Extending the Hardware Architecture of Systemic Computation to a Complete Programming Platform

Christos Sakellariou, Peter J. Bentley
Department of Computer Science
University College London
London, UK
c.sakellariou@cs.ucl.ac.uk, p.bentley@cs.ucl.ac.uk

***Abstract*—Systemic Computation is an unconventional paradigm which defines a model of natural behavior and implies a massively parallel computer architecture. It is designed to be a computational paradigm for natural systems and processes modeling. Existing software implementations have been too limited in terms of performance, flexibility and programmability. This paper solves key problems that remained in earlier work, introduced towards the first practical hardware Systemic Computation implementation using FPGAs. This is achieved by making various optimizations and software additions, resulting in a complete and efficient SC programming platform.**

Keywords—systemic computation; programming platform; FPGA; optimization; TCAM

I. INTRODUCTION

In the past century, continuous advancements in the fields of electronics and computer science have made modern computers extremely powerful computing machines. The vast majority of their architectures are loyal to the conventional von Neumann design which has now reached well over half a century in age. Unfortunately, nature is organized in an inherently different manner, without any sense of program or data memory and certainly without a sequential state machine traversing through well-defined states. In order to more efficiently model natural systems, the serial, deterministic and centralized design of conventional computers should be replaced with a parallel, stochastic and distributed architecture.

Systemic Computation [1] (SC) addresses this problem by defining any natural model as a pool of systems, leaving the level of abstraction to the user, acknowledging the importance of the interactions among systems in modeling their behavior. It uses the notion of scope to constrain and organize stochastic interacting systems, mirroring the dynamics and hierarchical organization of nature.

A proof-of-concept software implementation is provided in [1] along with introducing SC. The functionality of the paradigm was extensively investigated in [2], where a more high-level approach was used to model a genetic algorithm, artificial neural networks and artificial organisms to exploit SC properties as self-adaptation, fault-tolerance, self-organization. These first implementations had a crucial role validating SC but were not designed to take performance into consideration. They were simulating a systemic computer and any notion of parallelism. To address this, a GPU-based implementation was presented in [3], taking advantage of the parallel resources of a commercial GPU resulting in great performance gains. Yet, this implementation was too limited as its functionality was restricted to the demonstrated applications, since it supported only hardcoded software extensions. However, using a GPU revealed the potential of a design with a hardware constituent, overcoming the limited parallelism provided by modern CPUs.

Combining the desirable performance gains due to its inherent parallelism with the flexibility of designing a custom architecture, the core of a hardware SC implementation based on an FPGA was introduced in [4]. This so-called Hardware Architecture of Systemic computation (HAoS) supports a limited instruction set through dedicated hardware resources. It was also suggested that the instruction set may be expanded, depending on the user applications requirements, through high-level user-defined plugins to run on an optional CPU. An investigation on the implementation of the HAoS-CPU communication interface is given in [5].

However, in order to provide a practical SC programming platform, four key issues still need to be addressed:

- efficient random selection
- efficient schemata matching
- efficient I/O
- good programmability

These problems are not unique to HAoS - most bio-inspired hardware architectures (whether based on the cell, evolution, or neural networks) will also face similar problems as parallel, stochastic computation inevitably involves one of the operations: random selection, string matching, high-speed I/O and tools to ensure ease of programming. This paper uses SC as the exemplar for optimizations for these key problems.

By addressing these issues, this paper presents the evolution of HAoS to a practical programming platform. Section II provides additional background, section III gives an overview of SC while section IV outlines our novel architecture, as it was introduced in [4] and [5]. Section V details the enhancements to our initial design and section VI suggests an intuitive HAoS programming methodology. Section VII discusses our results while section VIII concludes the paper.

This work is sponsored by the EPSRC and Toumaz UK Limited.

II. BACKGROUND

SC was inspired by natural behavior and addresses the problem of modeling natural systems efficiently attempting to become a gateway to Natural Computation [6]. An extended overview of alternative paradigms tackling the same problem is given in [7]. Indicative examples are Constrained Generating Procedures (CGPs) [8] (finite state machines that analyze complex systems by reducing them in mechanisms and constraints of interactions), π-calculus [9] (a process calculus adapted for biological systems simulation) and Petri nets [10] (a graphical tool and mathematical theory for concurrent processes). Moreover, alternative novel architectures attempt to inherently support natural properties. Examples of such unconventional architectures are the Perplexus platform [11], a computing substrate made of reconfigurable devices endowed with bio-inspired capabilities, RALA [12] based on a lattice of reconfigurable logic automata cells asynchronously passing state tokens corresponding to physicals resources and spiNNaker [13] featuring a vast number of independent ARM-based multiprocessors to simulate real-time neuron activity.

There is also literature that relates to the four issues addressed in this paper. Random selection logic (RSL) is used to randomly select a candidate out of a qualifying subset of a pool of options, e.g. parent selection in genetic algorithm implementations [14] or register shuffling for cryptographic applications [15]. Schemata (or similarly template) matching has been previously used in the design of adaptive systems [16] for FPGA context switching (comparing the contents of a shift register with sequentially retrieved templates from a RAM) and also in hardware Artificial Immune Systems matching nonself strings [17]. Efficient on-chip communication is extensively discussed in [18] and the authors conclude that the use of FIFO-based computing nodes is preferred for point-to-point communication. The AXI4 protocol is suggested as the most preferable solution in [19] for memory-mapped register-based control interface, and for this reason is adopted in HAoS. Finally, [19] suggests a programming methodology which is used in numerous embedded applications [20]. In our work this is combined with the HAoS software framework and the updated hardware design to result in the first practical SC programming platform.

III. SYSTEMIC COMPUTATION OVERVIEW

SC emphasizes the dynamics among the elements of a system and its environment in the form of systemic interactions and thus uses a holistic approach to their modeling. In contrast with the conventional sequence of instructions executed in a CPU, SC relies on events (interactions) occurring in a parallel and stochastic fashion and defines a set of conventions [1] for every natural model: (a) everything is a system, (b) systems may comprise or share other nested systems, (c) systems can be transformed but never destroyed or created from nothing, (d) interaction between systems may cause transformation of those systems according to a contextual system, (e) all systems can potentially act as context and interact in some context, (f) the transformation of systems is constrained by the scope of systems, and finally (g) computation is transformation.

Each system is represented as a binary string with three parts, two "schemata" and one function. Inspired by nature, the notion of scope is used in [1], to define the interaction bounds of a system. Interactions between two systems are defined by a third "contextual" system (or context) which sets the transformation of any, none or both the interacting systems through the interaction (see Fig. 1c). This pair of systems along with the interaction-defining context form a valid interacting triplet. The context through its two schemata provides templates which are matched against all systems, in the scope it belongs to, to define a valid triplet. This implies the two main tasks involved in any SC interaction: (1) finding valid triplets (context and two randomly-chosen systems matching its templates in a shared scope) and (2) updating the two interacting systems according to the transformation function of the context. SC is further detailed in [1].

The graphical representation of a system in SC is illustrated in Fig. 1a. The two receptors of a system hold its two schemata while the transformation function is given in the middle. Three types of systems are supported: data systems, which do not define interactions hence their transformation function is always zero, context systems, defining interactions between data systems, and context adapters which define transformations between contexts or contexts and data systems. Fig. 1d illustrates an accumulator SC program using graphical notations. Given a pool of data systems and a context which places the sum of the data of the two interacting systems to one of them and zeroes the other one, and given enough time, only one non-zero system will hold the sum of all data values while the rest will be zero.

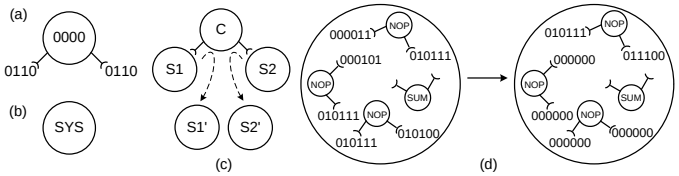

Figure 1. Graphical SC notation: (a) a data system revealing its binary contents; its transformation function is zero (b) alternative notation for a data system called SYS (c) Systems S1 and S2 interact according to the function of the context C; the notation may optionally include the resulting systems S1' and S2' (d) Sum operation on a pool of data systems (NOP: No Operation)

IV. THE HARDWARE ARCHITECTURE OF SYSTEMIC COMPUTATION (HAOS)

The design of our *H*ardware *A*rchitecture *o*f *S*ystemic computation (HAoS) was presented in [4] and [5], and we showed in simulation that it can form the basis for an FPGA-based platform that can outperform prior SC implementations (software and GPU-based). An overview of the design follows.

A. HAoS Base Architecture

The HAoS base architecture [4] is shown in Fig. 2. The CORE contains the various memories to hold the SC program binary information and logic to support the parallel schemata matching (see next section). The Control Unit (CU), named in a conventional way, handles the execution of the SC program and the communication with the optional CPU. The REG BANK includes configuration and data registers which form a control and debug interface between HAoS and the CPU. Basic

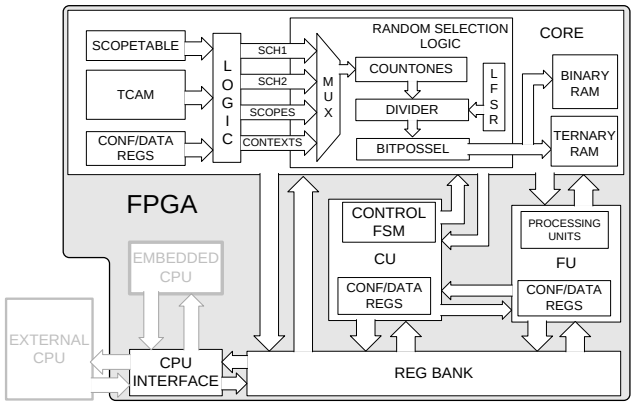

Figure 2. The base SC Hardware Architecture

local processing is provided by the Functional Unit (FU) as a limited instruction set is supported to minimize data transfers between the REG BANK and the CPU.

The role of the CPU is to allow for more complex high-level functions by letting the user define, when needed, new instructions. The SC compiler [1], translates SC source code in SC assembly. The CPU is also used to load the assembly into the memory elements of the CORE during initialization. Following the analysis in [5], it was concluded that an embedded soft CPU is sufficient in this prototype stage, as it can minimize communication latency and achieve adequate bandwidth. However, communicating data during CPU-handled functions was handled in a naive way in our initial design. Thus, HAoS-CPU I/O handling is revised in this paper.

B. HAoS Core

Valid triplet generation was the main limitation of prior software-based implementations. Commonly, three systems (one of them being a context) were randomly selected in a given scope and then the operand systems were matched against the schemata of the context. Identifying valid triplet with this approach was quite inefficient, especially for SC programs with a big number of systems and a small number of valid triplets. HAoS identifies valid triplets using an inherently parallel approach, using a Ternary Content Addressable Memory (TCAM) [21].

CAMs are essentially parallel comparators, providing all addresses that match their compare input in parallel. Furthermore, TCAMs can also perform ternary comparisons, identifying partial matches using "don't care" bits. This guarantees finding all, if any, systems matching the schemata of a context while a pseudo-random select circuit is used to nominate those to form valid triplets. Efficient schemata matching, and in extent the choice and implementation method of the TCAM, is vital for SC, being the mechanism identifying interacting systems. Thus, its design is refined in this paper as it has a great impact on the overall performance of HAoS.

As seen in Fig. 2, apart from the TCAM, the CORE (further detailed in [4]) consists of the system memories (storing the binary and ternary parts of systems), the scopetable memories, various status registers and the Random Selection Logic (RSL). The RSL returns the address of a (pseudo-)randomly selected set bit of its input bus. Its inputs provide candidates for random selection during all the stages of the SC program (see next section). It consists of COUNTONES which counts the set bits of a bus, a maximal-length Linear Feedback Shift Register (LFSR) for pseudo-random number generation, a combinatorial divider and a module (BITPOSSEL) that given a bus and the rank of one of its set bits (the position of the set bit with rank 1 is 2 in 01110*1*01), it returns its position. The random selection works as follows: a random number, from the LFSR, is divided by the sum of the set bits of the bus (from COUNTONES). The remainder of this division is used as the rank of the random set bit that is given to BITPOSSEL in order to identify its position. The RSL, being mostly a combinatorial circuit in [4], is also refined here to increase the overall operating frequency.

C. HAoS Control Flow

The control flow of a typical SC program is shown and described in Fig. 3. Adhering to the law of conservation of energy in nature, systems are never destroyed but only transformed. This is reflected in the computation infinite loop, although the program may halt when no further interactions are possible, as the state of the system cannot further change.

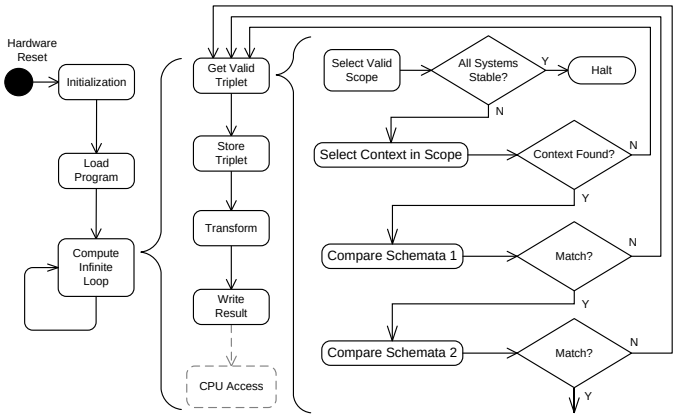

Figure 3. SC Program Control Flow : HAoS enters an infinite computation loop, after the SC program is loaded, which involves finding valid triplets and transforming the selected systems. A hard reset initializes the system and the program is loaded on HAoS memories. Each iteration comprises four main and an optional step. At first, a valid triplet is identified by randomly selecting a scope, a context in this scope and two systems matching the schemata of this context. Then, the selected systems are retrieved, they transform through their interaction and the changed systems are updated and stored back to local memory. Optionally, at the end, the user may pause execution to extract debug information. In the case of a context mismatch (any of its schemata does not match any system in the scope), it gets disabled to prevent future mismatches. Moreover, if a scope contains no valid contexts, it also gets disabled until a new system is added to it. The program halts if all scopes have been disabled.

V. Optimizations and Enhancements

The four key issues are addressed below with various solutions which are hardware optimizations and software enhancements made to the initial design to increase its performance and make it more user-friendly and flexible.

ISSUE 1: The Random Selection Logic is too slow

Like many nature-inspired architectures, randomness is fundamental to SC. The random selection logic is in the critical path of the base design. Improving the efficiency of the RSL would therefore be a tremendous advantage. Here we describe

optimizations designed to increase the overall operating frequency by 4 times, by carefully balanced pipelining and decrease the area utilization by sharing resources.

SOLUTION 1.1: Resource sharing

The BITPOSSEL module of the RSL, combined a parallel bit count with a branchless selection method. The bit count is used to provide partial sums which are then appropriately masked and passed through a barrel-shifter. As a result, we obtain the position of a bit with a given rank in the input bus. As seen in Fig. 4, the COUTNONES and BITPOSSEL modules of the RSL are now merged. The parallel sum-of-bits counter in COUNTONES is reused for the generation of the partial sums during the identification of the position of the selected bit.

SOLUTION 1.2: Design a constant-latency barrel-shifter

The length of the barrel shifter is equal to the size of the longest input bus to the RSL, which is in turn equal to the number of maximum supported systems. Thus, when this number is increased, the number of logic levels required for the barrel shifter implementation have a considerable impact to the delay along the critical path. For this reason, the conventional barrel shifter is replaced with a parallelized and pipelined version. This instead uses an array of multiplexers with registered pre-shifted (by the required pre-calculated number of bits) versions of only the possible subset of shifting combinations of the input buses. While this results in a slightly higher resource utilization as the number of maximum supported systems increases, it provides the ability to minimize its latency and moreover make it independent of the maximum number of systems, resulting in deterministic performance.

SOLUTION 1.3: Balanced pipelining

Moreover, rather than registering the inputs of the RSL or the output of the input selection multiplexer (Fig. 4), the output of the adder-tree that counts the set bits is registered, reducing the number of registers required to pipeline this stage from *Number of RSL Input Buses* x *Input Bus Length* to the length of the maximum sum of set bits (for 1024 maximum supported systems : from 5x1024=5120 registers down to just 11).

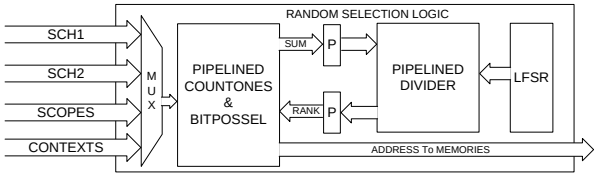

Figure 4. The Revised RSL module. P is a pipeline register. COUNTONES and BITPOSSEL modules have been merged to share the adder tree. The RSL has been pipelined where the data buses have minimim length having in mind the trade-off between minimizing latency and excessive resource utilization.

The module with the higher impact in the critical path delay of the RSL, however, is the combinatorial divider. Thus, the divider is now pipelined (one-level deep with registered input and outputs). The level of pipelining throughout the RSL was fine-tuned to match the critical path outside the RSL and it was decided that a latency of 20ns (50MHz) was adequate for the prototype, as deeper pipelining, although possible, would require considerable changes in the control logic and would affect resource utilization in order to achieve timing closure.

ISSUE 2: The schemata-matching logic latency is high

Nature-based hardware architectures have commonly used CAMs for efficient string-matching [17]. The performance of the schemata-matching logic is crucial for HAoS as well. The TCAM of our initial architecture had a high latency when being written. Here, we revise its design to always have a single-clock latency.

SOLUTION 2.1: Select a TCAM design with one clock latency

Standard FPGA CAM design techniques include registered-based, RAM-based and Look-Up Table based approaches [21]. Moreover, Xilinx provides an reference design which combines the LUT technique with the optimized shift-register blocks (SRL16E) found in its FPGAs [21]. Although RAM-based CAMs are the most efficient in terms of resource utilization [21], they do not support the ternary mode required for partial schemata matching in SC. The initial HAoS design used the suggested by Xilinx SRL16E-based approach which, according to [21], provides efficiency in terms of the trade-off between required area and operating frequency. However, since this design was effectively constructed by a chain of parallel 16-bit shift registers, that implied that each write operation, shifting data in, one bit at a time, required 16 clock cycles. It was noticed, that as the number of entries for the TCAM increased, reflecting the number of supported systems, for deep TCAM implementations (>128 entries) the area footprint was comparable to the one of the simple register-based design (5%-15% area overhead depending on size) with similar operating frequency. Since the TCAM is written every time a system is altered during an interaction, replacing the SRL16E-based TCAM with an array of registers and comparators, provided single-clock read and write operation, saving 15 or 30 clock cycles for interactions changing one or both systems.

ISSUE 3: CPU I/O is inefficient

Hi-speed I/O is vital for any modern computing system. Based on latency and bandwidth, the investigation in [5] suggested that on-chip interconnect is preferable to alternatives, as it accomplishes data transfers at wire speed. However, HAoS initially did not fully exploit this fact. Here, we present various optimizations which minimize communication and enable faster and fewer CPU (read and write) data accesses.

SOLUTION 3.1: Hardware handles interaction result writing

HAoS makes available to the user all the fields of an active triplet since those that will be used during an interaction depend on the transformation function. In the initial design all this user data are read from and written back to HAoS directly by the CPU. Looking for a more efficient way, in order to avoid the extra overhead on the software side, the CPU would directly just write the changed systems to HAoS registers and writing the memories would be handled efficiently in hardware. However, since writing a triplet to the memories is performed in one clock cycle, to reduce latency, all user data would have to be updated when a change was made. Enabling the option of independently writing parts of the triplet would greatly increase the control logic complexity and the required area footprint.

SOLUTION 3.2: Write-detection minimizes CPU writes

To address this issue, a write-detection mechanism was devised, inspired by the “dirty-bit” scheme commonly used in

page replacement and data cache design [22]. As mentioned above, since all user data are available in the beginning of an interaction, the user may read only the parts of the triplet that are going to be used in his custom function. The great enhancement comes when writing back the transformed triplet.

Each field of the transformed triplet is now associated with a write-detection flag. This flag array is reset when an interaction is assigned to the CPU. The address of the registers that hold each individual field of the transformed triplet (see Fig. 5) is already given in the predetermined memory map of the CPU (the memory management subsystem of the CPU accesses the REG BANK as any other memory location). While in "Transform" state (see Fig. 3), when each such field is altered by the CPU, the decoded write address from the memory subsystem is matched against each field address and sets its respective flag. At the end of the state, the active triplet (user data before interaction) is copied to the transformed triplet (user data after interaction) address space, updating at the same time only the fields that were actually changed by the CPU. Using this relatively simple write-detection approach, the need of accessing individual fields when writing the triplet is avoided, preserving the low area footprint of the HAoS memories writing logic, but also minimizing the required user accesses to enable the write-back of the interaction result.

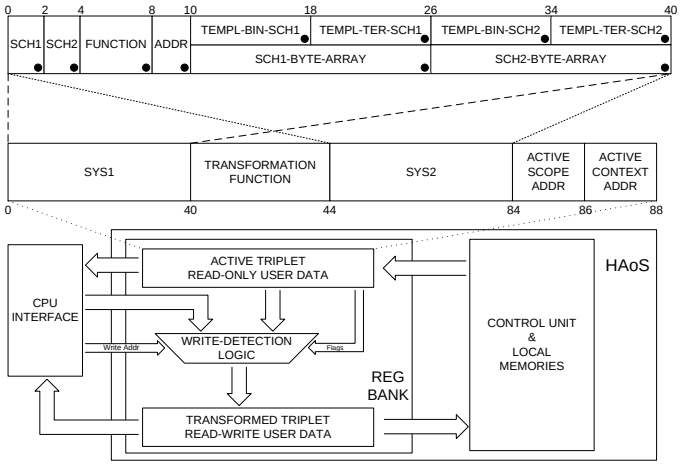

Figure 5. Revised Triplet Memory Map and Write-Detection Mechanism. Upper part: the revised registers organization for each system in a triplet is shown along with the sizes (in bytes) for each field. Fields (left to right): data system schemata 1 & 2, transformation function, system address, binary and ternary parts for each schemata of a context share the same address space with a byte-array formatted version of the respective schemata of a data system. All fields (with a dot) have an associated write-detection flag, set when a field is modified. Middle part: the two systems, the active interaction function and the scope and context addresses form the user data. Bottom part: when writing-back the transformed triplet, the write address from the CPU is used to update only those fields that have actually been changed while the rest are copied over from the local copy (active triplet), minimizing CPU I/O operations.

*SOLUTION 3.3: Array-formatted and byte-aligned schemata*s

In order to further minimize user effort while manipulating the HAoS user data, taking into consideration that each, 16-bit in this implementation, SC schema may be used as a whole (e.g. a 16-bit number) or as a bit-array (e.g. a 16-element chromosome), each schema can be accessed (read/written) in both modes (2-byte value or 16-byte array with one effective bit for each byte). This saves time-consuming bit-manipulation through bit-masking by operating on an array and also avoids bit-to-byte conversions, now handled by hardware. Moreover, parts of a compressed template of a context were rearranged to get a more compact memory utilization while their respective registers in the REG BANK were also re-arranged to account for any compiler byte-alignment restrictions.

ISSUE 4: Lack of programmability

Ease of programmability is imperative for any computing platform. The initial design, being a hardware prototype, lacked any software tools of a practical programming platform. This section describes the enhancements which provide HAoS programmers with all the necessary tools to easily develop SC models. This software framework enables quick model verification using a high-level language, efficient debugging and a comprehensive API, abstracting low-level details. Here we approach the challenge of programmability gradually. We answer to the needs of a HAoS programmer trying to simulate his first SC model.

SOLUTION 4.1: Ease model functionality verification

Since the SC programming language, defined in [1], and the SC compiler are in place, the model can be written and compiled. The programmer should have a tool to quickly verify its functional behavior, a debugger. Thus, in order to avoid time-consuming low-level system simulations and expedite SC models development, a software-based HAoS simulator functionally equivalent to our platform was built. It provides a software interface for the programmer to develop abstract high-level interaction behaviors, similar to the (C/C++) plug-in approach in [2].

SOLUTION 4.2: Minimize program loading time

Once the model is verified, it should be loaded to HAoS. A post-compiler tool was developed to reduce the assembly code to binary format, minimizing the amount of data to be transferred to HAoS and the processing time during program loading. The programmer then uses a storage device to transfer the binary file into HAoS. The Compact Flash Card (a common feature for FPGA development boards) was selected for this purpose, rather than the on-chip Block RAM or the off-chip RAM, in order to make HAoS a standalone platform. The CF card, when FAT-formatted, it can support a basic file system using the XilFatfs library [20].

SOLUTION 4.3: Enable efficient logging

During the SC program execution, the programmer will also need to access log runtime information. Conveniently, the CF card can be used for this efficiently. Although access to a real-time console is possible during live hardware debugging, just the data-logging overhead can account for the majority of the run time as all text is communicated to a computer though a high-latency UART channel. Storing log data locally drastically reduces runtime.

SOLUTION 4.4: A driver handles low-level functionality while the programmer is provided with a comprehensive API

The programmer will also require a CPU to execute his high-level sequential functions. As the available development board was the Xilinx ML605, selecting the MicroBlaze processor was a natural choice, due to its compatibility with the

design tools. An operating system would probably be a useful tool for the programmer. However, all SC high-level interaction processing runs as a bare-metal application, as an operating system would heavily impact performance. Therefore, a low-level driver was developed to handle HAoS-MicroBlaze communication. The driver works as follows: at first, it resets HAoS, initializes the communication interfaces and loads the program. When program execution starts, it waits for an interrupt, by constantly reading a predetermined HAoS status register, to either pass control to the user code to perform some high-level interaction or halt the program in case no further interactions can or need to be executed. In the end, it also optionally gives some useful statistics. All these processes are transparent to the programmer, who only has to define the transformation functions in the SC source code to be executed on the CPU. However, he/she will need an interface between the driver and his SC "application". Thus the driver is complemented by a basic but comprehensive API, in order to enhance the flexibility of the platform and the accessibility of the programmer to the internal state of HAoS. The features of the API include optimized access to any HAoS memory-mapped control register, the local memories and a high-precision (±10ns) real-time counter. The programmer is now equipped with a complete software programming framework.

VI. A Practical SC Programming Platform

Having presented solutions to the key problems of our initial design, this section completes the revised HAoS architecture and combines it with the developed software framework to suggest a programming methodology and result in the first practical hardware-based Systemic Computation prototype and standalone programming platform.

A. The Complete Platform

Since the soft CPU (MicroBlaze) and the communication interface (AXI4) were selected, the platform was completed with some other useful peripherals (64KB local Block-RAM instruction and data memories, 512MB of external DDR3 memory and other common embedded IP cores). From the processor point of view, HAoS is used just as another peripheral in terms of connectivity and accessibility. A slight modification was required to the IP interconnect (IPIC) logic as the Xilinx AXI4-Lite interface natively supports up to 32 4-byte registers. In order to waive this restriction, the multiplexer-based read/write logic (an input bus with a set bit at the position of the register to be read/written is decoded to get the position of the register) and the simple register-array was replaced by an interface providing the exact access address. This address is then encoded to give access to any set of registers, depending on the size of the data to be accessed.

B. HAoS Programming Methodology

Further focusing on the practical aspect of using the platform, a programming methodology, for developing natural models targeting HAoS, is suggested below and illustrated in Fig. 6 with layers, to separate the distinct development phases.

Assuming that an existing natural system or process needs to be simulated, it is important to first understand its behavioral dynamics and identify its quantitative characteristics in order to conceptualize it (Conceptual Layer). This will aid the systemic analysis which is used to identify the interacting systems, the interactions among them (any contextual behavior defining their transformation function) and their organization (scopes). The SC calculus notation [2], can be used to describe the interactions, while the SC model may be visualized using the SC graphical notation (see Fig. 1). Each element in the SC graph should correspond directly to a specific part of the SC source code. This fact implies that source code extraction from SC graphs can be automated in the future, enabling building SC models by using a high-level SC graph tool. This direct mapping also extends in the SC calculus notation making the transition from the Conceptual Layer to the Application Layer fully automated once these SC high-level tools are developed. The Application and Link Layers form the HAoS software framework. In the Application Layer, the SC source code is translated to SC human-readable assembly code. This may then be used as input to the functionally equivalent to HAoS software program, along with the high-level processing plugins (implied by transformation functions not supported natively by HAoS) until the model behaves as expected. The Link Layer is the back-end phase were the SC binary is generated by the post-compiler and the user code is linked with the HAoS driver (using the Xilinx Software Development Kit) to generate the bare-metal executable to run on the MicroBlaze.

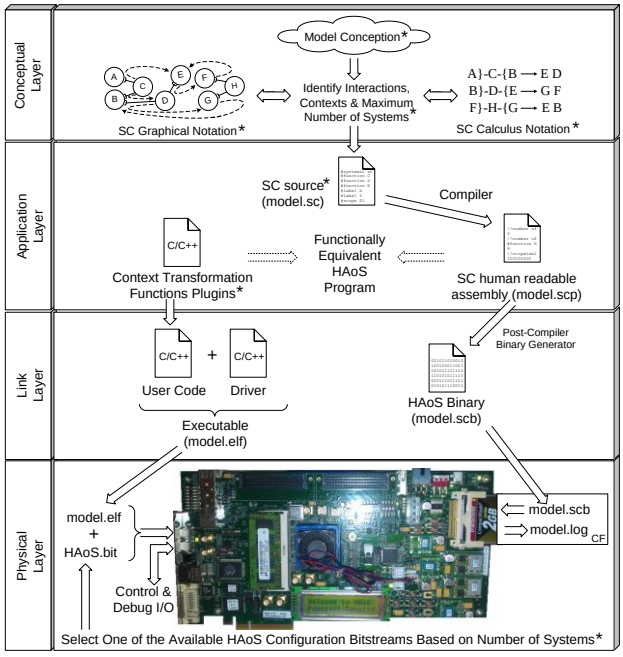

Figure 6. HAoS suggested programming methodology (* implies user input)

At the Physical Layer, the program is loaded to the CF card and HAoS is implemented on the target FPGA board. Based on the number of systems of the SC model, the appropriate configuration bitstream should be selected and combined with the output executable from the Link Layer to form the final bitstream to program the FPGA. A hardware reset starts the simulation. The CF card acts as the storage unit of the platform, storing the HAoS binary program and runtime log information. The whole process is fairly automated. User input is required mainly on the tasks with an asterisk (*) in Fig. 6.

VII. EVALUATION

A. Scalability in the Single-Chip Implementation

Depending on the number of systems required for an SC model, HAoS can be easily scaled to accommodate any number of systems as long as the design can fit on the selected device (assuming a single-FPGA implementation). HAoS has been written in highly-parameterizable VHDL code. Thus, scaling the design is a matter of setting a single parameter, which is proportional to the number of maximum supported systems. In this way, the size of the SC model in terms of systems, is limited solely by the size of the available FPGA. Moreover, the Xilinx ML605 board, features a Virtex-6 LX240T FPGA which is a mid-range 40-nm device, while high-end 28-nm devices can offer nearly 10 times more reprogrammable fabric real estate and significant performance potential. Table I shows the implementation statistics of available variations of the complete HAoS platform, in agreement with the initial estimates in [4]. It is interesting to note that the size of the design scales linearly as the number of systems is doubled, as illustrated in Fig. 7. This implies that, assuming availability (and affordability) of the largest modern FPGA device (Virtex-7 2000T with 305400 slices), SC models with up to 8196 systems may be efficiently modeled with a single FPGA.

TABLE I. HAOS PLATFORM IMPLEMENTATIONS STATISTICS FOR DESIGNS SUPPORTING 64-1024 SYSTEMS BASED ON VIRTEX-6 LX240T

Maximum Systems		**0** **No HAoS**	**64**	**128**	**256**	**512**	**1024**
	Total	***Used (%)***	***Used (%)***	***Used (%)***	***Used (%)***	***Used (%)***	***Used (%)***
Slices	37680	6841 (18)	13492 (35)	15525 (41)	18269 (48)	24882 (66)	34522 (91)
Slice LUTs	150720	14283 (9)	29972 (19)	34338 (22)	43146 (28)	61481 (40)	98511 (65)
Slice Registers	301440	15061 (4)	25400 (8)	30818 (10)	41727 (13)	63768 (21)	108361 (35)
I/O Blocks	600	193 (32)	193 (32)	193 (32)	193 (32)	193 (32)	193 (32)
RAMs	416	56 (13)	61 (14)	64 (15)	70 (16)	106 (25)	148 (35)

HAoS performance will be identical for configurations of different sizes. However, fine-tuning the size of the design for a particular application may permit more functions to be hardware-accelerated, avoiding offloading all computation to the CPU, further increasing overall performance (e.g. obtaining here random numbers from the LFSR rather than the CPU).

The specification (number of maximum supported systems, performance of the soft processor, operating frequency of the HAoS subsystem) of the HAoS platform strongly depends on the characteristics of the FPGA device it is implemented on. Our prototype is merely an example of what a mid-range device can accomplish. It is expected that as new FPGA technologies emerge, HAoS, having been written in completely vendor-agnostic fully-synthesizable code, can be adopted with minimal effort to achieve greater performance.

B. Optimization Results

In order to quantify the performance improvements, the genetic algorithm binary knapsack problem [5] (see Fig. 8a) is reused here as a benchmark. The performance of the HAoS platform (supporting 64 systems and the MicroBlaze running at 200MHz) is measured in terms of the duration of the execution of the program until a number of interactions has been reached. The results our optimizations are given in Table II.

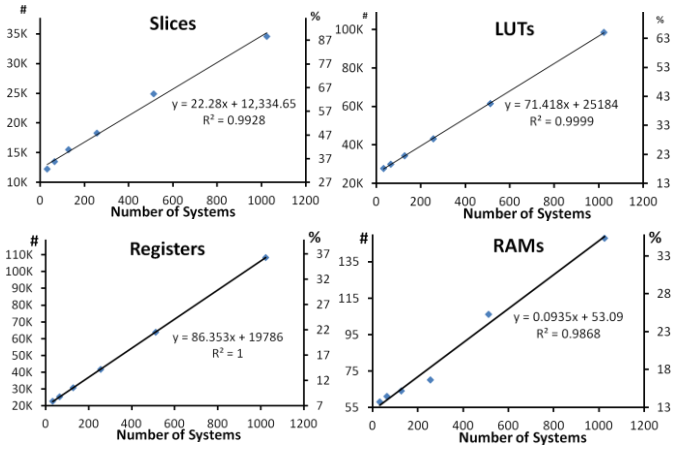

Figure 7. Linearity on area utilization increasing the number of maximum supported systems. Linear regression lines and corresponding determination coefficients given for slices, LUTs, registers and RAMs

As shown below, the most beneficial optimizations are the write-detection mechanism (solution 3.2) and the optimizations on the RSL (solution 1.1) which allowed a higher operating frequency. Furthermore, printing log information on an off-board terminal (e.g. a laptop connected to the board through USB), would heavily impact the performance of the system due to the high latency of the UART. Disabling logging would negatively impact the user experience. The solution of storing real-time information locally on the CF card enables logging with a minimal impact to performance.

TABLE II. BENCHMARK TIMING IMPROVEMENTS. RESULTS AVERAGED OVER 10 RUNS. REPORTED TIMING IS OBTAINED FOR EACH ROW USING ALL PRECEDING OPTIMIZATIONS. ON AVERAGE, THE CPU CONSUMES ~40MS FOR TRANSFORMATION FUNCTIONS AND ~15MS FOR LOW-LEVEL DRIVER.

Optimization Description	**Benchmark Timing(ms)**
No Optimization - Writing logging information (20 ASCII characters) to off-board terminal through UART	768.213
CPU Writes Back the Triplet - Writing logging information (20 ASCII characters) to on-board CF card	186.315
CPU Writes the Triplet to HAoS Registers - HAoS then writes it back to memories	176.613
Minimized CPU writes with Hardware Write-Detection	135.928
HAoS offers byte-aligned schematas in software-aware array-formatted registers for optimized CPU access	121.428
Enable hardware random numbers from the LFSR instead of using standard PRNG software functions	109.431
Optimized read/write data access functions	105.877
Updated TCAM design to have single-clock latency	101.704
Using constant-latency parallel & pipelined barrel-shifter	98.704
Resource-sharing and balanced pipelining in the RSL increased operating frequency from 12.5MHz to 50MHz	82.934

C. *Evaluation of Time Complexity for Schemata Matching*

We performed experiments for all size-variations of the hardware platform with the genetic algorithm problem (only simulated in [5]). The results are given in Fig. 8b and confirm the initial estimates in [5]. HAoS easily outperforms previous implementations (500x over [1] and 8x the over [3] for 512 systems on the GA problem). In addition to the comparison of overall performance for this problem, the complexity of the schemata matching mechanism can also be estimated. As the number of system increases, the sequential implementation struggles to find matching triplets due to its inefficient loop-based schemata matching mechanism which is of $O(n^2)$ time complexity, while the GPU version, by utilizing multiple stream processors, parallelizes part of this loop and achieves to decrease it in $O(n)$ [3]. The truly parallel nature of the TCAM is the differentiating feature for HAoS since schemata matching is executed in constant time (one clock cycle – implying $O(1)$), clearly shown in Fig. 8b. Moreover, it should be noted that these results do not only apply in the problem class of genetic algorithms, but they can be generalized to any SC model due to the importance of the schemata matching mechanism.

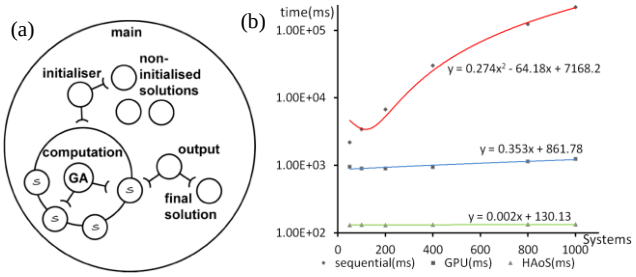

Figure 8. (a) Genetic Algorithm binary knapsack problem: Maximize the sum of the weighted values in the knapsack, keeping the sum of their weights under a threshold. Here in SC graphical notation [3]: Non-initialized solutions are initialized by the initializer context and added into the computation scope. There, they are transformed through crossover and mutation genetic operators. If a fitter solution is produced, the output context copies it to the final solution (b) Experimental results across a range of numbers of maximum systems comparing the sequential[1], GPU-based[3] and HAoS implementations

VIII. CONCLUSION

In this paper, four key issues are identified in the initial HAoS design. These issues, which involve the efficiency of random selection, schemata matching and CPU I/O access, and the programmability of the resulting architecture, are addressed here to provide a complete SC programming platform. Optimizations include a write-detection mechanism, a random selection circuit incorporating balanced pipelining and resource sharing and the latency-aware implementation of a TCAM which can execute schemata matching in constant time. A software framework completes our platform while a formal SC model development methodology is also suggested to assist prospective SC programmers. Experimental results confirm that using 512 systems HAoS can outperform previous SC implementations between 8x and 500x.

REFERENCES

[1] P. J. Bentley, "Systemic computation: A model of interacting systems with natural characteristics," *International journal of parallel, emergent and distributed systems*, vol. 22, no. 2, pp. 103–121, 2007.

[2] E. Le Martelot, "Investigating and Analysing Natural Properties Enabled by Systemic Computation within Nature-inspired Computer Models,". EngD Thesis, Department of Electrical & Electronic Engineering, UCL, London, p. 363, 2010.

[3] M. Rouhipour, P. J. Bentley, and H. Shayani, "Systemic Computation using Graphics Processors," in *Proc. of 9th International Conference on Evolvable Systems - From Biology to Hardware*, 2010.

[4] C. Sakellariou and P. Bentley, "Introducing the FPGA-Based Hardware Architecture of Systemic Computation (HAoS)," in *Mathematical and Engineering Methods in Computer Science*, vol. 7119, Z. Kotásek, J. Bouda, I. Cerná, L. Sekanina, T. Vojnar, and D. Antoš, Eds. Springer Berlin / Heidelberg, 2012, pp. 179–190.

[5] C. Sakellariou and P. Bentley, "Describing the FPGA-Based Hardware Architecture of Systemic Computation (HAoS)," *Computing And Informatics*, vol. 31, no. 3, pp. 485–505, 2012.

[6] L. N. de Castro, "Fundamentals of natural computing: an overview," *Physics of Life Reviews*, vol. 4, no. 1, pp. 1–36, 2007.

[7] L. Kari and G. Rozenberg, "The many facets of natural computing," *Communications of the ACM*, vol. 51, no. 10, pp. 72–83, 2008.

[8] J. H. Holland, *Emergence: From Chaos to Order*. Oxford University Press, 2000.

[9] R. Milner, "The Polyadic pi-Calculus: A Tutorial." Technical Report, Laboratory for Foundations of Computer Science, Computer Science Department, University of Edinburgh, 1991.

[10] C. A. Petri, "Communication with Automata." Applied Data Research Inc, Princeton NJ, 1966.

[11] A. Upegui, Y. Thoma, E. Sanchez, A. Perez-Uribe, J. M. Moreno, J. Madrenas, and G. Sassatelli, "The perplexus bio-inspired hardware platform: A flexible and modular approach," *International Journal of Knowledge-based and Intelligent Engineering Systems*, vol. 12, no. 3, pp. 201–212, 2008.

[12] N. Gershenfeld, D. Dalrymple, K. Chen, A. Knaian, F. Green, E. D. Demaine et al. "Reconfigurable asynchronous logic automata: (RALA)," *SIGPLAN Not.*, vol. 45, no. 1, pp. 1–6, Jan. 2010.

[13] M. M. Khan, D. R. Lester, L. A. Plana, A. Rast, X. Jin, E. Painkras, and S. B. Furber, "SpiNNaker: Mapping neural networks onto a massively-parallel chip multiprocessor," in *Neural Networks, 2008. IEEE International Joint Conference on*, 2008, pp. 2849–2856.

[14] B. Shackleford, G. Snider, R. J. Carter, E. Okushi, M. Yasuda, K. Seo, and H. Yasuura, "A High-Performance, Pipelined, FPGA-Based Genetic Algorithm Machine," *Genetic Programming and Evolvable Machines*, vol. 2, no. 1, pp. 33–60, 2001.

[15] D. May, H. Muller, and N. Smart, "Random Register Renaming to Foil DPA," in *Cryptographic Hardware and Embedded Systems — CHES 2001*, vol. 2162, Ç. Koç, D. Naccache, and C. Paar, Eds. Springer Berlin / Heidelberg, 2001, pp. 28–38.

[16] J. Torresen, "Reconfigurable logic applied for designing adaptive hardware systems," in *Proc. of the International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, and e-Medicine on the Internet (SSGRR 2002W)*, 2002.

[17] D. Bradley and A. Tyrrell, "A hardware immune system for benchmark state machine error detection," in *Evolutionary Computation, 2002. Proceedings of the 2002 Congress on*, 2002, vol. 1, pp. 813–818.

[18] A. Patel, C. A. Madill, M. Saldana, C. Comis, R. Pomes, and P. Chow, "A Scalable FPGA-based Multiprocessor," in *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, 2006, pp. 111–120.

[19] Xilinx, "EDK Concepts, Tools and Techniques," UG683(v13.4), 2012.

[20] N. Sulaiman, Z. A. Obaid, M. H. Marhaban, and M. N. Hamidon, "Design and Implementation of FPGA-Based Systems-A Review," *Australian Journal of Basic and Applied Sciences*, vol. 3, no. 4, pp. 3575–3596, 2009.

[21] K. Locke, "Parameterizable Content-Addressable Memory." Xilinx Application Note XAPP1151, 2011.

[22] M. Cekleov and M. Dubois, "Virtual-address caches. Part 1: problems and solutions in uniprocessors," *IEEE Micro*, vol. 17, no. 5, pp. 64–71, 1997.