Received January 27, 2020, accepted February 11, 2020, date of publication February 20, 2020, date of current version March 2, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2975360

NOCA – A Notification-Oriented Computer Architecture: Prototype and Simulator

ROBSON R. LINHARES[1,2], **LEONARDO F. PORDEUS**[1], **JEAN M. SIMÃO**[1,2], **AND PAULO C. STADZISZ**[1,2]

[1]Graduate Program on Electrical Engineering and Industrial Informatics, Universidade Tecnológica Federal do Paraná (UTFPR), Curitiba 80230-901, Brazil
[2]Graduate Program on Applied Computing, Universidade Tecnológica Federal do Paraná (UTFPR), Curitiba 80230-901, Brazil

Corresponding author: Robson R. Linhares (linhares@utfpr.edu.br)

This work was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001 - and in part by a grant from Araucaria Foundation.

ABSTRACT The Notification Oriented Paradigm (NOP) introduced a new organization of software and hardware logic based on notifications among computational entities. This NOP new organization avoids processing redundancy and allows processing unit decoupling, therefore permitting proper processing performance and processing parallelism/distribution. Thus, the NOP provides means to make efficient use of the parallel execution capabilities of modern computing systems. However, as expected, the execution dynamics of NOP, based on notifications, is not efficiently performed by the hardware of most current computing systems. This paper presents a new solution called Notification-Oriented Computer Architecture (NOCA), which is suitable for the execution of software developed according to the NOP computing model. The NOCA was designed according to principles of generality and scalability, which allow it to execute NOP software of any size by fetching the application from memory. The proposed architecture is organized as a fine-grained multiprocessor that hierarchically executes instructions through sets of specialized processing cores. Preliminary experiments performed on prototypal FPGA implementation of the NOCA showed the expected behavior of executing NOP applications according to its theoretical computing model. This paper also presents experiments performed on a NOCA simulator extending the scale of parallelization of applications. Results show improvements in maximizing the speedups at higher scales of parallelization, as well as minimizing the effects of processor-to-memory communication bottlenecks by reducing the number of required memory accesses during execution.

INDEX TERMS Computer architecture, notification-oriented paradigm, parallel computer architecture, computing paradigm.

I. INTRODUCTION

Computer architectures have lately evolved towards the design and construction of processors with multiple cores [1], [2]. However, performance improvements are dependent on the efficient use of the multiple parallel cores made available by these novel execution platforms. This is an issue for both von Neumann-based and alternative architectural solutions for parallelism, namely as dataflow approaches, whose parallel processing capabilities are usually not adequately exploited by the current high-level and flexible programming solutions [3]–[5].

As a matter of fact, some of the most used state-of-the-art programming techniques, such as the Object-Oriented Paradigm (OOP) from imperative programming or Rule-Based Systems (RBS) from declarative programming, are subject to limitations or deficiencies related to coupling and redundancies that impair the use of parallelism features provided by multicore architectures. Those deficiencies regard to coupled and unnecessary logic-causal evaluations caused by a loop and/or search orientation of the usual paradigms in general [6]–[8]. As a consequence, there is an increasing need for techniques and tools that are suitable for the development of parallel software [9], [10].

In this context, the Notification-Oriented Paradigm (NOP) has been proposed, based on a previous control and inference theory [7], [11]. The NOP proposes eliminating some of

The associate editor coordinating the review of this manuscript and approving it for publication was Songwen Pei.

those deficiencies found in the current most used sequential paradigms, namely imperative and declarative ones. However, at same time, a new paradigm should preserve and improve some qualities from those ancient paradigms, such as named state from imperative paradigm (namely from the object-oriented sub paradigm), as well as high-level development from declarative paradigm (namely from the logic sub paradigm) [7], [11], [12].

This new paradigm called NOP is characterized by a logical model organized as logical-causal and factual-executional entities whose inference occurs collaboratively by means of very precise notifications among them. The logical-causal entities act as *Rule* entities whose activation occurs reactively in response to state changes of the factual-executional entities, called *Fact Base Elements* (*FBE*). By using precise notifications among constituents of the *FBEs* and the *Rules*, the need for additional inference mechanisms to perform the matching among them is eliminated [7], [11]. These properties of the NOP allow decreasing or even eliminating code/entity redundancies and coupling, thereby improving the processing performance and facilitating parallelism and distribution. Moreover, the NOP development occurs by means of a high-level declarative style that automatically establishes the notification links [7], [10], [12]–[15].

The NOP solutions have been experimented to improve performance and even parallelism/distribution in the usual von Neumann-based platforms, both mono and multi-core ones [10], [14]. However, such platforms cannot fully achieve NOP benefits. Thus, in order to provide a platform that allows the proper execution of NOP applications, this paper presents a computer architecture referred to as Notification-Oriented Computer Architecture (NOCA).

The NOCA aims at implementing the dynamics of the NOP execution model as closely as possible to its theoretical model, in a manner that particularly allows it to implicitly profit from the facility of distribution and parallelism provided by the NOP. Moreover, the NOCA aims at generality in the sense that it is capable of alternating the execution of different NOP applications only by replacing software in memory, which is a feature traditionally expected for the so-called modern computers.

In this paper, the NOCA is described by presenting its structural and dynamic models in detail, which is prototyped and tested in an FPGA platform. In addition, results of experiments in a software simulator are presented by comparing the execution performance of NOP software running on the NOCA at different scales of parallelization, as well as the consequent effects on the speedup and memory and bus contentions.

This paper is organized as follows. Section II summarizes the theory concerning the NOP. Section III shows related work. Section IV details the proposed architecture. Section V presents the case of study, results from the experiment, and the discussion about these results. Section VI compares the NOCA with other current architectures. Section VII presents conclusions and perspectives for future work.

II. NOTIFICATION ORIENTED PARADIGM (NOP)

The NOP was proposed as a new paradigm for systems, firstly software but subsequently hardware as well, in order to provide a new logical model that intends to contribute to higher productivity and quality in the development and execution of automated processes. Particularly, this paradigm intends to promote a better system execution performance and greater construction facility of complex systems, especially parallel and distributed systems [7], [10]–[18].

The NOP logical model allows reducing or even eliminating some of the issues concerning classical development paradigms such as Imperative (IP) and Declarative (DP) Paradigms, which respectively and even particularly include Object-Oriented Programming (OOP) and Rule-Based Systems – (RBS). As examples of those issues, it can be pointed out the often strong coupling among entities on logical-causal calculation as well as structural and temporal redundancies [7], [11].

Structural redundancies occur when a given logical and/or causal calculation is unnecessarily repeated elsewhere in the code. In turn, temporal redundancies occur when a given logical and/or causal calculation is unnecessarily re-executed over time, even though the data involved in this calculation has not changed since its last execution [7], [8], [19]. These redundancies tend to cause code coupling, which makes processing parallelism and distribution more difficult to achieve [11], [12].

A. NOP STRUCTURE

A system developed according to the NOP, referred to as a NOP system, comprises a set of logical-causal and factual-executional entities declaratively defined by the designer. These entities collaborate by means of very precise notifications among them, thereby defining an innovative inference process that is distinct from those used in IP and DP [7], [11], [16].

The NOP logical model can be described by means of a class diagram, as shown in Fig. 1. The factual-executional elements are modeled by means of the *Fact Base Element (FBE)* class. Each element is defined by an *FBE* and aggregates a set of *Attributes* (i.e., notifiable state variables) and *Methods* (i.e., functions or operations able to be instigated) described by the *Attribute* and *Method* classes, respectively.

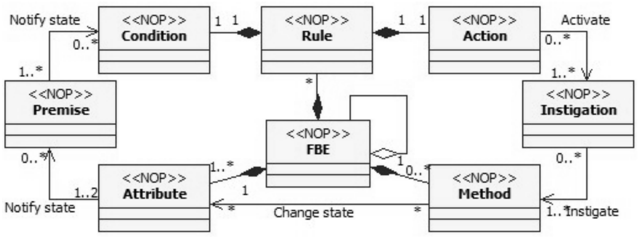


FIGURE 1. NOP logical model [7].

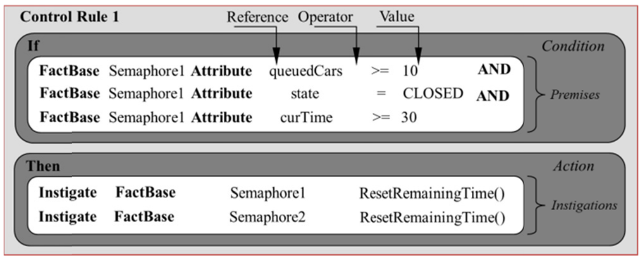

Control Rule 1
Reference
Operator
Value
If
Condition
FactBase Semaphore1 Attribute queuedCars >= 10 AND
FactBase Semaphore1 Attribute state = CLOSED AND
FactBase Semaphore1 Attribute curTime >= 30
Premises
Then
Action
Instigate FactBase Semaphore1 ResetRemainingTime()
Instigate FactBase Semaphore2 ResetRemainingTime()
Instigations

FIGURE 2. NOP Rule entity example expressed as a logic-causal rule.

The logical-causal elements, in their turn, are modeled by means of the *Rule* class. Each entity defined as a *Rule* represents a logic-causal rule, as exemplified in Fig. 2. Still, each *Rule* (entity) aggregates a *Condition* and an *Action* respectively, described by the *Condition* and *Action* classes. Also, each *Rule* is an entity to be notified by some *FBE* [7], [11].

The *Condition* of a *Rule* defines the logical calculation on a set of *Premises* whose result determines whether the *Rule* is activated or not. Each *Premise* is an instantiation of the *Premise* class and comprises a relational expression of two *Attributes* or an *Attribute* and a constant.

In turn, the *Action* of a *Rule* is responsible for triggering a set of *Instigations*, which are described by the *Instigation* class. These *Instigation* instances represent what must be processed when a *Rule's Action* is activated. Actually, *Instigations* activate *Methods* to be executed, which are described by the *Method* class. The *Methods* contain the functional logic that acts over *FBE Attributes* [7], [11].

Fig. 2 presents an example of *Rule* expressed in the form of a logic-causal rule. This *Rule* is part of a traffic control simulator using traffic lights (semaphores) encapsulated as *FBE* s. The *Condition* of this *Rule* handles the decision of decreasing the "closed" time of a *Semaphore FBE*, depending on the number of vehicles waiting for the semaphore to "open". This *Condition* evaluates three *Premises* that perform the following evaluations of the *Semaphore FBEs:* a) Is the number of queued vehicles greater than or equal to 10? b) Is the semaphore closed? c) Is the semaphore closed for 30 seconds or more?

In this example, the *Action* contains two *Instigations* to a) reset the remaining time for Semaphore1, which will subsequently trigger another *Rule* that will change its state to "open"; and b) reset the remaining time for the Semaphore2, which is the opposite semaphore belonging to the same crossing, to change its state, subsequently triggering another *Rule* that will change Semaphore2's state to "closed". Effectively, what each *Instigation* does is to instigate one or more *Methods* responsible for performing the services or operations defined by an *FBE*.

B. NOP INFERENCE

The given NOP organization of entities and their relationship allows an innovative form of inference process based on notifications, nowadays called Notification Oriented Inference (NOI) [7]. The NOI is exemplified in Fig. 3, which shows the notifications among NOP entities, such as *FBEs* notifying *Rules* by means of *Attributes*, *Premises*, and *Conditions*.

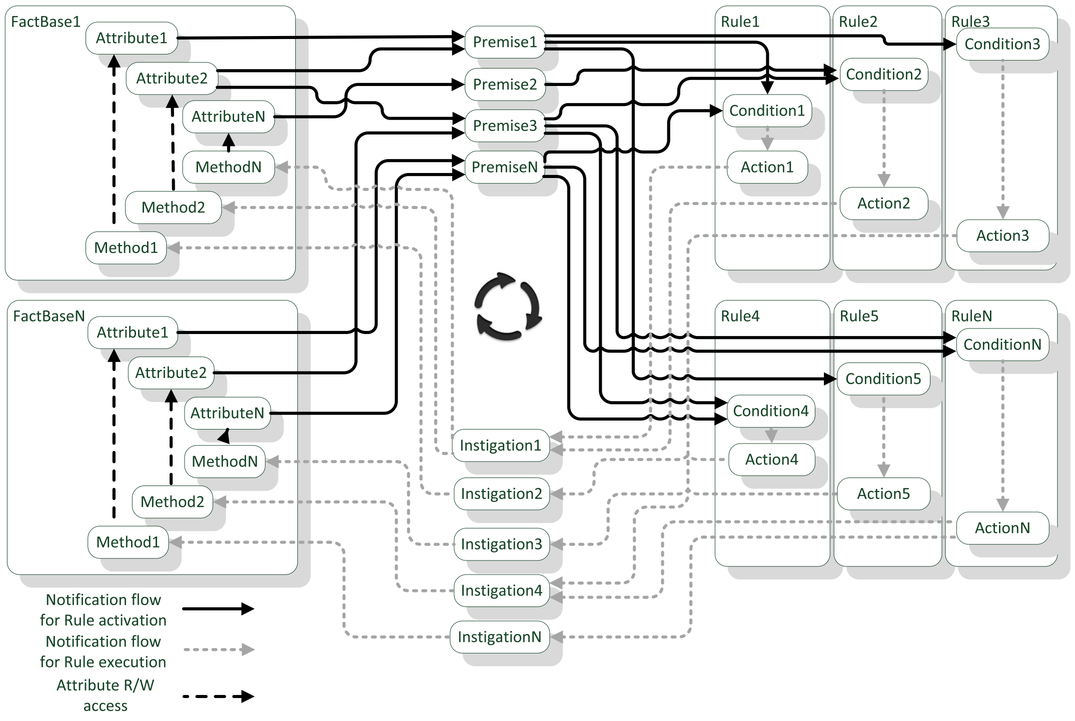


FIGURE 3. Example of notification chain [17].

In short, in NOI, for each change in the state of each *Attribute* of an *FBE*, this *Attribute* precisely notifies its concerned *Premises*, which perform their logical operation, thereby updating their states. Also, for each change in the state of each *Premise*, this *Premise* precisely notifies its concerned *Conditions*, which perform their logical-causal calculation, thereby updating their state [7], [11], [16].

If the result of that calculation of a *Condition* is true, the respective *Rule* is approved and can be activated if there is no conflict. In the case of conflict, distinct priorities of *Rules* can be used to solve it under some given approach. Once a *Rule* is activated, this results in the execution of its *Action*. Upon execution, the *Action* notifies its connected *Instigations*, which, in turn, are responsible for notifying the corresponding *Methods* and triggering their execution [7], [11], [16].

Still, the notification links are connected in build time. For instance, when a *Premise* mentions an *Attribute*, the latter takes into account the former as a receiver for its notifications in runtime. Naturally, the links connected in build time allow the execution dynamics above described and exemplified in Fig. 3.

Fig. 3 presents an example of a NOP system composed of six logical-causal entities (i.e., *Rules*) and two factual entities (i.e., FBEs). This figure illustrates the composition of the notification chain that integrates the entities according to the NOP logical model. It also shows the structure that enables the execution dynamics according to NOP.

The bold lines in Fig. 3 represents the notification flow for *Rule* activation, starting with an *Attribute* change and propagating to the notified elements in order to the approval of a set of *Rules* as a result. The lighter dotted lines, in their turn, represent the notification flow for *Rule* execution, resulting in the activation of *Methods* that update *Attribute* values as indicated by the dashed lines. It can be noticed that the essence of computation under the NOP is organized and distributed among reactive entities, which collaborate by means of notifications. This framework defines the notification mechanism and the application execution flow as a consequence.

The NOP leads to a new form of system designing, under which the execution flows are distributed among a set of quite uncoupled entities (e.g., *Attributes*, *Premises*, *Conditions*, and *Rules*). Different parallel or sequential execution flows may exist due to the logic of the *NOP entities*, without the designer having to set their structural linkages. Thus, the parallelism expression is intrinsic to the NOP logical model, making it an alternative for the design of parallel software in a way, potentially even more advantageous than existing techniques [7], [13], [17].

C. PREVIOUS NOP IMPLEMENTATIONS

Some previous approaches to design systems, according to the NOP, have been developed. These approaches include: NOP frameworks (in C++, C#, Java. and Elixir), allowing the implementation of NOP software, for von Neumann architecture, using typical OOP abstractions such as classes and objects, but now under notification approach of-course [16], [18]; prototype of a language and compiler called NOPL, for von Neumann architecture, that allow generating low-level language, namely C/C++, but under the NOP background [16]; circuit templates called NOP Digital Hardware (DH), afterward associated with new version of NOPL that allow the high-level implementation, in total conformance with NOP principles, of a determined application in reconfigurable hardware [13]; and a NOP co-processor (CoNOP), which is based on NOP-DH and can be reconfigured for a determined application, thereby allowing the hybrid execution of the NOP inference process in collaboration with a von Neumann core [17], [20].

All these implementations have been absolutely important to demonstrate a set of NOP properties, such as declarative high-level implementation, implicit entity decoupling allowing parallelism and distribution, actual avoidance of entity redundancy allowing high performance in execution terms, and usefulness for both software and hardware. However, even if very useful, these approaches present certain shortcomings in executing notification-oriented software.

The approach based on the C++ framework relies on software structures that are typical of the Imperative Paradigm (IP) as well as on data structures to implement the notification dynamics of the NOP, which can aggregate significant overhead to the execution performance. Even so, the results present performance improvements when compared to usual C++ imperative implementation and CLIPs/Rete declarative implementation, and these performance improvements are even increased when running the C++ framework as a fine-grained multithreaded application in a multicore system [7]. Nevertheless, in order to minimize such aforementioned overhead, in each core, some effort has been employed in developing a NOP language/compiler prototype to generate C/C++ source code specific for each NOP application, therefore optimizing and simplifying the use of the data structures [16].

The hardware-based approaches, in their turn, map the design of a NOP application (or part of it, as in the CoNOP) directly into a specialized electronic circuit. This mapping results in a high-performance circuit that is specific to a NOP application and executes in a highly-parallelized way, in opposition to the sequential execution imposed by implementations running on von Neumann cores. Nowadays, this mapping can be performed by means of an adaptation of the NOPL (NOP language/compiler) that generates VHDL, thereby allowing developing a hardware-based NOP application from a high-level development language. However, those solutions lack generality and scalability, in the sense that any change on the application logic implies the hardware reconfiguration of the platform where it is implemented and has its size limited to the capacity of that hardware device [17].

In this current paper, it is presented another hardware solution to NOP called NOCA (Notification-Oriented Computer Architecture). The NOCA proposes a new hardware-based approach that aims to minimize or even eliminate some of the drawbacks of the previous approaches. However, before presenting the NOCA, the next section reviews related work about architectural concepts that have been considered when designing the NOCA and are pertinent for its better understanding.

III. RELATED WORK

The ease with which software is developed and executed represents one of NOP's implicit characteristics that should be addressed in the new proposed architecture. This section briefly reviews the history of parallel computer architectures, in particular the most relevant ones, to allow afterward positioning the NOCA in relation to these models, inclusive in terms of ease of development.

Along with the recent history of computing, much effort has been made to develop and improve computer architectures, aiming to run software with some level of parallelism. Two computing models have been, particularly, more widely employed in the design of parallel computer architectures:

1) Von Neumann model, in which a program is stored in memory and has its next instruction fetched by the processor from the memory address pointed to by a program counter. The address pointed to by the program counter is incremented with sequential addresses or updated with results of branch instructions used to redirect the program control flow.
2) Dataflow model, according to which the program operations (instructions) have their execution enabled and performed as the data (operands), on which they depend, become available [3].

A. PARALLELISM IN THE VON NEUMANN MODEL

It is possible to achieve some parallelism at several abstraction levels, even under an essentially sequential execution model such as the von Neumann model. Instruction Level Parallelism (ILP) may be achieved by employing techniques such as pipelining. This technique allows more than one instruction to be executed at the same clock cycle by a processor by dividing its processing into many execution stages [21].

At a higher level of abstraction, Thread Level Parallelism (TLP) may be achieved by partitioning software into different control flows, which can be concurrently executed but with well-defined synchronization points. At the hardware level, architectural resources such as multithreading aim at facilitating TLP by controlling the thread execution and scheduling according to the state of other threads. For example, this occurs in a situation when a thread is stalled due to a pending I/O operation with high latency [22]–[24].

Another technique for TLP implementation is based on Chip Multiprocessor (CMP) architectures, which are composed of multiple execution cores on the same chip, also called multicore architectures. Even though this technique is currently the most commonly employed approach for designing microprocessors, this kind of architecture presents some additional challenges, e. g. managing concurrency among simultaneous memory accesses performed by the different cores as well as ensuring cache memory coherency in implementations where each core has a separate cache for its individual use.

Concerning programming issues, the focus on developing computers compatible with the von Neumann model along the years [25] has favored the propagation of compatible programming models and languages. The programming model of a von Neumann computer, that is fundamentally based on variables as storage, control instructions as a way to manage the execution flow, and assignment instructions to implement memory reads and writes [26], is the technological base for the imperative paradigm and, consequently, to the most popular procedural and object-oriented programming languages, such as C/C++, Java, among others. Thus, the developer's adaptation to new frameworks, development environments or imperative programming languages that are compliant to the von Neumann model requires relatively little effort, which is a considerable advantage over using other less popular computing or programming models.

As a disadvantage, regardless of the category or level of parallelism, the von Neumann model presents a characteristic known as "von Neumann bottleneck" that consists of frequently fetching instructions from memory outside the execution core, whose latency and bandwidth are typically limited [25]. These limitations have increased in the latest years due to the effect known as "memory wall", which is the increasing discrepancy between the processor execution performance and the memory access performance [27]. According to [25], this discrepancy reached a factor of 1000 by 2007 and kept on increasing.

Indeed, the von Neumann model was not primarily designed for the execution of parallel software. Even though the execution of programs can be parallelized to some extent, this parallelization would impose the ways of communication and synchronization between the involved threads, with the consequent execution overhead [28]. Additionally, the execution model constrains the expression of the parallel logic by the programmer.

B. PARALLELISM IN THE DATAFLOW MODEL

In the context of the dataflow model, in its turn, the program structuring itself favors the parallel execution of individual instructions. That means instructions are able to be executed as long as their operands are available. From a technological point of view, several architectural techniques have been developed to help improve some characteristics of dataflow execution. One example is the use of an Explicit Token Stores (ETS) for optimizing the search and dispatch of the available data to the proper execution units [29] Another example is the I-Structures, consisting of memory blocks that define specific synchronization semantics in order to generate a notification to a dataflow instruction about the availability of the data on which it depends [30].

Examples of typical architectures based on the dataflow model are SDF [5], WaveScalar [31] and MAD [32].

The SDF (Scheduled Dataflow) consists on a hardware that executes small non-blocking threads according to the availability of their input operands, thus in a dataflow-oriented way; the WaveScalar groups the dataflow instructions to be executed in sets called ''waves'', which are defined in build time in order to keep the instructions that produce and consume the data close to each other, therefore facilitating the flow of data between them; and the MAD aims at optimizing memory access performance by means of a dataflow execution strategy.

The dataflow model is theoretically more suitable for parallel computing than the von Neumann model because its execution is intrinsically parallel and because it presents more suitable abstractions for the development of concurrent programs [33]. However, some issues regarding the need to do a search process for operand matching and the impact of fine execution granularity on the pipeline and on the number of accesses to lower memory levels, affect the performance of those architectures and have hampered their use. There are also some drawbacks inherent to the dataflow programming model, such as the absence of variables that make dataflow programming more similar to a functional programming model that can be difficult to use and inefficient for many applications [4].

C. NOP IN HARDWARE VERSUS ARCHITECTURES BASED ON VON NEUMANN AND DATAFLOW MODELS

Concerning the dynamics of the execution model, the main differences between hardware solutions based on NOP model and on von Neumann and dataflow models reside in the following aspects:

1) The execution of NOP applications is fundamentally non-sequential since it basically depends on the mapping between facts (data) and *Rules* (relations among data and the consequent actions). This mapping can naturally lead to the simultaneous (thus, potentially parallel) triggering of several *Rules*.
2) The execution of NOP instructions is not dependent on the availability of operands, as in dataflow instructions, but on the availability of notifications that are generated as a result of the causal-logical calculations applied over data being updated. Indeed, the semantics of NOP *Methods* allows them to operate over data stored in memory without necessarily consuming them, but keeping the ability to generate notifications due to changes in *Attribute* values.

As a matter of fact, NOP approaches in hardware such as NOP-DH and CoNOP are alternatives to the current architectures. Nevertheless, one problem of the current NOP approaches in hardware would be that each application in NOP-DH must be a specific circuit, and in CoNOP, the number of NOP entities to be executed is limited to a given number. In this sense, the NOCA is proposed in order to allow a NOP application to be loaded from memory to a set of processors and executed in a more scalable and flexible way.

IV. DESCRIPTION OF THE PROPOSED ARCHITECTURE

This section presents the proposed notification-oriented computer architecture, referred to as NOCA [17], [34].[1] This architecture supports NOP software execution by means of its own assembly code, which can be generated by means of a Rule-based high-level NOP programming language/compiler [16]. This model gives flexibility to the application, once applications can be developed in a high-level language and changed or replaced in memory when needed.

A. NOCA LOGICAL MODEL

The development of the NOCA was based on a set of generic requirements related to the NOP features and to its execution model, as listed below.

1) The NOCA shall be capable of executing software solely composed of NOP elements and, optionally, of sequential functions according to the von Neumann model.
2) The NOCA shall be generic, in the sense that any change in the NOP application being executed depends only on software changes, thus not requiring any hardware reconfiguration.
3) The NOCA shall define a low-level instruction set (and corresponding assembly language) that implements the functionalities of the NOP notification elements.
4) The NOCA shall define processing units that are capable of executing the low-level architecture instructions and the flow of notifications with parallelism.
5) The NOCA shall be capable of executing a NOP application even if it is composed of more notifying elements than the number of processing units available for their execution. This enables scalability in the sense that the size of a NOP application to be executed is limited only by the amount of memory available for storing the respective software.

Based on these requirements, the logical model for the NOCA was designed as shown in Fig. 4. The notification model is positioned on the left side and the way the model elements map to the NOCA instruction set is in the middle. Some of the model elements (*Rule, Action* and *Instigation*) are not mapped to any specific instruction, once they do not perform any calculation or causal-logical operation but act as aggregators or notification routers for NOP software only.

In its turn, the package on the right side of Fig. 4 shows the instruction set mapped to some hardware processes, which are responsible for executing the respective instructions. These processes are grouped together as sets, each of them representing a group of processors that send and receive notifications across the same channels. In addition, it is defined: a specific process for *Attribute* change detection,

[1] This architecture was subject of a Ph.D. thesis [17] and has been previously and shortly described in [34] in Portuguese language. Indeed, NOCA is described in [34] with fewer details than presented herein and the reported experiments, performed on a FPGA implementation, do not contain neither the NOCA integration with NOP Language/Compiler nor the data obtained from the NOCA simulator.

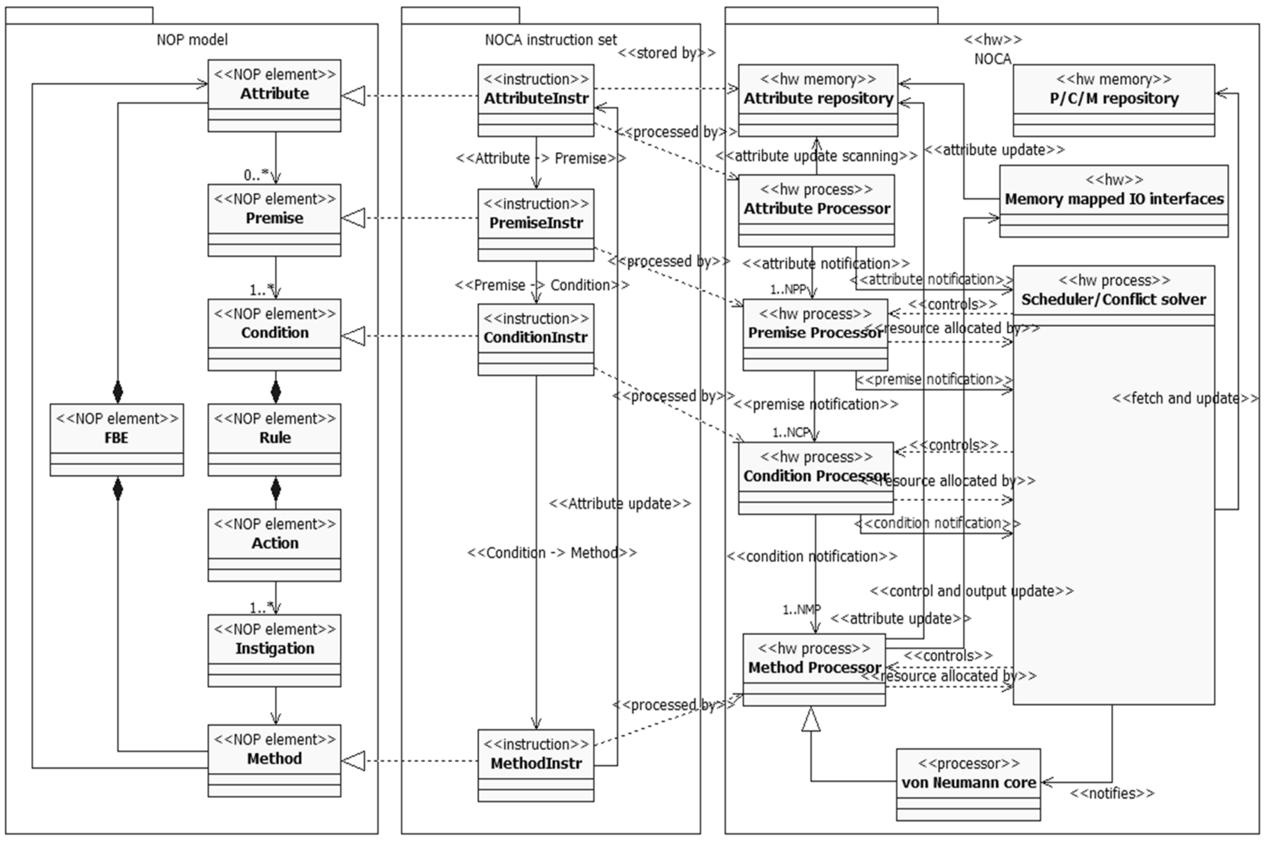

NOP model
<<NOP element>> Attribute
<<NOP element>> Premise
<<NOP element>> Condition
<<NOP element>> FBE
<<NOP element>> Rule
<<NOP element>> Action
<<NOP element>> Instigation
<<NOP element>> Method
0..*
1..*
1..*
NOCA instruction set
<<instruction>> AttributeInstr
<<instruction>> PremiseInstr
<<instruction>> ConditionInstr
<<instruction>> MethodInstr
<<stored by>>
<<processed by>>
<<Attribute -> Premise>>
<<Premise -> Condition>>
<<Attribute update>>
<<Condition -> Method>>
<<hw>> NOCA
<<hw memory>> Attribute repository
<<hw memory>> P/C/M repository
<<hw>> Memory mapped IO interfaces
<<hw process>> Attribute Processor
<<hw process>> Premise Processor
<<hw process>> Condition Processor
<<hw process>> Method Processor
<<hw process>> Scheduler/Conflict solver
<<processor>> von Neumann core
<<attribute update scanning>>
<<attribute update>>
<<attribute notification>>
<<controls>>
<<resource allocated by>>
<<premise notification>>
<<fetch and update>>
<<condition notification>>
<<control and output update>>
<<notifies>>
1..NPP
1..NCP
1..NMP

FIGURE 4. NOCA logical model [17].

associated with the corresponding repository; a process responsible for scheduling actions and solving conflicts among *Rules* (S/CS); and a von Neumann auxiliary core, which is controlled by the S/CS for the execution of sequential *Methods* when required. As a complement, repositories for *Attributes* and *Premises/Conditions/Methods* (P/C/M) are also defined, which are memory regions responsible for storing the data and instructions that correspond to each of the respective NOP entities. A set of memory-mapped I/O interfaces is also defined in order to allow NOCA interfacing with external systems.

The *Attribute* processor uses the communication channel that connects it to the *Premise* hardware processes to send a notification when an *Attribute* changes. All the hardware processes responsible for *Premise* processing are able to consume this notification simultaneously. However, it will be effectively processed only by the subset of processes to which *Premises*, whose causal-logical calculations depend on that notification, are allocated. In case the logical value of any *Premise* is changed, the corresponding process notifies the hardware processes responsible for *Condition* processing, which consume this notification in case they depend on that *Premise*. Similarly, if the logical value of any *Condition* is changed, the corresponding process notifies the hardware processes responsible for *Method* processing, which consume this notification in case they depend on that *Condition*, and execute the operation defined by the corresponding *Method*.

In addition, the Scheduler/Conflict Solver module also consumes notifications sent through all the notification channels mentioned, allowing it to execute the conflict resolution tests and internal processes as well as instruction allocation and deallocation to the hardware processes responsible for instruction execution. This allocation and deallocation mechanism allows the execution of NOP programs with more entities than the number of available hardware processes, thereby enabling the scalability of applications, as stated on NOCA requirements.

B. NOCA INSTRUCTION SET ARCHITECTURE (ISA)

The Instruction Set Architecture (ISA) of the NOCA, which is based on its logical model, has the following features:

1) Definition of instructions corresponding to *Premises* (PREMISE-OP), *Conditions* (CONDITION-OP), and *Methods* (METHOD-OP). Each of these instructions defines the sources (i.e., other NOCA instructions) from which it receives notifications (according to NOP logical model), its corresponding operation (relational, logical or logical-arithmetical, respectively) and

a list of other NOCA instructions that it can notify (according to NOP logical model). METHOD-OP instructions can optionally contain the address of another METHOD-OP instruction corresponding to a dependent *Method.*

2) There is a specific instruction for an *Attribute* (ATTRIBUTE-DECL). This instruction is stored in the repository of *Attributes* and keeps the value of the *Attribute* as well as defines configuration data relative to its role in the notification mechanism (for example, whether it is notifying or not) and the list of *Premises* to be notified when the *Attribute* value changes.
3) Additionally, an instruction is defined for triggering a von Neumann method execution (METHOD-VN-OP). This instruction is executed by the S/CS, which sends a signal for the von Neumann core to start the method execution (METHOD-VN-OP). This instruction is executed by the S/CS, which sends a signal for the von Neumann core to start the method execution.

Table 1 summarizes the instruction set. *NP* is the number of notified *Premises, NC* is the number of notified *Conditions, NM* is the number of notified *Methods,* and *NDM* is the number of notified dependent *Methods.*

TABLE 1. NOCA instructions.

Instruction	Description	Template (bits)
ATTRIBUTE-DECL	Declaration of an *Attribute*	32*(2 + NP)
PREMISE-OP	Definition of a *Premise*	32*(3 + NC)
CONDITION-OP	Definition of a *Condition*	32*(3 + NM)
METHOD-OP	Definition of a *Method*	32*(5 + NDM)
METHOD-VN-OP	Definition of a von Neumann *Method* trigger	64

C. NOCA MICROARCHITECTURE

From the design assumptions and logical model previously presented, a microarchitecture organization for the NOCA is proposed, as described in the next sections.

1) MICROARCHITECTURE GRANULARITY

The fundaments of the NOP execution model imply the potentially parallel propagation of notifications among the notification chain elements (*Attributes, Premises, Conditions, Actions, Instigations,* and *Methods*) that are connected for implementation of the causal logic. That is, it is possible that elements of different types in the notification chain are running their operations simultaneously at any given instant in time.

Each notification chain element has its behavior executed by a specific processing unit, exclusively allocated for the execution of this task, and able to be reallocated for executing other elements when necessary. These units are specialized according to the operation type, in order to make their hardware simpler and thus facilitate higher degrees of scalability for a certain hardware platform, besides hierarchizing them to simplify the interconnections for notification propagation. Indeed, a unit that executes a certain type *T* of NOP element needs to be interconnected only to those units that generate notifications for type *T* or that receive notifications generated by type *T.*

In summary, the NOCA defines a fine-grained microarchitecture with specialized processors that are dedicated to executing one instruction at a time. The following types of specialized processing units are defined:

1) PP (*Premise Processor*): processing unit responsible for executing the logical calculation of a PREMISE-OP instruction.
2) CP (*Condition Processor*): processing unit responsible for executing the logical calculation of CONDITION-OP instruction.
3) MP (*Method Processor*): processing unit responsible for executing the operation of a METHOD-OP instruction upon notification of a *true* logic value of a *Condition* (performed by a CONDITION-OP instruction).

2) MICROARCHITECTURE OVERVIEW

Fig. 5 presents the NOCA microarchitecture overview. The sets of PPs, CPs, and MPs are shown in the middle, with the respective connections. The S/CS is connected to each processor of each set via an allocation path (bus), allowing this module to perform allocation and deallocation operations to/from each processor when necessary.

The upper part of Fig. 5 shows the memory subsystem, which is divided into 3 modules: the NOP *Attributes* memory, the NOP *Premises/Conditions/Methods* memory, and the Startup and von Neumann memory. The NOP *Attributes* memory is responsible for storing the application fact base. The NOP *Premises/Conditions/Methods* memory is responsible for storing the application *Rule* base,which is the set of *Premises, Conditions*, and *Methods.* The Startup and von Neumann memory block are accessed by the von Neumann core for the execution of startup routines and sequential von Neumann methods triggered by the S/CS.

Fig. 5 also shows the notification activator on the left side. It corresponds to a set of I/O peripherals that are able to update *Attribute* values, through the *Attribute* R/W path, and thus initiate a new notification cycle (see Section “I/O Interfaces” for further details).

3) INTERCONNECTION AMONG PROCESSING UNITS

In order to implement the communication channels presented in the logical model (Fig. 4) and implemented by the microarchitecture (Fig. 5), the interface between every two interconnected hierarchical groups of processing units (*Attribute* repository/memory to PPs, PPs to CPs, and CPs to MPs) is performed by means of a single notification propagation bus. Although this interconnection organization could theoretically lead to scalability, reusability, and reliability

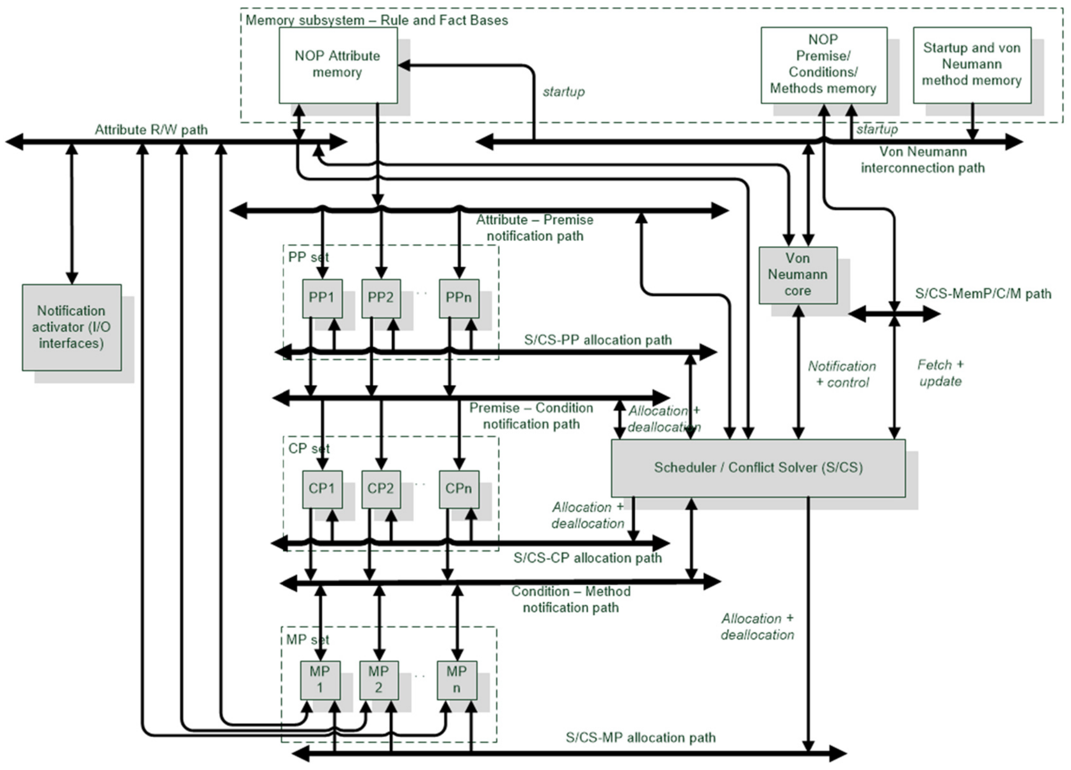

Memory subsystem – Rule and Fact Bases
NOP Attribute memory
startup
NOP Premise/ Conditions/ Methods memory
Startup and von Neumann method memory
Attribute R/W path
startup
Von Neumann interconnection path
Attribute – Premise notification path
PP set
PP1
PP2
PPn
Von Neumann core
S/CS-MemP/C/M path
Notification activator (I/O interfaces)
S/CS-PP allocation path
Notification + control
Fetch + update
Premise – Condition notification path
Allocation + deallocation
CP set
CP1
CP2
CPn
Scheduler / Conflict Solver (S/CS)
Allocation + deallocation
S/CS-CP allocation path
Condition – Method notification path
Allocation + deallocation
MP set
MP 1
MP 2
MP n
S/CS-MP allocation path

FIGURE 5. NOCA overview [17].

limitations [35], [36] due to bus contention and electric issues related to signal propagation [21], [37], in practice it simplifies the NOCA design. The mentioned limitations, in their turn, are minimized by the NOP's intrinsic characteristic of relatively low traffic on notification buses, since notifications are only propagated when changes in *Attributes* states occur.

Concerning the bus used by the S/CS to fetch and update instructions during the scheduling process, its contention is also dependent on the amount of propagated notifications, as they trigger the allocation of instructions into the specialized processing units. However, the greater the amount of processing units of each type and the amount of reevaluations of the same logical-causal expression due to the notification dynamics, the smaller tends to be the need for instruction (re)allocation, therefore minimizing contention on the scheduler bus.

For instance, in iterative applications where the same NOP elements are frequently re-notified, the contention is minimized as those NOP elements tend to remain allocated to specific processors for a longer time. This is similar, to some extent, to what is performed in WaveScalar [31] by using wavecaches to group dependent instructions and minimize the latency of execution. However, it is adapted to the NOP dynamics, which triggers changes into the allocated instructions only when incoming notifications demand new instructions to be allocated.

4) MEMORY SYSTEM AND STARTUP PROCESS

In the NOCA, both *Attribute* and P/C/M memories can be partitioned into internal and external memory devices and, optionally, define cache memory levels for access time optimization. The main difference between *Attribute* and P/C/M memories resides in the ability of the former to send notifications to PPs and to the S/CS, via a specific interface, when the value of a notifying *Attribute* is changed.

The NOCA also defines a startup and von Neumann memory space, as shown in Fig. 5. It comprises a non-volatile memory, to store the startup code and von Neumann methods, as well as a RAM working memory to store data used by these methods (stack and global variables). The von Neumann methods are sequential routines executed by the von Neumann core, when triggered by the METHOD-VN-OP execution, which are optional parts of a NOP application that can be better executed according to sequential logic.

The startup routines are sequential methods, executed by the von Neumann core upon NOCA start, with the purpose of initializing the *Attribute* and P/C/M memories. The initialization includes: copying the *Attribute* and P/C/M instructions to the corresponding memory spaces; propagating the default values of *Attributes* to their dependent *Premises* by means of notifications; and propagating the initial logic values of *Premises* to their dependent *Conditions* also by means of notifications.

5) PP (PREMISE PROCESSOR)

Fig. 6 details the structure of a *Premise* processor (PP). Notifications addressed to *Premises* are received via PP-NOTIF_IN interface and have their source addresses (i.e., the address of the *Attribute* whose value change generated the notification) compared to the operand addresses of the allocated *Premise* (i.e., the values of the *Operand* 1 *address* and *Operand* 2 *address* registers).In case the notification source address matches any of the operands, the notification is consumed by executing the relational operation corresponding to the allocated *Premise* (*Operator* register).If the logical result of the executed relational operation is different from the previous value (as tested by the *Comparator* circuit), PP generates a notification via the PP-NOTIF_OUT interface. This notification is filled with the address of the notifying *Premise* (*Premise address* register value) and forwarded to the *Premise-Condition* notification path (bus) to be consumed and processed by the destination *Conditions.*

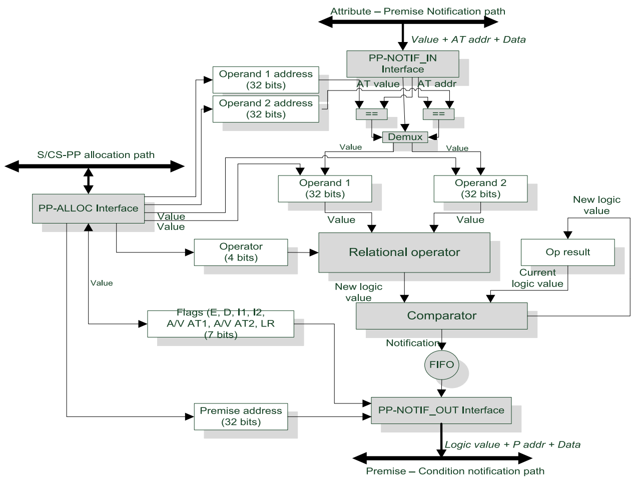


FIGURE 6. **Premise Processor (PP) [17].**

The allocation and deallocation process, in its turn, is performed via the PP-ALLOC interface, which is connected to the S/CS via a specific bus. This process updates the values of the *Operand* 1 *address, Operand* 2 *address, Operand 1, Operand* 2*, Operator, Flags,* and *Premise address* registers.

6) CP (CONDITION PROCESSOR)

Fig. 7 details the structure of a *Condition* processor (CP). Notifications addressed to *Conditions* are received via CP-NOTIF_IN interface and have their source address (i.e., the address of *Premise* or *SubCondition* whose change of logical value generated the notification) compared to the operand (*Premises)* addresses of the allocated *Condition* (values of *Premise* 1 *address* and *Premise* 2 *address* registers). In case the notification source address matches any of the operands, the notification is consumed by executing the logical operation corresponding to the allocated *Condition* (*Operator* register).

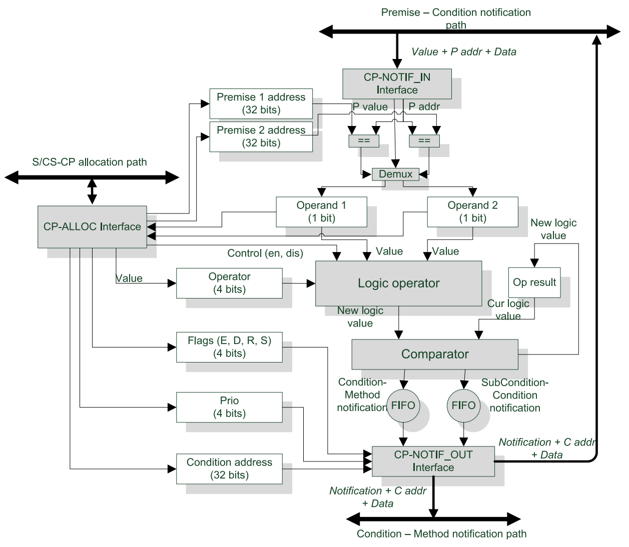


FIGURE 7. **Condition Processor (CP) [17].**

If the logical result of the executed logical operation is different from the previous value (as tested by the *Comparator* circuit), CP generates a notification via the CP-NOTIF_OUT interface. This notification is filled with the address of the notifying *Condition* (*Condition address* register value), with the priority of this *Condition* (*Prio* register value) to be eventually used by the S/CS conflict solver process, and forwarded to the *Condition-Method* notification path (bus) to be consumed and processed by the destination *Methods.* Additionally, the notification is also forwarded back to the *Premise-Condition* path (bus) to be eventually consumed by another *Condition* whose logical calculation depends on the logical value of the notifying *(Sub)Condition.*

The allocation and deallocation process, in a similar way to PPs, is performed via the CP-ALLOC interface, which is connected to the S/CS via a specific bus. This process updates the values of the *Premise* 1 *address, Premise* 2 *address, Operand 1, Operand* 2*, Operator, Flags, Prio,* and *Condition address* registers.

7) MP (METHOD PROCESSOR)

Fig. 8 details the structure of a *Method* processor (MP). Notifications addressed to *Methods* are received via the MP-NOTIF_IN interface and have their source address (i.e., the address of *Conditions* whose change of logical value generated the notification) compared to the address of the *Condition* that triggers the allocated *Method* (value of the *Condition / Master Method address* register). In case the addresses match and the logical value of the *Condition* is true, the notification is consumed by executing the operation defined by the allocated *Method.* The MPs support the execution of the "minimal *Method*" only, which is defined by means of the METHOD-OP instruction with one of the following formats:

$$Res = opn1\ OP\ opn2$$

$$Res = OP\ opn1$$

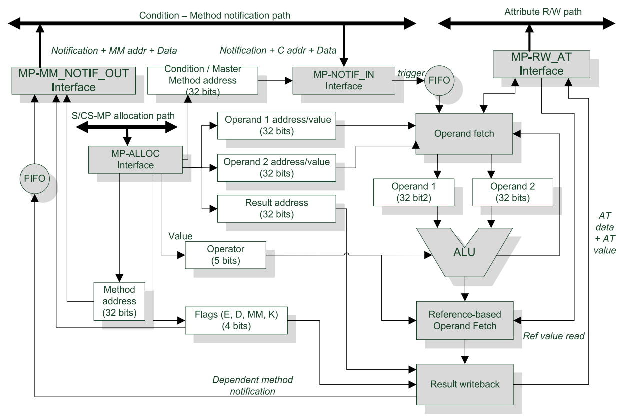

Condition – Method notification path
Attribute R/W path
Notification + MM addr + Data
Notification + C addr + Data
MP-MM_NOTIF_OUT Interface
Condition / Master Method address (32 bits)
MP-NOTIF_IN Interface
trigger
FIFO
MP-RW_AT Interface
S/CS-MP allocation path
Operand 1 address/value (32 bits)
Operand fetch
MP-ALLOC Interface
Operand 2 address/value (32 bits)
FIFO
Operand 1 (32 bit2)
Operand 2 (32 bits)
Result address (32 bits)
AT data + AT value
Value
Operator (5 bits)
ALU
Method address (32 bits)
Flags (E, D, MM, K) (4 bits)
Reference-based Operand Fetch
Ref value read
Dependent method notification
Result writeback

FIGURE 8. **Method Processor (MP) [17].**

In which *opn1* and *opn2* are operands, in the form of *Attributes,* constants or addresses that are fetched from the *Attribute* memory by the *Operand fetch* circuit, when required, via MP-RW_AT interface and *Attribute* R/W path (bus), and stored into the *Operand 1* and *Operand 2* registers. *OP* is the operation defined by the allocated METHOD-OP instruction (*Operator* register), which is used by the Arithmetical-Logical Unit (ALU) to execute the corresponding *Method* operation. If necessary, the *Reference-based Operand fetch* circuit may be used to fetch other data from the *Attribute* memory via the MP-RW_AT interface. *Res,* in its turn, is the address of an *Attribute* that receives the result of the operation by means of a write operation that is performed in the *Attribute* R/W path (bus) by the *Result writeback* circuit. Due to the relative simplicity of the minimal *Method*, the MP hardware is also simple, facilitating its on-scale replication in the hardware device (e.g., FPGA or ASIC) in which the NOCA is implemented.

In case the MM (*Master Method*) flag is on, the MP generates a notification to the *Condition-Method* path (bus) via the MM_NOTIF-OUT interface. This notification is filled with the address of the allocated *Method* (*Method address* register) with the purpose of directly triggering the execution of other *Methods* that depend on it, enabling the execution of a sequence of operations without needing to execute the complete notification cycle.

The allocation and deallocation process, in a similar way to PPs, is performed via the MP-ALLOC interface, which is connected to the S/CS via a specific bus. This process updates the values of the *Condition / Master method address, Operand* 1 *address/value, Operand* 2 *address/value, Result address, Operator, Flags,* and *Method address* registers.

8) S/CS (SCHEDULER/CONFLICT SOLVER)

The S/CS (Scheduler/Conflict solver) is split into 3 subsections, each one responsible for managing the allocation and deallocation of *Premises* (P-Scheduler), *Conditions* (C-Scheduler),and *Methods* (M-Scheduler) via the respective paths (buses) connected to PPs, CPs, and MPs. Each subsection implements a set of allocation and replacement tables used by the allocation and deallocation processes.

Additionally, the subsections share the access to the P/C/M memory for fetching the instructions to be allocated and updating the status of the instructions being deallocated. The whole scheduling process is dynamic and centralized into the S/CS, which is a different approach from architectures such as TIA (Triggered Instruction Architecture), where the set of triggered instructions must be pre-configured to be scheduled into specific processing elements [38].

The processes executed by each of the subsections are triggered by the notifications received through the respective notification paths (buses). The destination addresses of notifications are checked, and the corresponding allocation and deallocation process is started if needed. Particularly for the P-Scheduler, the execution of allocation may involve fetching the value of the second operand, used by the corresponding PREMISE-OP instruction, through the interface that is connected to the Attribute R/W path.

In addition to the instruction allocation and deallocation processes, S/CS executes processes that are responsible for ensuring determinism and solving conflicts among approved *Rules*. The determinism process is triggered by specific flags defined by the ISA with the purpose of ensuring that a certain set of instructions, which are dependent on the same notification, have their execution started simultaneously as the notification arrives. To do so, S/CS must provide for the allocation of all the instructions of the deterministic set being considered into their respective processing units, before enabling them to process the incoming notification. The conflict solver process, in its turn, is executed when two or more *Conditions* marked as conflicting are approved (i.e., change their logical value to ''true''). In this case, S/CS shall choose one of the approved *Conditions* to be the conflict winner, by evaluating a priority value that is defined in the CONDITION-OP instruction opcode. The notification generated by the winner *Condition* is propagated, whereas the notifications originated from the other conflicting *Conditions* are ignored.

Some S/CS operations need to have their flow of execution controlled. When a PP/CP/MP is busy (i. e. under allocation process or still processing a previous notification), it signals the S/CS via a proper bit. When this happens, the busy PP/CP/MP does not consume any incoming notification that is directed to it. Instead, the S/CS detects this condition and enqueues the notification into a specific FIFO for that bus because it snoops every notification bus. So, it can be later reissued when the PP/CP/MP is ready again, which is called a ''delayed notification'' protocol. If some of the S/CS delayed notification FIFOs gets full, the S/CS holds the notification generators, so the enqueued delayed notifications can be delivered, and the FIFO gets able to hold further incoming notifications.

S/CS also executes the METHOD-VN-OP instructions in the M-Scheduler. This causes the triggering of a notification to the von Neumann core, which is responsible for executing the corresponding von Neumann *Method*.

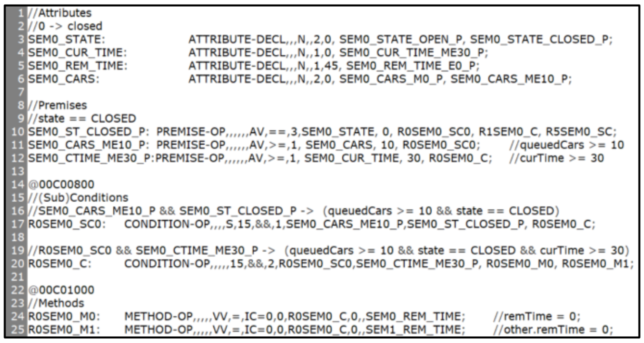
9) I/O INTERFACES

The notification activator (I/O interfaces), shown in Fig. 5, corresponds to a generic set of memory-mapped peripherals that can be used to control specific I/O interfaces. The control registers for these peripherals are memory-mapped, allowing the "minimal Method" implementation supported by MPs as the instruction METHOD-OP (see Section "MP (Method Processor)") to be also used to read/write from/to these peripherals. The read/write operations can be performed as direct accesses to the registers' memory addresses, as well as reads/writes to/ from *Attributes* that are mapped to these addresses.

The peripheral devices that implement the input interfaces can also be programmed to start a notification cycle. This is achieved by declaring a common *Attribute* and configuring the input peripheral to directly update this *Attribute*, allowing generating notifications to the connected *Premises* and thus similarly emulating the hardware interrupt mechanism that is commonly implemented by processors based on the von Neumann model.

D. BUILDING AN APPLICATION AND EXECUTING IT IN THE NOCA

Fig. 9 presents an example of an assembly for a NOCA application, corresponding to the NOP *Rule* presented in Fig. 2. The corresponding NOP high-level code, shown in Fig. 10, can be written directly into a text editor or generated from the NOP high-level language (NOPL) by the NOP compiler [16].

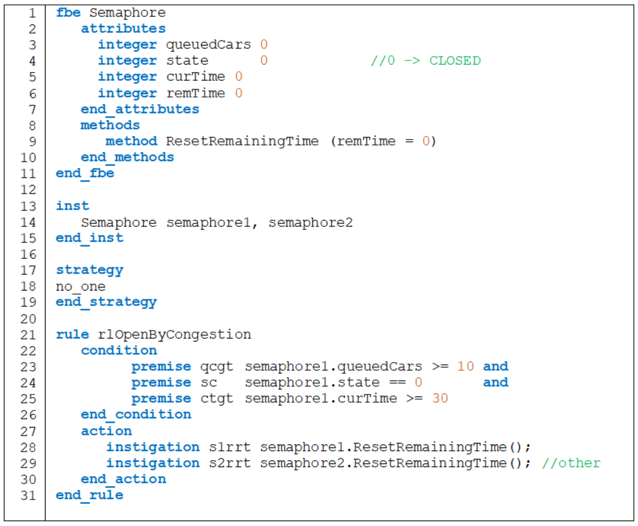


```
//Attributes
//0 -> closed
SEM0_STATE:             ATTRIBUTE-DECL,,,N,,2,0, SEM0_STATE_OPEN_P, SEM0_STATE_CLOSED_P;
SEM0_CUR_TIME:          ATTRIBUTE-DECL,,,N,,1,0, SEM0_CUR_TIME_ME30_P;
SEM0_REM_TIME:          ATTRIBUTE-DECL,,,N,,1,45, SEM0_REM_TIME_E0_P;
SEM0_CARS:              ATTRIBUTE-DECL,,,N,,2,0, SEM0_CARS_M0_P, SEM0_CARS_ME10_P;

//Premises
//state == CLOSED
SEM0_ST_CLOSED_P:  PREMISE-OP,,,,,,AV,==,3,SEM0_STATE, 0, R0SEM0_SC0, R1SEM0_C, R5SEM0_SC;
SEM0_CARS_ME10_P:  PREMISE-OP,,,,,,AV,>=,1, SEM0_CARS, 10, R0SEM0_SC0;       //queuedCars >= 10
SEM0_CTIME_ME30_P:PREMISE-OP,,,,,,AV,>=,1, SEM0_CUR_TIME, 30, R0SEM0_C;    //curTime >= 30

@00C00800
//(Sub)Conditions
//SEM0_CARS_ME10_P && SEM0_ST_CLOSED_P ->  (queuedCars >= 10 && state == CLOSED)
R0SEM0_SC0:    CONDITION-OP,,,,S,15,&&,1,SEM0_CARS_ME10_P,SEM0_ST_CLOSED_P, R0SEM0_C;

//R0SEM0_SC0 && SEM0_CTIME_ME30_P ->  (queuedCars >= 10 && state == CLOSED && curTime >= 30)
R0SEM0_C:      CONDITION-OP,,,,,15,&&,2,R0SEM0_SC0,SEM0_CTIME_ME30_P, R0SEM0_M0, R0SEM0_M1;

@00C01000
//Methods
R0SEM0_M0:     METHOD-OP,,,,,VV,=,IC=0,0,R0SEM0_C,0,,SEM0_REM_TIME;      //remTime = 0;
R0SEM0_M1:     METHOD-OP,,,,,VV,=,IC=0,0,R0SEM0_C,0,,SEM1_REM_TIME;      //other.remTime = 0;
```

FIGURE 9. NOCA assembly example.

The assembly syntax is compatible with a prototypical assembler that was developed as a support tool. The binary code for each instruction is generated from the assembly code. The instructions are allocated in contiguous addresses, starting from a base address defined as an assembler command-line option or redefined by means of the @ tag, such as the declaration @00C00800 shown in Fig. 9. This allows creating memory allocation sections that are useful, for example, for debugging purposes.

Each instruction of the NOCA ISA defines a set of memory references corresponding to the instructions from which they receive notifications and also to the instructions to which they send notifications when necessary. These references can be encoded as numeric addresses or as instruction labels (such as SEM0_STATE_CLOSED_P and R0SEM0_M0 in Fig. 9. The labels are replaced by the corresponding memory addresses during the assembler processing.



```
fbe Semaphore
   attributes
     integer queuedCars 0
     integer state      0           //0 -> CLOSED
     integer curTime 0
     integer remTime 0
   end_attributes
   methods
     method ResetRemainingTime (remTime = 0)
   end_methods
end_fbe

inst
   Semaphore semaphore1, semaphore2
end_inst

strategy
no_one
end_strategy

rule rlOpenByCongestion
   condition
         premise qcgt semaphore1.queuedCars >= 10 and
         premise sc   semaphore1.state == 0       and
         premise ctgt semaphore1.curTime >= 30
   end_condition
   action
       instigation s1rrt semaphore1.ResetRemainingTime();
       instigation s2rrt semaphore2.ResetRemainingTime(); //other
   end_action
end_rule
```

FIGURE 10. NOPL code example.

The base addresses for PREMISE-OP, CONDITION-OP, METHOD-OP, and METHOD-VN-OP instruction allocation shall be contained into the address space configured to be used by the P/C/M Memory Control. The base address for ATTRIBUTE-DECL instruction allocation, in its turn, shall be contained into the address space configured to be used by the *Attribute* Memory Control.

The execution of a NOP application by the NOCA involves the following operations:

1) Execution of the startup routines (presented in Section "Memory System and Startup Process") by the von Neumann core, in order to initialize the *Attribute* and P/C/M memories. This initialization includes: copying the instructions related to *Attributes* and P/C/M to the respective memory spaces; propagating the default value of each *Attribute* to its dependent *Premises,* by means of notifications; and propagating the initial logical value of each *Premise* to their dependent *Conditions,* also by means of notifications.
2) Update the value of an *Attribute* to start the notification process, which can be performed by the startup code or by the notification activator (Section "I/O Interfaces"). This update causes the propagation of a notification to the PPs and to the S/CS via the *Attribute-Premise* notification path (bus).The S/CS fetches the PREMISE-OP instructions referenced by the *Attribute* from the P/C/M memory and allocates them at the PPs, in case they are not already allocated.
3) Every concerned PP executes its PREMISE-OP instruction and eventually generates a notification

to the CPs and to the S/CS via the *Premise-Condition* notification path (bus). The S/CS fetches the CONDITION-OP instructions referenced by the notifying *Premise* from the P/C/M memory and allocates them at the CPs, in case they are not already allocated.

4) Every concerned CP executes its CONDITION-OP instruction and eventually generates a notification to the MPs and to the S/CS via the *Condition-Method* notification path (bus). The S/CS fetches the METHOD-OP instructions referenced by the notifying *Condition* from the P/C/M memory and allocates them at the MPs, in case they are not already allocated. Additionally, the S/CS can trigger the von Neumann core in case the CONDITION-OP references a METHOD-VN-OP instruction, causing the execution of a sequential von Neumann method referenced from the memory address contained in this instruction.
5) Every concerned MP executes its METHOD-OP instruction if the corresponding *Condition* notified "true". The execution of a METHOD-OP instruction updates the value of an *Attribute* in memory (via Attribute R/W path), eventually starting a new notification cycle.

Given the detailed descriptions for the NOCA and its application building process, the next section shows a case of study that was designed and experimented in order to test and evaluate the NOCA's characteristics. For this purpose, a NOCA processor was first implemented as a prototype in FPGA and subsequently in the form of a simulator [39].

V. CASE OF STUDY

A case of study was developed to allow evaluation of the NOCA. It aimed at evaluating the NOCA performance at different scales of parallelization (i.e., the number of PPs, CPs, and MPs) by means of a simulator (see Section B). This included not only the speedup and variations in execution time but also the variations in contention and percentage of utilization for some of the NOCA internal buses.

As these are preliminary experimentation data, which are used not only for this performance comparison but also for testing and debugging the NOCA prototypes, the case of study was kept limited enough. This was done in order not to introduce uncertainties related to the software behavior, which could interfere with evaluating the behavior and correctness of the prototype itself.

A. SOFTWARE IMPLEMENTATION

The case of study consists of simulating the activation and reading of a set of sensors. Each sensor is represented by an entity that defines the following Boolean *Attributes*: *isRead,* which indicates whether the sensor has already been read; and *activated,* indicating whether the sensor is active or inactive.

The application *Rules* (presented as an example *Rule* named Rule_ReadSensor on Fig. 11) consist of changing the *isRead Attribute* to "TRUE" as the following *Conditions* are met: the sensor is active, and it has not been read yet.

```
//Generic rule for the sensor reading control.
rule Rule_ReadSensor
   condition
      premise Sensor.isRead == false and
      premise Sensor.activated == true
   action
      instigation Sensor.mtRead();
      instigation Sensor.mtDeactivateSensor();
      instigation Counter.mtIncrementCounter();
//Rule for restart control.
rule Rule_Counter
   condition
      premise Counter.count == LIMIT_OF_ACTIVATED_RULES
   action
      instigation Counter.mtRestartCounter();
```

FIGURE 11. Rules defined for the case of study.

The example of *Rule* Rule_ReadSensor, as shown in Fig. 11, is instantiated for each of the N sensors (the figure shows its instantiation for sensor number 1).

Thus, the execution dynamics consist of running M iterations, at which N sensors have their states changed to "active". In case the active sensor has not been read yet, i.e., *isRead* equals to "FALSE", the corresponding *Rule* is activated and the sensor is read. The number of active sensors is changed for every experiment, changing the number of activated *Rules* as a consequence.

In addition, this application defines another entity responsible for counting the sensors that have already been read on the current iteration. This entity defines the *Attribute count,* storing the value to be atomically incremented at every sensor read. When the value of *count* reaches the limit of activated *Rules* (i.e., the total of sensors to be read defined by the LIMIT_OF_ACTIVATED_RULES constant), the *Attributes* are reinitialized and a new iteration is executed. This *Rule* is shown in Fig. 11 as Rule_Counter.

The application for the case of study was implemented using the NOP high-level language (as exemplified in Fig. 10) and further compiled by the NOP compiler to generate NOCA assembly (as exemplified in Fig. 9).

In terms of NOCA assembly, every sensor *Attribute*, as well as the *count Attribute* of the counting entity, is directly mapped to *Attributes*, by using the declaration instruction ATTRIBUTE-DECL. The application *Rules* are mapped to *Premises* by using PREMISE-OP, *Conditions* by using CONDITION-OP, and *Methods* by using METHOD-OP, naturally respecting their corresponding logical relations.

B. EXECUTION PLATFORM

The previous experiments presented in [34] were performed on a NOCA prototype implemented in FPGA. This imposed some scale limitations (maximum of 4 processors of each type) due to the relatively high complexity of the synthesized hardware, therefore limiting the replication of processing units due to the limited amount of hardware resources provided by the prototyping platform.

Because of the scale limitations imposed by the FPGA prototype, a NOCA simulator called NOCASim [39] was proposed. This simulator allows the execution of a NOP application into a NOCA processor with a larger parallelization scale than the FPGA prototype, provided that the multiple processing units can be replicated by means of multiple software instances (i.e., object instances).

NOCASim is a software simulator for a specific NOCA implementation, which aims at simulating the execution of NOP software, built according to the NOCA ISA, similar to the NOCA prototype implemented in FPGA [39]. The internal structure of NOCASim is modularized according to NOCA logical model (Fig. 4), by means of a set of classes and objects that represent each of the logical blocks and their corresponding implementation in an object-oriented language (Java). This allows multiple instances of the classes that represent the processing units (PPs, CPs, and MPs) to be created, as previously mentioned.

The simulation occurs on a clock-by-clock basis and allows visualizing the NOCASim internal states as well as generating log data for further processing. Every clock updates the internal states of the NOCASim in a way similar to the updates on the digital circuits that compose the NOCA prototype implemented in FPGA. This similarity was validated by means of unit tests, where single instances of the NOCA logical blocks were simulated, and the resulting timings were compared to the corresponding timings on the same modules of the NOCA prototype in FPGA [39].

The NOCASim used for the case of study is run on a laptop with a Core i7 4500U processor (2 physical cores), 8 GB of RAM memory, and Windows 10 installed as OS.

C. RESULTS

For executing the experiment, the percent of activated *Rules* (i.e., activated sensors) and the number of processing units are changed for every run. The performance data (i.e., number of clock cycles) and percent of bus contention are then measured and obtained from the NOCASim log data. The experiment instantiates 100 sensors, which are activated for every run in percentages starting from 10% up to 100%, in steps of 10%. The number of processing units of each type (PPs, CPs, and MPs) is varied from 4 to 8, 16, 32, 64, 128, 256, and 512.

Table 2 shows the number of NOP elements that are processed as a function of the percent of *Rules* activated for this application. Fig. 12 presents a chart with the performance data for every percent of activated *Rules*, as a function of the number of processing units of each type. This chart shows that, as the number of processing units increases, the NOCA performance also increases. This happens because an ever-increasing number of instructions remains allocated on the respective processing units, thus depending on a smaller number of (de)allocations and on a smaller number of P/C/M memory accesses as a consequence. As the number of processing units becomes greater than the number of NOP elements to be processed, the performance is no longer increased.

Fig. 13 presents the calculated speedup as the number of processing units is multiplied by 2. This chart shows that the NOCA presents speedup rates up to 2.5 as the number of processing units exceeds the number of processed NOP elements. From this point on, doubling the number of processing units does not present any significant performance gains. As stated before, this happens because, as the number of processing units equals the number of NOP elements, there is no more need to fetch instructions from P/C/M memory as the instructions remain allocated into their respective processing units.

Fig. 14 presents the percentage of P/C/M memory bus utilization (sum of the percentages in states different from IDLE) as a function of the number of processing units of each type and the percentage of activated *Rules*. The percentage

TABLE 2. Number of NOP elements as a function of the percent of activated *Rules*.

% of activated Rules	Number of Premises	Number of Conditions	Number of Methods
10%	21	11	31
20%	41	21	61
30%	61	31	91
40%	81	41	121
50%	101	51	151
60%	121	61	181
70%	141	71	211
80%	161	81	241
90%	181	91	271
100%	201	101	301

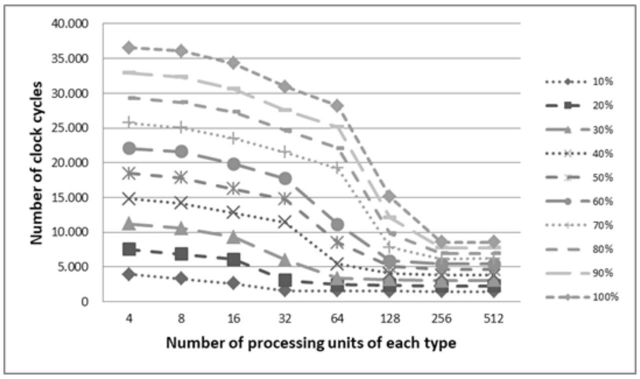


FIGURE 12. Performance chart for different percentages of activated Rules as a function of the number of processing units [39].

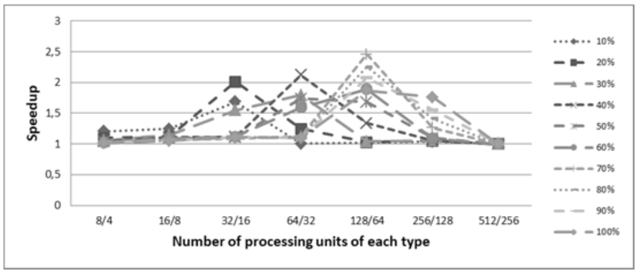


FIGURE 13. Speedup as the number of processing units is multiplied by 2 [39].

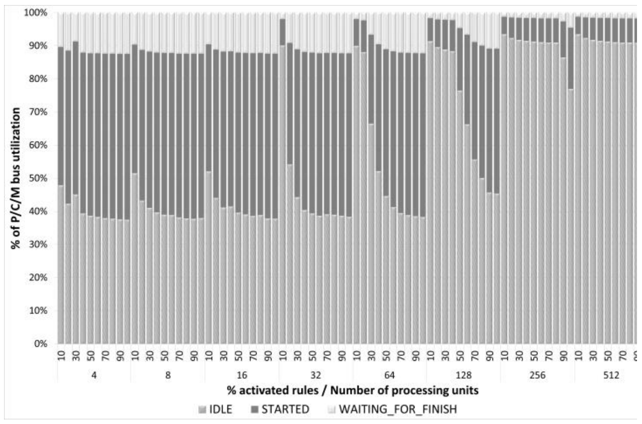


FIGURE 14. Percentage of P/C/M bus utilization [39].

of bus utilization decreases as the number of processing units increases. This happens because, as the number of processing units increases, there is a decrease in the number of instruction (de)allocations on the PPs, CPs and MPs that would occur as the *Rules* are re-executed on the multiple iterations of the case of study.

D. DISCUSSION

The execution of the case study into the NOCASim allows experimenting with a larger scale of parallelization, as replicating the NOCA processing units (PP, CP and MP) depends only on replicating their respective instances in software. Thus, it is possible to reduce the limitations in the scale of parallelization that were imposed by the previous implementation in FPGA [34].

This experiment shows that the scale of parallelization effectively influences the execution performance, as increasing the number of processing units decreased the execution time. However, as the number of processing units is equal to or greater than the number of equivalent NOP elements to be processed, the performance is no more increased. This is due to the fact that the basic NOP elements (*Premises, Conditions* and *Methods*) cannot have their internal execution parallelized (i.e., they define the minimum granularity level), thus imposing a maximum level of parallelization for a given NOP application. Moreover, despite the NOP being intrinsically parallel, the NOCA logical model itself imposes some sequentialization due to characteristics such as bus concurrency, need for instruction allocation by the Scheduler, memory accesses, and initialization of a NOCA application performed by routines running on a von Neumann core.

Regarding the bus activity, the experiment shows that the notification buses remain relatively idle even with the increase on the number of processing units and the increase on the average number of executed *Rules* (and the increase on the frequency of notifications, as a consequence) which occurs as a function of the speedup. This can be interpreted as evidence that the NOP eliminates redundancies (i.e., unnecessary processing and the consequent flow of notifications) due to its execution model. The same does not happen to the *Attribute* memory bus because the number of *Attribute* accesses increases at the same rate as the number of executed *Rules* increases and the MP model requires *Attribute* memory accesses for every *Method* execution (not a load/store model).

Inversely, as the number of processing units increases, the P/C/M memory bus utilization decreases. This happens because the NOCA makes use of temporal locality principles, mainly when the number of processing units is equal to or greater than the number of NOP elements to be re-processed. In this scenario, recently allocated instructions remain on the processing units and do not need to be re-fetched from memory, thus helping on NOCA's performance improvement as the number of processing units is increased.

Additionally, the decreasing impact of P/C/M memory access on the overall NOCA performance, as the scale of parallelization increases, was shown by means of an extra experiment, where the memory latency was reduced from three cycles to one. By reducing the memory latency to one third and keeping a small number of processing units of each type, the NOCA performance was increased by 30%. However, as the scale of parallelization increased, this performance gain was not observed anymore. This happens because, as the scale of parallelization increases beyond a certain limit, every NOP element of the tested application remains allocated to a specific processing unit and, thus, the number of P/C/M memory accesses for allocation and deallocation tends to zero.

VI. NOCA VERSUS CURRENT ARCHITECTURES

As a main distinctive characteristic, the NOCA has been developed to be fully compatible with the NOP execution model. Therefore, it has been conceived from scratch to execute rule-based software without the need for an inference engine, which is conceptually different from what happens under execution models such as von Neumann and dataflow. However, some of its architectural aspects have been inspired and/or can be compared to previous work regarding architectures targeted to different execution models.

Regarding the execution dynamics, dataflow-based architectures such as SDF [5], WaveScalar [31], and MAD [32] trigger the execution flow by data arrival at the input queues of the computation blocks, so the data are processed regardless they have their values changed or not. This is different from what happens in the NOCA, which processes notifications that are generated only upon data change.

Similarly, the NOCA does not execute programs sequentially as typical architectures based on the von Neumann model such as ARM Cortex-A9 [40] and IA-32 [41] do because NOP does not define a program counter. However, the NOCA benefits from the possibility of sequential execution, by means of the implementation of the concept of dependent *Methods*. This is performed by allowing their respective MPs to notify one another directly. Also, the NOCA can dispatch code that is inherently sequential to an auxiliary von Neumann core.

Regarding the scheduling mechanism, the NOCA works differently from architectures such as TIA [38]. Even though each TIA processing element (PE) has its own scheduler, it schedules only the triggered instructions that have been pre-configured in build time to be scheduled into that PE. The NOCA, on its turn, defines a single scheduler that is capable of scheduling instructions in runtime to any of its specialized processing units, depending only on the compatibility between the instruction type and the processor type. This runtime scheduling allows higher flexibility and scalability on scheduling.

Regarding the caching or fetching of new instructions, WaveScalar [31] caches instructions into the wavecaches by grouping dependent instructions, which minimizes latency of execution as consequence. The NOCA also keeps the instructions cached into its specialized processing units because it is not necessary to replace them with other instructions immediately if there are no incoming notifications to be processed by different instructions. However, the dynamics is different and the NOCA does not depend on a compile-time mapping for grouping instructions together.

With respect to the partitioning and parallelization of execution, the NOCA is a fine-grained architecture with specialized processing units designed to execute simple instructions that map to the NOP logical model. The concept of fine-grained execution (sometimes called ‘‘microthread’’ in the literature) is inspired by architectures such as SDF [5] and SDAARC [1], although they perform their fine-grained execution scheduling based on dataflow rather than the notification flow used by the NOCA.

VII. CONCLUSION AND FUTURE WORK

The proposal of the NOCA is an alternative to conventional parallel computer architectures based on von Neumann or dataflow model since the NOCA was developed so as to totally adhere to the NOP, whose execution model is based on notifications. Particularly, this allows the NOCA to take advantage of the parallel execution characteristics coming from the structure of NOP applications.

The NOCA is a flexible and generic computer architecture. Flexibility and generality are achieved due to the fact that, distinctly from previous implementations of execution platforms for NOP software, the NOCA allows software development based on a flexible process. That is, the software is fetched from a program memory, thereby making it editable and flexible. Still, the hardware consists of a fixed processor implementation in terms of internal architecture, thereby being generic for the execution of any NOP application. The experimented case of study shows that NOCA can present significant speedup even for relatively simple NOP applications, such as the one experimented, for scales of parallelization in the order of tenths of processors or more. This happens because NOP software executed by the NOCA is able to efficiently exploit the availability of multiple processors due to the relatively low granularity of a NOP application.

The experiment also shows that the P/C/M memory bus utilization decreases significantly as the scale of parallelization increases and the re-execution of *Rules* does not demand a great amount of P/C/M memory accesses for instruction allocation and deallocation, provided that the entities that compose these *Rules* (*Premises, Conditions,* and *Methods*) remain allocated on the respective processors. This characteristic, combined with the hierarchization of instructions to be executed by specific processors, define a new caching level in the NOCA, which takes advantage of temporal locality (i.e., elements that are frequently re-notified tend to remain allocated to a certain processor when they receive a new notification).

As observed in the case of study, in a scenario where a set of logic/causal processing elements (*Rules*) needs to be repeatedly re-evaluated and re-executed, this caching characteristic eliminates part of the effects of the communication bottleneck between the processors and the memories. Such a bottleneck is still a problem on modern architectures based on von Neumann principles [25], despite the efforts to provide support for parallelization in order to increase performance. In these terms, the possibility of retaining the current instructions into the processors while they are not required to execute new instructions is an innovative aspect of NOCA's execution model when compared to the von Neumann's execution model.

It is worth mentioning that, even in cases where the application size (in the amount of NOP entities) is greater than the number of available processing units, the amount of P/C/M memory access can be relatively low. This can happen in situations where the application being executed defines different use cases that activate only subsets of *Rules,* therefore not requiring all the NOP entities to be necessarily allocated at least once during the application execution.

However, the use of a single memory space for storing *Premises/Conditions/Methods* can become a bottleneck when the utilization of the corresponding bus by the S/CS is relatively high. This is particularly true in situations in which the relatively small parallelization scale requires frequent (de)allocations of instructions from/to the processing units since they depend on read or write accesses to the P/C/M memory.

Although the *Attribute* and P/C/M memories implement a cache level to improve their access performance, in practice, this improvement is hampered by the lack of spatial locality of a NOP application since the execution flow is not necessarily sequential across adjacent instructions in memory. Therefore, it is necessary to further examine the memory access dynamics of the NOCA in order to propose a caching strategy that is more adequate to its notification model.

Finally, the case of study also shows that the utilization rate for the notification buses is not high, even when adding more processors as producers and consumers of notifications. Thus, the frequency of notifications tends not to be a bottleneck in increasing the parallelization efficiency, at least for the amounts of hundreds of processors of each type.

Besides the case of study that is presented on this paper, it is worth mentioning that NOP applications, running on NOCA, have been previously compared with the same applications under IP approach and running on a von Neumann platform. These comparisons are presented in [17] and [34] and show that the NOP with NOCA can outperform IP with von Neumann in cases where the IP execution demands a high number of logical-causal processing, despite the performed experiments have not been exhaustive and have not yet been executed in adequate scale of parallelization due to the limitations of the former FPGA prototype.

Future research may focus on the following topics: investigate the effects of increasing the NOCA parallelization scale with respect to the parallelization efficiency; perform further comparisons between the NOP with NOCA and IP with von Neumann implementations, using test cases with different characteristics (more structural redundancies, predominantly sequential calculation, higher scale of parallelization, among others); investigate the effects of changing some NOCA memory subsystem parameters on its execution performance, mainly the P/C/M cache configuration and memory latency; develop techniques to optimize the NOP assembly code, including the development of an adequate compiler.

As a general result of this research and its implications, better-founded conclusions about the advantages and shortcomings of the NOCA (and of the NOP) can be achieved, allowing a better assessment of the applicability of this technology to the development of systems in general.

REFERENCES

[1] K. Yi, W. W. Ro, and J.-L. Gaudiot, "Importance of coherence protocols with network applications on multicore processors," *IEEE Trans. Comput.*, vol. 62, no. 1, pp. 6–15, Jan. 2013, doi: 10.1109/TC.2011.199.

[2] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, no. 5, pp. 67–77, May 2011, doi: 10.1145/1941487.

[3] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *Proc. 2nd Annu. Symp. Comput. Archit. (ISCA)*, New York, NY, USA, 1975, pp. 126–132, doi: 10.1145/642089.642111.

[4] G. Gupta and G. S. Sohi, "Dataflow execution of sequential imperative programs on multicore architectures," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, New York, NY, USA, 2011, pp. 59–70, doi: 10.1145/2155620.2155628.

[5] K. M. Kavi, R. Giorgi, and J. Arul, "Scheduled dataflow: Execution paradigm, architecture, and performance evaluation," *IEEE Trans. Comput.*, vol. 50, no. 8, pp. 834–846, Aug. 2001, doi: 10.1109/12.947003.

[6] M. Gabbrielli and S. Martini, *Programming Languages: Principles and Paradigms* (Undergraduate Topics in Computer Science), 1st ed. London, U.K.: Springer-Verlag, 2010.

[7] A. F. Ronszcka, R. F. Banaszewski, R. R. Linhares, C. A. Tacla, P. C. Stadzisz, and J. M. Simao, "Notification-oriented and rete network inference: A comparative study," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Hong-Kong, Oct. 2015, pp. 807–814.

[8] J. G. Brookshear, *Computer Science: An Overview*. Reading, MA, USA: Addison-Wesley, 2006.

[9] I. Bell, N. Hasasneh, and C. Jesshope, "Supporting microthread scheduling and synchronisation in CMPs," *Int. J. Parallel Program.*, vol. 34, no. 4, pp. 343–381, Jul. 2006, doi: 10.1007/s10766-006-0017-y.

[10] R. N. Oliveira, V. Roth, A. F. Henzen, J. M. Simão, E. C. G. Wille, and P. Nohama, "Notification oriented paradigm applied to ambient assisted living tool," *IEEE Latin Amer. Trans.*, vol. 16, no. 2, pp. 647–653, Feb. 2018.

[11] J. M. Simão and P. C. Stadzisz, "Inference based on notifications: A holonic metamodel applied to control issues," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 39, no. 1, pp. 238–250, Jan. 2009, doi: 10.1109/TSMCA.2008.2006371.

[12] J. M. Simão, R. F. Banaszewski, C. A. Tacla, P. C. Stadzisz, "Notification oriented paradigm (NOP) and imperative paradigm: A comparative study," *J. Softw. Eng. Appl.*, vol. 5, no. 6, pp. 402–416, 2012, doi: 10.4236/jsea.2012.59083.

[13] R. Kerschbaumer, R. R. Linhares, J. M. Simão, P. C. Stadzisz, and C. R. Erig Lima, "Notification-oriented paradigm to implement digital hardware," *J. Circuits, Syst. Comput.*, vol. 27, no. 8, Apr. 2018, Art. no. 1850124, doi: 10.1142/S0218126618501244.

[14] F. Schütz, J. A. Fabro, A. F. Ronszcka, P. C. Stadzisz, J. M. Simão, "Proposal of a declarative and parallelizable artificial neural network using the notification-oriented paradigm," in *Neural Computing and Applications*. London, U.K.: Springer, Jun. 2018. [Online]. Available: https://rdcu.be/S5DJ

[15] W. R. M. Barretto, A. C. B. K. Vendramin, J. M. Simão, "Notification oriented paradigm for distributed systems," in *Proc. Computer Beach (COTB)*, Florianópolis, Brazil, 2018, pp. 110–119.

[16] A. F. Ronszcka, C. A. Ferreira, P. C. Stadzisz, J. A. Fabro, and J. M. Simão, "Notification-oriented programming language and compiler," in *Proc. 7th SBESC-Brazilian Symp. Comput. Syst. Eng.*, Curitiba, Brazil, vol. 10, Nov. 2017, pp. 125–131.

[17] R. R. Linhares, "Contribution to development of a computing architecture proper to the notification oriented paradigm," Ph.D. dissertation, UTFPR, Curitiba, Brazil, 2015.

[18] D. L. Belmonte, R. R. Linhares, P. C. Stadzisz, and J. M. Simão, "A new method for dynamic balancing of workload and scalability in multicore systems," *IEEE Latin Amer. Trans.*, vol. 14, no. 7, pp. 3335–3344, Jul. 2016, doi: 10.1109/TLA.2016.7587639.

[19] C. L. Forgy, "RETE: A fast algorithm for the many pattern/many object pattern match problem," *Artif. Intell.*, vol. 19, pp. 17–37, Sep. 1982, doi: 10.1016/0004-3702(82)90020-0.

[20] E. Peters, R. P. Jasinski, V. A. Pedroni, and J. M. Simao, "A new hardware coprocessor for accelerating notification-oriented applications," in *Proc. Int. Conf. Field-Program. Technol.*, Seoul, South Korea, Dec. 2012, pp. 257–260.

[21] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 4th ed. San Francisco, CA, USA: Morgan Kaufmann, 2009.

[22] R. Bianchini and B.-H. Lim, "Evaluating the performance of multithreading and prefetching in multiprocessors," *J. Parallel Distrib. Comput.*, vol. 37, no. 1, pp. 83–97, Aug. 1996, doi: 10.1006/jpdc.1996.0109.

[23] T. Ungerer, B. Robič, and J. Šilc, "A survey of processors with explicit multithreading," *ACM Comput. Surv.*, vol. 35, no. 1, pp. 29–63, Mar. 2003, doi: 10.1145/641865.641867.

[24] C. Shin, S.-W. Lee, and J.-L. Gaudiot, "Adaptive dynamic thread scheduling for simultaneous multithreaded architectures with a detector thread," *J. Parallel Distrib. Comput.*, vol. 66, no. 10, pp. 1304–1321, Oct. 2006, doi: 10.1016/j.jpdc.2006.06.003.

[25] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, and W. Dally, "ExaScale computing study: Technology challenges in achieving exascale systems," DARPA-IPTO, Arlington County, VA, USA, Tech. Rep. TR-2008-13, Sep. 2008, Accessed: Feb. 7, 2012. [Online]. Available: http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf

[26] J. Backus, "Can programming be liberated from the von neumann style?: A functional style and its algebra of programs," *Commun. ACM*, vol. 21, no. 8, pp. 613–641, Aug. 1978, doi: 10.1145/359576.359579.

[27] S. A. McKee, "Reflections on the memory wall," in *Proc. 1st Conf. Comput. Frontiers Comput. Frontiers (CF)*, New York, NY, USA, 2004, p. 162, doi: 10.1145/977091.977115.

[28] T. Ungerer, J. Silc, and B. Robic, "Asynchrony in parallel computing: From dataflow to multithreading," *J. Parallel Distrib. Comput. Practices*, vol. 1, pp. 1–33, Mar. 1998.

[29] G. M. Papadopoulos, *Implementation of a General Purpose Dataflow Multiprocessor* (Electrical Engineering). Cambridge, MA, USA: MIT Press, 1988.

[30] R. S. Nikhil and K. K. Pingali, "I-structures: Data structures for parallel computing," *ACM Trans. Program. Lang. Syst.*, vol. 11, no. 4, pp. 598–632, 1989, doi: 10.1145/69558.69562.

[31] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers, "The WaveScalar architecture," *ACM Trans. Comput. Syst.*, vol. 25, no. 2, May 2007, Art. no. 4, doi: 10.1145/1233307.1233308.

[32] C.-H. Ho, S. J. Kim, and K. Sankaralingam, "Efficient execution of memory access phases using dataflow specialization," in *Proc. 42nd Annu. Int. Symp. Comput. Archit. (ISCA)*, New York, NY, USA, 2015, pp. 118–130, doi: 10.1145/2749469.2750390.

[33] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, T. Yuba, "An architecture of a dataflow single chip processor," in *Proc. 16th Annu. Int. Symp. Comput. Archit. (ISCA)*, New York, NY, USA, 1989, pp. 46–53, doi: 10.1145/74925.74931.

[34] R. Ribeiro Linhares, J. M. Simao, and P. C. Stadzisz, "NOCA—A notification-oriented computer architecture," *IEEE Latin Amer. Trans.*, vol. 13, no. 5, pp. 1593–1604, May 2015, doi: 10.1109/tla.2015.7112020.

[35] B. Yang, L. Guang, T. Säntti, and J. Plosila, "Mapping multiple applications with unbounded and bounded number of cores on many-core networks-on-chip," *Microprocess. Microsyst.*, vol. 37, nos. 4–5, pp. 460–471, Jun. 2013, doi: 10.1016/j.micpro.2012.08.005.

[36] M. Baklouti, Y. Aydi, P. Marquet, J. L. Dekeyser, and M. Abid, "Scalable mpNoC for massively parallel systems—Design and implementation on FPGA," *J. Syst. Archit.*, vol. 56, no. 7, pp. 278–292, Jul. 2010, doi: 10.1016/j.sysarc.2010.04.001.

[37] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Comput. Surv.*, vol. 38, no. 1, p. 1, Jun. 2006, doi: 10.1145/1132952.1132953.

[38] A. Parashar, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, J. Emer, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, and M. Gambhir, "Triggered instructions: A control paradigm for spatially-programmed architectures," in *Proc. 40th Annu. Int. Symp. Comput. Archit. (ISCA)*, New York, NY, USA, 2013, pp. 142–153, doi: 10.1145/2485922.2485935.

[39] L. F. Pordeus, "Simulation of a computing architecture proper to the notification oriented paradigm," M.S. thesis, UTFPR, Curitiba, Brazil, 2017.

[40] ARM (Advanced RISC Machines Limited). (2012). *Cortex-A9 Revision r4p1 Technical Reference Manual*. Accessed: Jan. 8, 2019. [Online]. Available: https://static.docs.arm.com/ddi0388/i/DDI0388I_cortex_a9_r4p1_trm.pdf?_ga=2.126603341.1791039363.1546966039-601212789.1546966039

[41] Intel Corporation. (2012). *Intel 64 and IA-32 Architectures Software Developers Manual*. Accessed: Jan. 8, 2019. [Online]. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[42] F. Eschmann, B. Klauer, R. Moore, and K. Waldschmidt, "SDAARC: An extended cache-only memory architecture," *IEEE Micro*, vol. 22, no. 3, pp. 62–70, May 2002.

ROBSON R. LINHARES was born in Curitiba, Brazil, in 1975. He received the B.Sc. degree in electrical engineering, with emphasis on electronics and telecommunications from the Federal University of Technology - Paraná (UTFPR), Brazil, which is formerly known as Federal Center for Technological Education of Parana, in 1999, and the M.Sc. degree in software engineering for real-time systems and the Ph.D. degree in computer engineering from the Graduate Program in Electrical Engineering and Industrial Informatics (CPGEI), UTFPR, in 2001 and 2015, respectively.

He is currently a Professor with UTFPR, teaching in a set of educational programs such as the undergraduate courses of Computer Engineering and Information Systems and the Graduate Program in Applied Computing (PPGCA). His research interests include software engineering, programming languages and paradigms, embedded systems, and computer architectures.

LEONARDO F. PORDEUS was born in Curitiba, Brazil, in 1990. He received the B.Sc. degree in electrical engineering, with emphasis on electronics and telecommunications from the Federal University of Technology - Paraná (UTFPR), Brazil, in 2014, the M.Sc. degree in computer engineering from the Graduate Program in Electrical Engineering and Industrial Informatics (CPGEI), UTFPR, in 2017, where he is currently pursuing the Ph.D. degree in computer engineering. His research interests include software and system engineering, programming languages and paradigms, reconfigurable logic, and computer architectures.

JEAN M. SIMÃO was born in Ponta Grossa, Brazil, in 1976. He received the B.Sc. degree in computer science from the State University of Ponta Grossa (UEPG), in 1998, the M.Sc. degree from the Graduate School in Electrical Engineering and Industrial Computer Science (CPGEI), University of Technology - Paraná (UTFPR), in 2001, and the Ph.D. degree, after a double thesis, in the domains of industrial computer science from the CPGEI/UTFPR, the Computer Engineering & Automatics, Research Center for Automatic Control of Nancy (CRAN), Henry Poincaré University (UHP), and the University of Lorraine (UL), France, in 2005. Subsequently, in 2005 and 2006, he developed teaching and research activities in a Master at UHP and at CRAN in a Postdoctoral Context.

Since August 2006, he has been with UTFPR, where he is currently a Professor with the Department of Informatics, Centre of Technology Innovation (CITEC), and CPGEI. Still, his teaching activities concern computer science, whereas his research ones include artificial intelligence, software/system engineering, and development/computer paradigm, namely the notification oriented paradigm (NOP) of his authorship.

PAULO C. STADZISZ was born in Blumenau, Brazil, in 1963. He received the Diploma degree in data processing technologies from the Federal University of Paraná (UFPR), Brazil, in 1987, the M.Sc. degree in industrial computer science from the Graduate Program in Electrical Engineering and Industrial Computer Science (CPGEI), Federal Center for Technological Education of Parana (CEFET-PR), in 1990, and the Ph.D. degree from the Franche-Comté University, France, in 1997.

He is currently a Professor with the Federal University of Technology - Paraná (UTFPR) teaching in a set of educational programs. He namely teaches and coordinates researches in the CPGEI/UTFPR doctoral program associated to CNPq (National Council for Scientific and Technological Development) and CAPES (Coordination for the Improvement of Higher Education Personnel) of Brazil. He also coordinates the Laboratory of Intelligent Manufacturing Systems (LSIP) and the Laboratory of Innovation in Technology (LIT), UTFPR. His publications and researches include systems modeling and analysis, renewable energy, and software engineering.