at the end of the simulation.

C. Simulated Device

1) Master: The master implementation inherits most of the `ModuleStruct` class structure (see section III), although some of its methods are locally implemented. Its main action procedure is constantly checking for events and tasks that need to be performed. In this case, it concerns the arrival of messages and the distribution of the data-set under processing to the several slave nodes.

Furthermore, it is also in charge of managing the several slave nodes. To accomplish this task, it keeps a two-dimension array which stores the identifier of each slave and its current condition: *free* or *busy*. Each time a slave finishes its processing or initiates its computation, its status is changed accordingly. This slave management procedure is accomplished through three dedicated methods: `next_destiny`, `isjobFree` and `finishjob`.

2) Slave: Just like the master, the slave structure is entirely built on `ModuleStruct` class. The only difference is a variable (`work_time`), which parametrizes the time that such node should take to execute each step of the data processing algorithm, according to the considered execution model.

The `main_action` process is permanently active and its job is to check for incoming messages. Usually, the execution of each received command requires transferring data from the shared memory into the local memory, processing it according to the command, writing the result back in the shared memory and informing the master about the completion of the job.

3) Memory Devices: Memory is a key point of the system, since it is the principal communication means between the master and the slaves. The prototyped architecture includes several memory units, such as the shared memory, the internal local memory of each slave/master node (see Fig. 1) and their corresponding mailboxes. All these memories are fully parametrized in the parameters file of the prototyping framework, in what concerns their capacity, read/write access times, etc. Likewise, the capacity of the mailboxes is defined through a set of constants.

Either the shared memory or the local memories residing in the master and slave nodes were implemented with a vector of characters within the same `memory_unit` class, which also

provides suitable read/write methods. In order to perform these operations, it is necessary to provide the parameters regarding to the address where the data will be read (written) from (to) and the length of the corresponding data.

4) Communication Hub: As it was referred before, the communication hub was structured in order to allow the implementation of a wide set of communication infrastructures, including single or multiple-bus paths, crossbar switches, NoCs, etc.. This allows to easily adapt the prototyped platform to the needs of the underlying application. In the presented prototype, it was considered the implementation of a single bus communication link, in order to evaluate its influence in the resulting scalability of the multi-core architecture.

The bus module (see Fig. 1) is composed by a main action method and built-in ports that communicate with all the nodes within the simulated device. Every node that requests the access to the bus has to be previously granted by an arbiter. For such purpose, the arbiter has a built-in interface that allows the bus to query the status of every node that places a request in the bus.

The bus module has three independent queues implemented with the SystemC `sc_fifo` data type. Each one is used to store the requests according to their priority (*high*, *normal* and *low*). When a node requests the access to the bus and is not granted to access it, it is inserted into the queue that corresponds to its assigned priority. Hence, as long as *high* priority requests exist, the *normal* and *low* priority requests will not be attended. Mailbox messages are always transmitted with the highest priority. Whenever the main action of the bus is executed, a routine check is run in all the request queues, executed by the `arbitrate` method provided by the arbiter.

The `is_owner` method is used by the bus to ask the arbiter if a specific node is owning the bus at a given instant. Every time a new request arrives at the bus, the arbiter is inquired about the status of the requester node, so that the request may or may not be immediately attended or inserted into the queue line. The ownership of the bus is executed in two steps: *i)* the bus queries the arbiter about the current owner; *ii)* only if there is not any current owner will the requester node be allowed to become the new owner of the bus.

V. Case-Study: DNA Sequence Alignment

Among the vast set of algorithms and tools extensively adopted to process and analyze biological sequence data (e.g. DNA and proteins), the Smith-Waterman (SW) dynamic programming algorithm [7] is widely used to determine the optimal local alignment between two given sequences. Considering any two strings S_1 and S_2 of an alphabet ε with sizes n and m, respectively, the local alignment of strings S_1 and S_2 reveals which pair of sub-strings of S_1 and S_2 optimally align, such that no other pairs of sub-strings have a higher alignment score. Let $G(i, j)$ represent the best alignment score between a suffix of strings $S_1[1..i]$ and a suffix of string $S_2[1..j]$. The SW algorithm allows the computation of $G(n, m)$, by recursively calculating $G(i, j)$, which will reveal the highest alignment score between the sub-strings of strings S_1 and S_2.

TABLE I
Example of an alignment score matrix.

		0	1	2	3	4	5	6	7	8	9	10	11	12
	G	ø	A	A	T	G	C	C	A	T	T	G	A	C
0	ø	0	0	0	0	0	0	0	0	0	0	0	0	0
1	C	0	0	0	0	0	3	3	0	0	0	0	0	3
2	A	0	3	3	0	0	0	2	6	2	0	0	3	0
3	G	0	0	2	2	3	0	0	2	5	1	3	0	2
4	C	0	0	0	1	1	6	3	0	1	4	0	2	3
5	C	0	0	0	0	0	4	9	5	1	0	3	0	5
6	T	0	0	0	3	0	0	5	8	8	4	0	2	1
7	C	0	0	0	0	2	3	3	4	7	7	3	0	5
8	G	0	0	0	0	3	1	2	2	3	6	10	6	2
9	C	0	0	0	0	0	6	4	1	1	2	6	9	9
10	T	0	0	0	3	0	2	5	3	4	4	2	5	8

The recursive relation to calculate the local alignment score $G(i, j)$ is given by Eq. 1, where $Sbc(S_1(i), S_2(j))$ denotes the substitution score value obtained by aligning character $S_1(i)$ against character $S_2(j)$ and α represents the gap penalty cost. An example of an alignment score matrix is shown in Table I. The substitution scores are usually positive for characters that match, thus denoting some similarity level between them. Mismatching characters may have either positive or negative scores, depending on the alignment type that is being performed.

$$G(i,j) = max \begin{cases} G(i-1,j-1) + Sbc(S_1(i), S_2(j)), \\ G(i-1,j) - \alpha, \\ G(i,j-1) - \alpha, \\ 0 \end{cases}$$

$$G(i,0) = G(0,j) = 0 \tag{1}$$

The main difficulty to efficiently implement this algorithm is concerned with the strict data dependencies in the computation of the several cells of the score matrix, as defined in Eq. 1, since each cell depends on the score obtained for its upper, left and upper-left neighbors. Such difficulty, allied with the tremendous demand for efficient processing structures, has pushed the development of several architectures specifically optimized for the implementation of this algorithm [8] and naturally justified its adoption as a proof-of-concept application to evaluate the proposed prototyping framework. Naturally, other different algorithms, from either the bio-informatics or other applications domains, could equally be used to evaluate the performance of the modeled multi-core architecture with the proposed prototyping framework, in what concerns several different design parameters.

VI. Experimental Evaluation

The evaluation of the proposed prototyping framework was conducted by simulating the execution of a parallel implementation of the SW alignment algorithm. The considered evaluation was based on the application of the widely adopted High Throughput Short Read (HTSR) technologies [9], where the DNA query sequences under analysis are cut in shorter fragments (*reads*), which are individually aligned against a

reference sequence. The considered data-set comprises one 1000 nucleotides long reference sequence extracted from the Homo Sapiens chromosome 1 GRCh37 primary reference assembly, and 500 query sequences (*reads*), each one with 35 nucleotides, extracted from the Homo Sapiens genome (Run ERR004756 of study ERP000053).

Along this procedure, the master distributes the *reads* and the reference sequence to the slaves and these become in charge of executing the DNA alignment algorithm and of sending back the scoring result, corresponding to the best matching score value between the two analyzed sequences. The SW instances in each slave node are completely independent from each other and there is no data dependency. Hence, the master node is the one in charge of issuing the commands to the slaves, specifying the query (*read*) that each slave should fetch from the shared memory and align against the reference sequence.

Within this application context, the main aim of the presented experiment was to evaluate the ability of the developed framework to assess the parallelization scalability that is offered by the prototyped multi-core architecture, by evaluating not only the speedup values that may be achieved, but also the possible degradation losses that arise due to the inherent contention in the bus when the several nodes concurrently try to access the shared memory. Such degradation will depend not only on the number of slave nodes that will be incorporated in the multi-core processor, but also on their specific performance (achieved through software optimizations or dedicated architectures) to implement the alignment procedure.

For such purpose, it was defined a processing-balance parameter (K) in the considered model of the multi-core architecture, in order to model the alignment performance of the slave nodes. This parameter represents the relation between the amount of time (clock cycles) that each slave node needs to compute one single cell-update in the scoring matrix (see eq. 1) and the time that it needs for a single local memory access (one read or write operation). Slave configurations with a K parameter lower than 1.0 represent highly optimized implementations that are capable of computing more than one cell-update per clock cycle, achieved either by using SIMD programming models [10] or dedicated VLSI architectures [8]. By introducing this parameter in the prototyped model, the system designer will be offered the possibility of evaluating different alternatives to implement the slave computing nodes, trying to obtain, in an early design stage, the best compromise between the resulting overall performance and the corresponding implementation cost.

A. Alignment performance (speedup)

The chart presented in Fig. 3 depicts the alignment time and the speedup (using a single-slave configuration as reference) offered by the prototyped multi-core architecture to process the benchmarked DNA data-set by considering a variable number of slave nodes. As it was expected, the alignment performance (speedup) increases linearly with the number of slaves. To illustrate the relation between the processing-balance parameter (K) and the resulting contention in the bus, this chart represents two distinct scenarios, for K=1 and K=0.2. From the obtained results, it can be observed that not only is the conceived framework able to accurately model the parallel scalability offered by the prototyped multi-core architecture, but it can also represent the bus contention effect when the access rate to the shared bus increases for lower values of K.

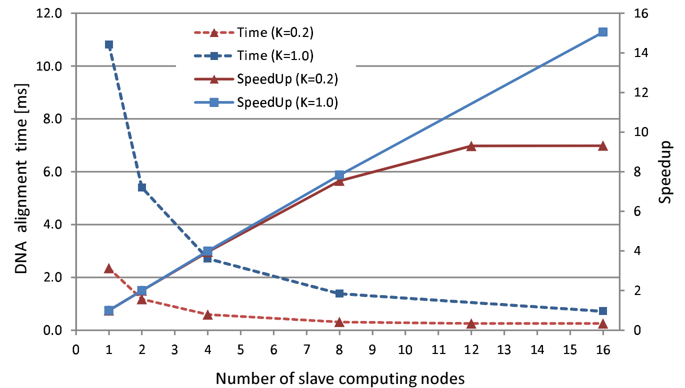

Fig. 3. Evaluation of the prototyped multi-core performance scalability, by using the conceived simulation framework.

B. Bus contention

To demonstrate the relation between the inherent bus contention and the resulting performance loss of the prototyped architecture, the conceived framework also provides a time-diagram that represents the state of each processing node (master and slaves) for any instant of the simulation (see Fig. 4, for 8 slave nodes). In this chart, it can be observed that as soon as the master issues the first set of commands to the slaves (left side), the processing cycle initiates (right side) and the slaves start executing the alignment algorithm as soon as they fetch the DNA data from the shared memory. Then, each slave

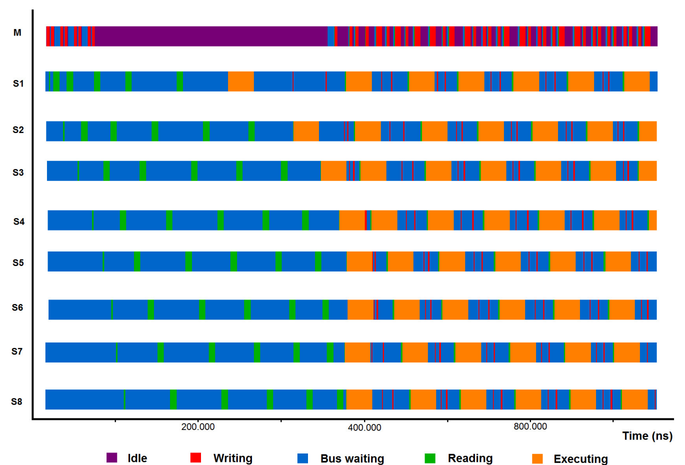

Fig. 4. Time diagram provided by the conceived framework with the state of the processing nodes (master (M) and slaves (Sn)) along the simulation.

writes the alignment scoring result in the shared memory and waits for another command, to be issued by the master.

This diagram clearly illustrates the bus contention effect. Whenever more than one processing node requests the access to the bus, the arbiter only grants access to one single node, thus making all the others remain in the *Bus waiting* state.

C. Effect of the processing-balance parameter (K)

The chart presented in Fig. 5 illustrates the relation between the considered performance level for the slaves computing architecture (modeled by the K parameter) and the resulting speedup that is offered by the prototyped multi-core architecture, when using 8 slave nodes. The presented variation clearly demonstrates the influence of the data-transfers when the alignment time required to process each query is shorter. This information is most useful for the system designer, in order to allow him to anticipate the adoption of better technologies for the communication hub and more efficient data-transfer mechanisms when higher performance slave architectures are considered, thus avoiding undesired speedup penalties.

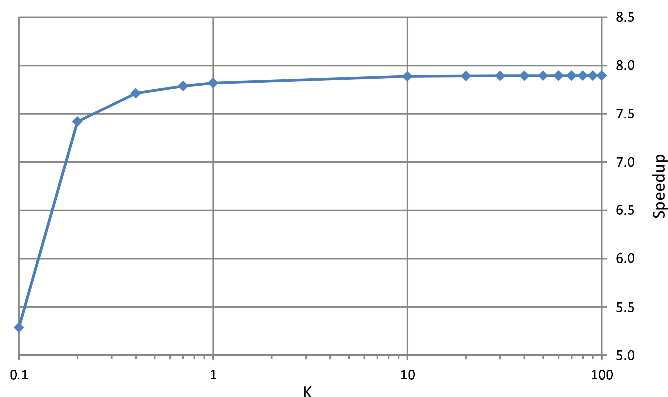

Fig. 5. Influence of the processing-balance parameter (K), relating the computing time and the bus contention penalty arisen from the data transfers, evaluated by the conceived framework.

D. Framework simulation time

Table II presents the simulation time when the conceived prototyping framework was used to evaluate the performance of the underlying multi-core architecture, by considering K=10, 500 query sequences and a variable number of slave nodes. These simulations were executed in an off-the-shelf computer, equipped with an Intel Core 2 Quad Q6600 processor, running at 2.8 GHz with 1GB of RAM. The obtained results demonstrate that any of the considered simulations was executed in less than one minute, which evidences the practicability of the prototyping framework to accelerate this preliminary design stage of the multi-core architecture.

TABLE II
FRAMEWORK SIMULATION TIME.

Number of slave nodes	1	2	4	8	16
Simulation time [s]	47.7	29.2	15.7	10.8	10.3

VII. CONCLUSION

A prototyping simulation framework to support the design and early development stages of an heterogeneous multi-core architecture for DNA sequences processing was proposed. The conceived framework was entirely developed using the SystemC description language and offers a wide flexibility to customize and parametrize several characteristics of the prototyped architecture, such as the number of computing nodes, the effective processing throughput of each node, the capacity and access time of the memory devices, etc.

The included experimental evaluation demonstrated that the conceived framework provides the system designer with a very useful preliminary characterization of the prototyped architecture. In particular, the presented evaluation demonstrates the relation between the number and performance of the computing nodes and the resulting alignment performance gain (speedup), as well as the inherent bus contention losses in the shared resources (bus and shared memory). Such information is usually regarded with uttermost importance, in order to assist the system architect in the selection of the most suitable communication technologies and data-transfer mechanisms. With such support, not only will be possible to avoid undesired speedup penalties, but the best compromise between the resulting overall performance and the corresponding implementation cost will be achieved in a earlier design stage.

ACKNOWLEDGMENTS

This work was partially supported by national funds through FCT – Fundação para a Ciência e a Tecnologia, under the project "HELIX: Heterogeneous Multi-Core Architecture for Biological Sequence Analysis" with reference PTDC/EEA-ELC/113999/2009, and project PEst-OE/EEI/LA0021/2011.

REFERENCES

[1] J. Gerlach and W. Rosenstiel, "System level design using the SystemC modeling platform," in *Workshop on System Design Automation*, 2000, pp. 185–189.

[2] Çağkan Erbaş, "System-level modelling and design space exploration for multiprocessor embedded system-on-chip architectures," Ph.D. dissertation, Advanced School for Computing and Imaging, Turkey, 2006.

[3] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *Proc. International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 2003, pp. 19–24.

[4] M. Alassir, J. Denoulet, O. Romain, and P. Garda, "A SystemC AMS model of an I2C bus controller," in *Proc. Int. Conf. Design and Test of Integrated Systems in Nanoscale Technology (DTIS)*, 2006, pp. 154–158.

[5] G. B. Defo, C. Kuznik, and W. Muller, "Verification of a CAN bus model in SystemC with functional coverage," in *Proc. Int. Symp. Industrial Embedded Systems (SIES)*. IEEE, 2010, pp. 28–35.

[6] J. Booth and J. Kulick, "SystemC modeling of a parallel processor broadcast interconnection system," in *Proc. of SoutheastCon*. IEEE, 2002, pp. 76–81.

[7] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *J. Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[8] N. Sebastião, N. Roma, and P. Flores, "Integrated hardware architecture for efficient computation of the n-best bio-sequence local alignments in embedded platforms," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 7, pp. 1262–1275, Jul. 2012.

[9] M. J. Chaisson and P. A. Pevzner, "Short read fragment assembly of bacterial genomes," *Genome Research*, vol. 18, no. 2, pp. 324–330, 2008.

[10] M. Farrar, "Striped smith-waterman speeds database searches six times over other SIMD implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2007.