Static and Dynamic Configurable Systems

Eduardo Sanchez, *Member*, *IEEE*, Moshe Sipper, *Senior Member*, *IEEE*, Jacques-Olivier Haenni, Jean-Luc Beuchat, André Stauffer, *Member*, *IEEE*, and Andrés Perez-Uribe, *Student Member*, *IEEE*

Abstract—Field-programmable gate arrays (FPGAs) are large, fast integrated circuits—that can be modified, or configured, almost at any point by the end user. Within the domain of configurable computing, we distinguish between two modes of configurability: *static*—where the configurable processor's configuration string is loaded once at the outset, after which it does not change during execution of the task at hand, and *dynamic*—where the processor's configuration may change at any moment. This paper describes four applications in the domain of configurable computing, considering both static and dynamic systems, including: SPYDER (a reconfigurable processor development system), RENCO (a reconfigurable network computer), Firefly (an evolving machine), and the BioWatch (a self-repairing watch). While static configurability mainly aims at attaining the classical computing goal of improving performance, dynamic configurability might bring about an entirely new breed of hardware devices—ones that are able to adapt within dynamic environments.

Index Terms—Configurable computing, FPGAs, static configurability, dynamic configurability.

1 Introduction

When one sets about to implement a certain computational task, then obtaining the highest performance (speed) is unarguably achieved by constructing a specialized machine, i.e., hardware. Indeed, this possibility exists, e.g., in the form of Application-Specific Integrated Circuits (ASICs); however, the price per application as well as the turnover time (from design to actual operation) are both quite prohibitive. Except for a small number of specialized niches, the computing industry has, by and large, converged onto the so-called general-purpose architecture, trading off the best possible performance in favor of a much lower cost per application and shorter delivery time. The gap between these two paradigms has been narrowing over the past few years with the coming of age of configurable computing.

Field-programmable gate arrays (FPGAs) are large, fast integrated circuits—that can be modified, or configured, almost at any point by the end user [1], [2]. A primary distinction that this novel technology brings about is that between *programmable* processors and *configurable* ones. The programmable paradigm involves a (general-purpose) processor, able to execute a limited set of operations, known as the *instruction set*. The user's (programmer's) task is that of providing a description of the algorithm to be carried out, using only operations from this limited set. This algorithm need not necessarily be written in the target language (i.e., that of the given processor), since compilation tools may be used; however, ultimately one must be in possession of an assembly-language program, which can be directly executed on the processor in question. The prime advantage of programmability is the relatively short turnover time, as well as the low cost per application, resulting from the fact that one can (potentially swiftly) reprogram the processor to carry out any other programmable task.

The configurable computing paradigm can also be regarded as one involving a processor that is able to execute but a given set of operations—however, these are at a much lower *level*. One controls the actual types of the logic devices (such as AND, OR, registers, and flip-flops), the input signals, and the output signals. The level at which the engineer can control the system's operation, i.e., the design level, is perhaps the fundamental difference between programmable processors and configurable ones.

In both a programmable and a configurable processor, the algorithm is ultimately expressed as a string of bits that is stored in memory, with the difference being the manner in which these bits are *interpreted*. A programmable processor ceaselessly iterates through a three-phase loop, where an instruction is first *fetched* from memory, after which it is *decoded*, then to be passed on to the final *execute* phase—this latter of which may require several clock cycles; this process is then repeated for the next instruction, and so on. A configurable processor, on the other hand, can be regarded as having but a single, noniterative fetch phase: The configuration string, fetched from memory, requires no further interpretation and is directly used to configure the hardware. No further phases or iterations are needed as the processor is now configured for the task at hand. The ability to control the hardware in such a direct manner using a low-level "instruction set," is a double-edged sword: The user is able to access a much wider range of functionality, with the price to be paid being that of a more arduous design task.

So as to avoid any confusion, we shall speak of a *program* when referring to a design (algorithm) within the programmable paradigm, and to a *configuration* or *configuration string* (usually a simple bit sequence) when considering the description of a configurable processor. (In analogy to the term "programmer"—and again so as to avoid any confusion—one might refer to the user of a configurable processor as a *configurer*.)

• *The authors are with the Logic Systems Laboratory, Swiss Federal Institute of Technology, IN-Ecublens, CH-1015 Lausanne, Switzerland. E-mail: Moshe.Sipper@di.epfl.ch.*

For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 108284.

Within the domain of configurable computing one can distinguish between two types of configuration strings: *static* and *dynamic*. A static configuration string, aimed at configuring the processor so as to perform a given function, is loaded (once) at the outset, after which it does not change during execution of the task at hand. Static configurability has two main objectives: 1) Improving performance (i.e., execution speed) for a given function, which essentially results in a rapid coprocessor for the task at hand (e.g., an MPEG coprocessor)—thus, one can consider this an extension of the coprocessor concept; and 2) optimizing the utilization of resources (gates and power consumption) so as to use as much of the chip surface as possible at each clock cycle. For example, one might divide the task at hand into several subtasks, each of which is implemented as a separate configuration. Task execution is achieved by successively loading the subtask configurations, thus ensuring that at each point the processor is optimized to perform the part of the computation in question. Dynamic configurability involves a configuration string that can change during execution of the task at hand, with the two main objectives being: 1) to adapt to changing (dynamic) specifications (e.g., as with an autonomous robot that is placed in a new environment) as well as to be able to handle incomplete specifications; and 2) to eliminate human design altogether. The first objective involves *partial* design, namely, the configurer designs the system to exhibit a certain general functionality, which is not necessarily the ultimate task to be accomplished—this latter is attained when the system dynamically changes its configuration string during its operation (rather than at the design phase, as with static systems). Partial design can ultimately lead to the removal of the human configurer from the design cycle, whereupon the system's configuration is carried out dynamically online. (We note in passing that, with the advancement of configurable-computing technology, one may eventually be able to configure the processor anew at each clock cycle, producing, in effect, a rapid succession of new machines.)

This paper describes four projects in the domain of configurable computing, carried out in our lab over the past five years. We shall consider both static and dynamic systems that exhibit the wide range of characteristics discussed above. We begin in Section 2 with the description of two static systems: SPYDER (a reconfigurable processor development system) and RENCO (a reconfigurable network computer). Section 3 presents two dynamic systems: Firefly (an evolving machine) and the BioWatch (a self-repairing watch). Each system is described by four articles: type, functional description, hardware description, and performance gains (and—where relevant—a software description as well). Finally, we present our concluding remarks in Section 4.

2 Static Systems

2.1 SPYDER: A Reconfigurable Processor Development System

Type (Objective). Static (Improve performance).

Functional description. The main advantage of specialized coprocessors is also one of their weaknesses: They can execute only their intended application. SPYDER (an anagram of the letters of REconfigurable Processor Development SYstem) is a reconfigurable coprocessor that self-adapts to a given application in a manner which is transparent to the user: The application is written in a high-level language (rather than an assembly program) and the compiler generates the best-adapted hardware description [3].

A processor consists of two parts: a *control unit*—a finite state machine that handles the sequencing of operations of the algorithm being executed, and a *processing unit* or *data path*—the set of memory elements and operators needed to store and process the variables of the algorithm.

The control unit has little influence on the degree of adaptation of the processor to a given algorithm. Indeed, if it is implemented as a microprogrammed machine, its structure is almost fixed: A micromemory (to store the microinstructions) linked to a sequencer (to generate the address of the next microinstruction to be executed). On the other hand, the processing unit's architecture is of vital import where the performance of the processor is concerned: The number and the size of the memory elements, the type of available operators, and their interconnection with the memory elements, determine the number of clock cycles needed to realize a certain operation.

Most reconfigurable processors enable the implementation of the two parts of the processor—indeed, they are organized as an array of FPGA circuits, possibly connected to other resources (memories, for example); the configurer (or a compiler) generates the full processor configuration for a given application [4]. Given the minor influence on performance of the control unit's architecture, SPYDER takes a simpler approach, using a fixed control unit, equivalent to a microprogrammed control unit composed of a sequencer and a very large memory. The microprogram, however, does not interpret a given assembly language, rather, it is the program to be executed.

The reconfiguration of SPYDER thus takes place in the processing unit, which consists of three FPGA circuits connected to two banks of registers. Each FPGA maintains an independent access to the registers in order to permit parallel processing of the data and, hence, the implementation of superscalar architectures. This reconfigurability presents two major limitations: the size of the FPGAs and the number of registers.

The initial objective of the project was to provide transparent hardware reconfiguration: The user would write his program in a high-level language and the compiler would generate both the code to be executed (the contents of the memory of the control unit) and the configuration of the three FPGAs. Given a certain application, the compiler would automatically determine the optimal set of operators and their possible concurrent utilization.

Given the complexity of such a compiler, we implemented an intermediate solution: The user determines the operators and describes them in a high-level language (C++). The compiler then generates the corresponding configuration of the FPGAs. Finally, the user writes the application using the predetermined set of operators. The compiler generates the corresponding code and schedules

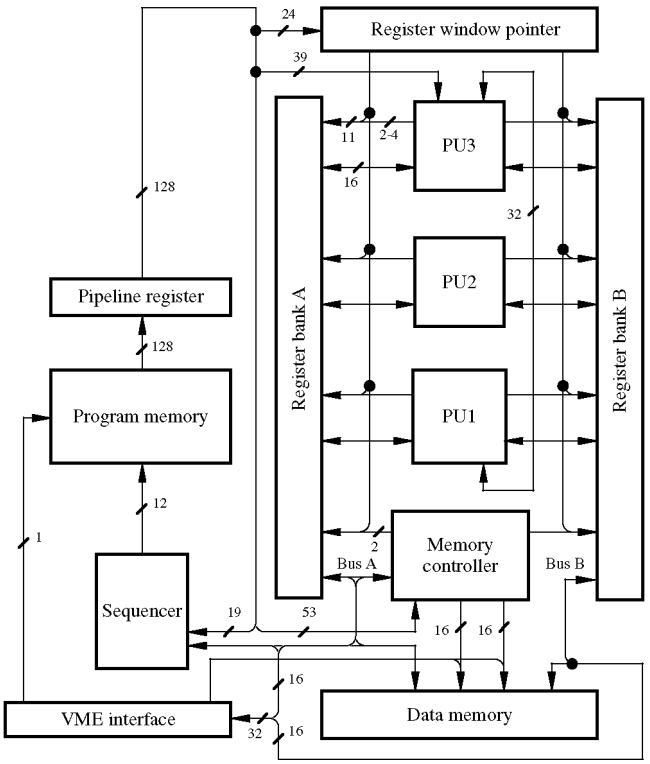

Fig. 1. The SPYDER architecture.

the operations so as to attain a maximal degree of parallelism.

As with standard coprocessors, SPYDER is connected to a host computer, which handles input/output and runs the development software.

Hardware description. SPYDER was implemented to function as a SPARCstation coprocessor, using a double-Europe board, connected to a SPARC processor by means of a VME bus (Fig. 1).

The sequencer of the control unit is implemented by means of a Xilinx XC4003 circuit. Its configuration is fixed: 16 different sequences divided into four categories (jump, call subroutine, return of subroutine, and return) are possible. The execution of each instruction takes four clock cycles, but a four-phase pipeline permits the sequencer to generate a new instruction address every clock cycle.

The program memory is separated from the data memory as in Harvard architectures: The instruction memory is 128 bits wide, while the data memory is 16 bits wide. To fully exploit the parallel-processing capabilities, the 128 bits of an instruction directly control all resources of the processor without any intermediate decoding.

The three processing units are implemented using the Xilinx XC4008 circuits. They are fully configurable and are organized in a `load/store` fashion: The data is loaded from the registers and the data memory is accessed only by means of `load` and `store` operations. At every clock cycle, each of the processing units can read two 16-bit data words and generate two 16-bit results, one per register block. They can also generate one condition bit, which is used by the sequencer, and up to 4-bit addresses to the registers. Finally, the four processing units are connected as a ring by means of two 16-bit buses, in order to facilitate pipeline operations.

The operations of the processing units are completely configurable and controlled by 21 bits of the instruction word. The distribution and function of these 21 bits are defined by the user and depend on the configuration of the units.

To facilitate a change of context, the system uses a register window mechanism, similar to that of the SPARC processors [5]. The number of registers per window is also configurable: windows of 4, 8, or 16 registers are possible.

Performance gains. SPYDER runs at only 8 MHz due to the technology used when the project began and due to economic reasons. However, the resulting performance of SPYDER on three different applications—a simulation of the Game of Life and two different image processing algorithms (skeletonization and edge detection)—surpasses several classical architectures.

A SPYDER implementation of Conway's Game of Life was compared with `xlife`, the most popular software version of this well-known cellular automaton. This application involves a grid of cells, each one of which can be in a given state at a given moment, which are updated simultaneously in discrete time steps. Our interest here was to study how fast the grid can be modified, i.e., how many cell states can be updated per second. For a 608×608 matrix of cells, SPYDER—running at 8 MHz—computes the future state of 115 million cells per second, while a microSPARC machine—running at 85 MHz—is only capable of computing the future state of 6.5 million cells per second. The results of the other two applications are delineated in [6].

The performance of SPYDER could be improved by using current-day devices. The communication with the host computer can also be improved: The VME bus was chosen due to its simple implementation and disregarding its low access speed. Nevertheless, the most important enhancements must be done in the software, using new developments in compilation techniques. We hope to one day see a compiler sufficiently powerful to accomplish our initial specifications: a system that automatically determines the optimal hardware implementation and maximal degree of parallelism.

2.2 RENCO: A Reconfigurable Network Computer

Type (Objective). Static (Improve performance).

Functional description. The ability to store an application in various (physical) locations, recently highlighted by the introduction of the *network computer*, presents many advantages: The vital resources of the computer (mass memories, applications, software libraries, etc.) are exclusively accessible through the network, thus reducing maintenance costs while adding flexibility to the system.

RENCO (REconfigurable Network COmputer) adds the power of reconfiguration to the network computer [1]: A reconfigurable surface is associated with a standard network computer in such a manner that the user can download from the network not only his or her application, but also the processor configuration able to optimally execute it.

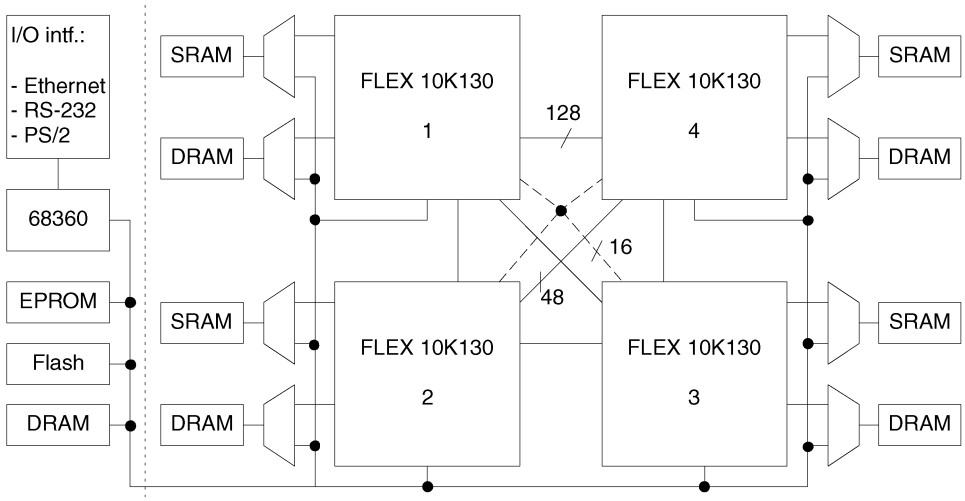

Fig. 2. RENCO block diagram.

Although, for the moment, no software manufacturer offers hardware configurations along with the software, we are quite certain of the viability of this approach. As mentioned above, there will soon be compilers able to generate a hardware description given a standard program, and one processor manufacturer (Motorola [7]) has already announced a processor with a reconfigurable on-chip surface.

Currently, RENCO is used for testing new codesign methodologies along with their associated CAD tools, and as a prototyping platform for dedicated processors.

Hardware description. RENCO is composed of two parts (Fig. 2): a conventional network computer, based on a Motorola MC68EN360 processor, and a reconfigurable area (a cluster of FPGAs connected to their own memories and to the processor bus). The user can design dedicated coprocessors for the 68360 or select them from a specialized library and dynamically download them through the network when necessary.

The network computer we have implemented is quite conventional: A microprocessor connected to three types of memory ($256\text{k} \times 16\text{b}$ of boot EPROM, $512\text{k} \times 32\text{b}$ of Flash RAM, and up to $16\text{M} \times 32\text{b}$ of DRAM). The 68360 has been chosen for its communication capabilities, for its integrated memory controller, and for the availability of many software tools.

The computer is connected to the network through an Ethernet 10Base-T interface. This communication interface is used at boot time for downloading the operating system, the applications, and the hardware configurations. An RS-232 interface is also available and is used to connect a console to the computer. Several extension connectors allow the user to expand the board features by adding specific extension boards.

The reconfigurable part contains four Altera Flex 10K FPGAs (10K130 or 10K250); these large FPGAs represent up to one million programmable logic gates. Since they are connected together, it is possible to split very large designs into up to four parts. The processor bus is connected to the four FPGAs, which can therefore be accessed as peripherals by the processor and act, e.g., as coprocessors. Each FPGA is connected to its own memories: $512\text{k} \times 8\text{b}$ of SRAM and up to $8\text{M} \times 32\text{b}$ of DRAM. The processor can also access these memories. RENCO is implemented on a 14-layer PCB.

Software description. The two parts of RENCO (network computer and reconfigurable area) each require a specific software:

- The network computer requires an operating system, with complete management of the network operations. After examining many possibilities, we chose RTEMS[1] (Real-Time Executive for Multiprocessor Systems). It is a preemptive multitasking operating system with rather modest memory requirements. It also contains the drivers for Ethernet and RS-232 and has already been adapted for the 68360 processor. Furthermore, its source code is free and a TCP/IP stack is available.
- Many software tools are necessary for the reconfigurable part: a synthesizer, a monitor allowing access to the resources and the configuration loading, a debugger, a user interface, etc. The implementation of all these tools is beyond our reach and we decided to use commercial tools when available (the synthesizer, for example) and to concentrate only on the tools specific to our system.

The basic idea is to use Java to develop some of these tools, a choice emanating from our desire to access RENCO from many different platforms through the network. The first step was to implement a Java virtual machine: We chose Kaffe[2] as the source code since it is freely available and because it has already been ported to the 68000 processor, therefore reducing our development work. In addition to the standard Java application programming interface (API), the user has at his or her disposal a board-specific API that provides classes and methods for accessing the board resources. Finally, board-specific code has been written and collected into the Custom Hardware Library (CHL), which includes utility functions for accessing the

1. http://www.oarcorp.com.
2. http://www.kaffe.org.

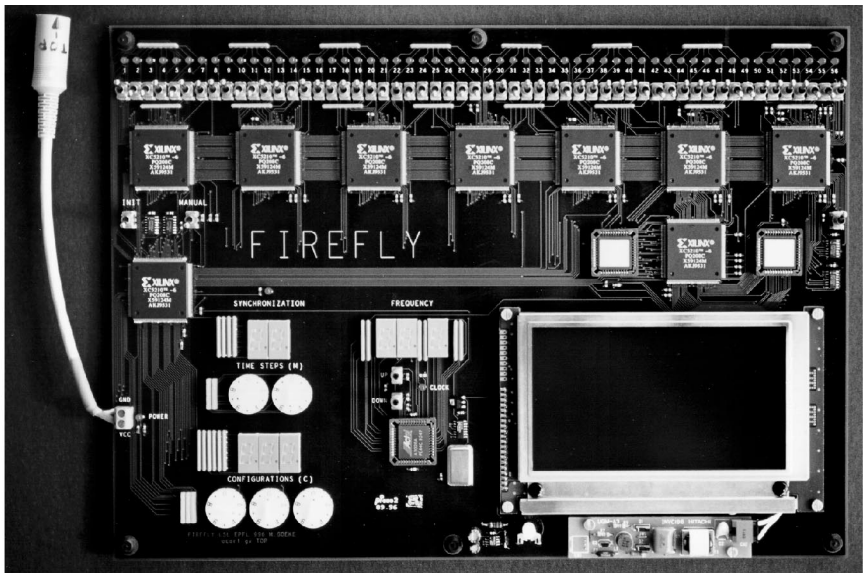

Fig. 3. The Firefly evolware board. The system is an evolving, one-dimensional, nonuniform cellular automaton. Each of the 56 cells contains a genome that represents its rule table; these genomes are randomly initialized, thereupon to be subjected to evolution. The board contains the following components: 1) LED indicators of cell states (top), 2) switches for manually setting the initial states of cells (top, below LEDs), 3) Xilinx FPGA chips (below switches), 4) display and knobs for controlling two parameters ("time steps" and "configurations") of the cellular programming algorithm (bottom left), 5) a synchronization indicator (middle left), 6) a clock pulse generator with a manually adjustable frequency from 0.1 Hz to 1 MHz (bottom middle), 7) an LCD display of evolved rule tables and fitness values obtained during evolution (bottom right), and 8) a power-supply cable (extreme left). (Note that this latter is the system's sole external connection.)

board resources. As with most reconfigurable systems, the complexity of RENCO's software is much higher than that of the hardware—it is still work in progress.

Performance gains. Our first goal is to test a novel but—in our opinion—very promising idea: Considering the hardware architecture as a downloadable resource (in addition to the software). As hardware architecture libraries are currently unavailable, we could not make full-blown evaluations to date and we have not yet proceeded further than the concept validation. Meanwhile, RENCO can also be used for complex logic design prototyping [8]. In this context, its large amount of reconfigurable logic and the large memories attached to it represent an important advantage.

3 Dynamic Systems

3.1 The Firefly Machine

Type (Objective). Dynamic (Handle changing and/or incomplete specifications).

Functional description. The idea of applying the biological principle of natural evolution to artificial systems, introduced more than four decades ago, has seen impressive growth in the past few years. Usually grouped under the term *evolutionary algorithms* or *evolutionary computation*, we find the domains of genetic algorithms, evolution strategies, evolutionary programming, and genetic programming [9]. As a generic example of artificial evolution, we consider genetic algorithms.

A genetic algorithm is an iterative procedure that involves a constant-size population of individuals, each one represented by a finite string of symbols, known as the *genome*, encoding a possible solution in a given problem space. This space, referred to as the *search space*, comprises all possible solutions to the problem at hand. The algorithm sets out with an initial population of individuals that is generated at random or heuristically. Every evolutionary step, known as a *generation*, the individuals in the current population are *decoded* and *evaluated* according to some predefined quality criterion, referred to as the *fitness*, or *fitness function*. To form a new population (the next generation), individuals are *selected* according to their fitness and then transformed via genetically inspired operators, of which the most well-known are *crossover* ("mixing" two or more genomes to form novel offspring) and *mutation* (randomly flipping bits in the genomes). Iterating this procedure, the genetic algorithm may eventually find an acceptable solution, i.e., one with high fitness.

One of the recent uses of evolutionary algorithms is in the burgeoning field of *evolvable hardware* [10], [11], which involves, among others, the use of FPGAs as a platform on which evolution takes place. The Firefly machine is one such example; our goal in constructing it was to demonstrate a system in which *all* evolutionary operations (selection, crossover, mutation, and fitness evaluation) are carried out *online*, that is, in hardware [11], [12].

Firefly is based on the cellular automata model (which we briefly encountered in Section 2.1 when describing the Game of Life application)—a discrete dynamical system that performs computations in a distributed fashion on a spatially extended grid. A cellular automaton consists of an array of cells, each of which can be in one of a finite number of possible states, updated synchronously in discrete time steps according to a *local*, *identical* interaction rule [13]. The *state* of a cell at the next time step is determined by the current states of a surrounding neighborhood of cells. This transition is usually specified in the form of a *rule table*, delineating the cell's next state for each possible

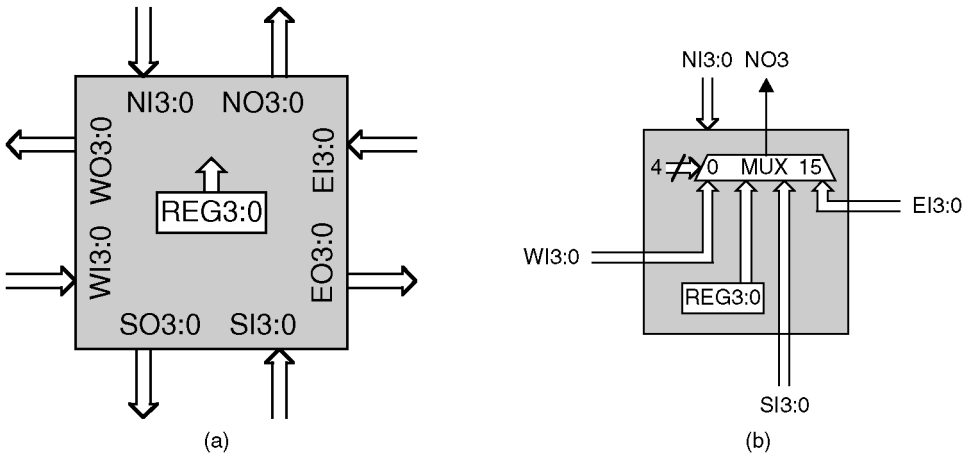

Fig. 4. MICTREE cell. (a) Diagram of connections to the four neighboring cells. (b) Programmable output connections.

neighborhood configuration. The cellular array (grid) is n-dimensional, where $n = 1, 2, 3$ is used in practice. Herein, we consider one-dimensional grids, where each cell can be in one of two states (0 or 1) and has three neighbors (itself and the cells to its immediate left and right); the rule table thus comprises 8 bits since there are eight possible neighborhood configurations. Nonuniform cellular automata have also been considered, where the local update rule need not be identical for all grid cells [13].

Based on the *cellular programming* evolutionary algorithm of Sipper [13] we implemented an evolving, one-dimensional, nonuniform cellular automaton. Each of the system's 56 binary-state cells contains a genome that represents its rule table. These genomes are initialized at random, thereupon to be subjected to evolution. The system must evolve to resolve a global synchronization task: Upon presentation of a random initial configuration of cellular states, the cellular automaton must reach, after a bounded number of time steps, a configuration whereupon the states of the cells oscillate between all 0s and all 1s on successive time steps (this may be compared to a swarm of fireflies that evolves over time to flash on and off in unison). Due to the local connectivity of the system, this global behavior—involving the entire grid—comprises a difficult task. Nonetheless, applying the evolutionary process of [13], the system evolves (i.e., the genomes change) such that the task is solved [12]. The machine is depicted in Fig. 3.

Hardware description. Firefly comprises 56 cells. The binary state of a cell is stored in a D-type flip-flop whose next state is determined either randomly, enabling the presentation of random initial configurations, or by the cell's rule table, in accordance with the current neighborhood of states. Each bit of the rule's bit string is stored in a D-type flip-flop whose inputs are channeled through a set of multiplexors according to the current operational phase of the system:

1. During the initialization phase of the evolutionary algorithm, the (eight) rule bits are loaded with random values; this is carried out once per evolutionary run.
2. During the execution phase of the cellular automaton, the rule bits remain unchanged. In this phase, several random configurations are run by the system so as to be able to calculate a fitness value.
3. During the evolutionary phase, the cell's genome (which represents its rule table) may evolve via the application of genetic operators. This is done in a completely local manner—only the genomes of the neighboring cells may be consulted.

Performance gains. The Firefly machine exhibits complete online evolution, all operations being carried out in hardware with no reference to an external computer. This demonstrates that evolving ware, *evolware*, can be constructed [12]. Such evolware systems enable enormous gains in execution speed to be had. The cellular programming algorithm, when run on a high-performance workstation, executes 60 initial configurations per second (as noted, random configurations are constantly presented to

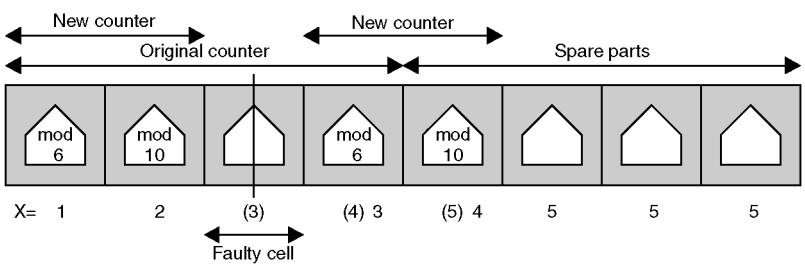

Fig. 5. Self-repair of the BioWatch. Old coordinates are shown in parentheses.

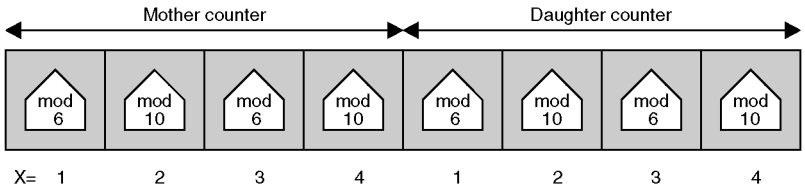

Fig. 6. Self-replication of the BioWatch.

the cellular automaton during evolution—these are used to compute the fitness value). In comparison, the Firefly machine executes $13,000$ initial configurations per second (this is achieved when the machine operates at the current maximal frequency of 1 MHz; in fact, this can easily be increased to 6 MHz, thereby attaining $78,000$ configurations per second).

While the synchronization task is not a real-world application and was selected to act as a benchmark problem for our evolware demonstration, Firefly does open up interesting avenues for future research. Evolware machines that operate in an autonomous manner can be used in the field of autonomous mobile robots, as well as for the construction, in general, of controllers for noisy, changing environments [11].

3.2 The BioWatch

Type (Objective). Dynamic (Handle changing and/or incomplete specifications).

Functional description. The BioWatch is one of the applications designed as part of the Embryonics (embryonic electronics) project, whose final objective is the development of very large scale integrated circuits, capable of self-repair and self-replication [9], [14]. These two bio-inspired properties, characteristic of the living world, are achieved by transposing certain features of cellular organization onto the two-dimensional world of integrated circuits on silicon.

The BioWatch is an artificial "organism" designed to count minutes (from 00 to 59) and seconds (from 00 to 59); it is thus a modulo-3600 counter. This organism is one-dimensional and comprises four cells aligned in a row, with identical physical connections and an identical set of resources. The leftmost cell counts tens of minutes, the one to its right counts minutes, the following one counts tens of seconds, and the rightmost cell counts seconds. Thus, in the BioWatch, each cell performs one of two specific tasks: a modulo-6 or a modulo-10 count. The organization is multicellular (as with living beings), with each cell realizing a unique function, described by a subprogram called the gene of the cell. We shall show below that a dynamic reconfiguration of the task executed by some of the cells occurs during the self-repair process of this artificial organism.

The genome is the set of all the genes of the BioWatch, where each gene is a subprogram, characterized by a set of instructions and by its horizontal coordinate X. Storing the whole genome in each cell renders the cell universal, i.e., capable of realizing any gene of the genome. This is another bio-inspired property: Each of our (human) cells also contains the entire genome, though only part of it is used (e.g., liver cells do not use the same genes as muscle cells). Depending on its position in the organism, each cell interprets the genome and extracts and executes the gene which configures it. The BioWatch thus performs what is known in biology as cellular differentiation.

Hardware description. The BioWatch is a four-cell, one-dimensional application of the two-dimensional cellular automaton defined in the Embryonics project [9], [14]. Each cell of the automaton is a binary decision machine whose microprogram represents the genome and each part of the microprogram is a gene whose execution depends on the physical position of the cell in the array, i.e., on its coordinates. Ultimately, we plan to implement the automaton using a novel kind of coarse-grained, field-programmable gate array, where each basic cell, called MICTREE (for tree of micro-instructions) has four neighbors (to the south, west, north, and east). The MICTREE cell holds a 4-bit state register $REG3:0$ (Fig. 4a). Four 4-bit buses enter the cell from its neighbors ($SI3:0$ from the south, $WI3:0$ from the west, $NI3:0$ from the north, and $EI3:0$ from the east) and, correspondingly, four output buses go out in the four cardinal directions ($SO3:0$ to the south, $WO3:0$ to the west, $NO3:0$ to the north, and $EO3:0$ to the east).

Each MICTREE cell thus has 16 outputs $SO3 \ldots EO0$. Each of these outputs can be programmed to take on a value from four possible sources (Fig. 4b). For example, output $NO3$ can take on one of 16 values from the following sources: the four bits $REG3:0$ of register REG, the four bits $SI3:0$ of the south input bus SI, the four bits $WI3:0$ of the west input bus WI, and the four bits $EI3:0$ of the east input bus EI.

The binary decision machine of the MICTREE cell executes microprograms written using a set of six instructions:

1. **if** VAR **else** $LABEL$,
2. **goto** $LABEL$,
3. **do REG** $= DATA$
4. **do X** $= DATA$,
5. **do Y** $= DATA$, and
6. **do** $VAROUT = VARIN$.

The first three instructions are used to compute the modulo-6 and modulo-10 counts of the BioWatch application. The next two are used when computing the $X3:0$ and $Y3:0$ coordinates of the cell. The last instruction is used to program the input/output connections.

While our long-term objective is the design of very large scale integrated circuits, each MICTREE cell is currently implemented in an Actel 1020 FPGA circuit and embedded within a small plastic box intended as a demonstration module.

Performance gains. Self-repair of an artificial organism allows partial reconstruction of the original device in case of a minor fault. In order to implement a self-repair process in the BioWatch, as many spare cells are required to the right of the array as there are faulty cells to repair (four spare cells in the example of Fig. 5). This process is achieved by bypassing the faulty cell and shifting to the right all or part of the original cellular array. The new coordinates, thus defined, lead to the dynamic reconfiguration of the task performed by the cell (modulo-6 or modulo-10 count).

Self-replication of an artificial organism allows for the complete reconstruction of the original device in case of a major fault. In the BioWatch, the self-replication process rests on two assumptions: 1) There exists a sufficient number of spare cells to the right of the array (four in our example), and 2) the calculation of the coordinates produces a cycle ($X = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$ in Fig. 6). As the same pattern of coordinates produces the same pattern of genes, self-replication can be easily accomplished if the microprogram of the genome, associated with the homogeneous network of cells, produces several instances of the basic pattern of coordinates.

With a larger number of cells, it becomes possible to add the extensions needed for a practical use of the BioWatch: Preserving the current time while self-repair is being effected and setting and resetting the time. It is also quite easy to introduce additional functions other than the counting of seconds, minutes, and hours; for example, computing the date, keeping track of the day of the week, or handling leap years.

4 Concluding Remarks

Our aim herein has been to demonstrate a number of FPGA applications that cover a wide range of characteristics. First and foremost, we made a distinction—which we believe to be of prime importance—between static and dynamic configuration strings. The former, aimed at configuring the processor so as to perform a given function, is loaded once at the outset, after which it does not change during execution of the task at hand. A dynamic configuration string, on the other hand, can continually change.

Static FPGA applications, such as SPYDER and RENCO, are mainly aimed at attaining the classical goal in computing: that of improving performance—be it in terms of speed, resource utilization, or area usage. With configurable processors slowly but surely inching their way toward the mainstream of the computing industry, we will probably be seeing more such static applications in the near future. Thus, the future may see a merging of the classical-processor industry with the configurable-computing industry.

Dynamic devices, such as Firefly and BioWatch, represent a less conventional approach that may, in fact, be quite revolutionary (though perhaps not in the immediate future). With the rise of bio-inspired computing, we expect to see more hardware devices imbued with properties usually associated up until now only with living beings: learning, evolution, self-repair, self-replication, and so forth. In general, this will result in systems that are more *adaptive*—able to undergo modifications according to changing circumstances, thus continuing to function within their dynamic environments. The applications of such systems are bounded only by our imagination.

Acknowledgments

We are grateful to Daniel Mange for helpful discussions. This work was supported in part by Grant 2000-049349.96 from the Swiss National Science Foundation and by a grant from the Werner Steiger Foundation..

References

[1] J. Villasenor and W.H. Mangione-Smith, "Configurable Computing," *Scientific Am.,* vol. 276, no. 6, pp. 54-59, June 1997.
[2] *Field-Programmable Gate Array Technology,* S.M. Trimberger, ed. Boston: Kluwer Academic, 1994.
[3] C. Iseli and E. Sanchez, "Spyder: A SURE (SUperscalar and REconfigurable) Processor," *J. Supercomputing,* vol. 9, no. 3, pp. 231-252, 1995.
[4] A. DeHon, "Architectures for General-Purpose Computing," A.I. Technical Report No. 1586, Artificial Intelligence Laboratory, MIT, Oct. 1996.
[5] *The SPARC Architecture Manual,* D.L. Weaver and T. Germond, eds. Englewood Cliffs, N.J.: Prentice Hall, 1994.
[6] C. Iseli, "Spyder: A Reconfigurable Processor Development System," PhD thesis, Computer Science Dept., Swiss Federal Inst. of Technology, Lausanne, thesis no. 1476, 1996.
[7] "Motorola Core+ Chip Merges CPU with FPGA," *Microprocessor Report,* vol. 12, no. 2, p. 10, 1998.
[8] Z. Salcic and A. Smailagic, *Digital System Design and Prototyping Using Field Programmable Logic.* Boston: Kluwer Academic, 1997.
[9] *Bio-Inspired Computing Machines: Toward Novel Computational Architectures,* D. Mange and M. Tomassini, eds. Lausanne, Switzerland: Presses Polytechniques et Universitaires Romandes, 1998.
[10] *Towards Evolvable Hardware,* E. Sanchez and M. Tomassini, eds. Heidelberg: Springer-Verlag, 1996.
[11] M. Sipper, E. Sanchez, D. Mange, M. Tomassini, A. Pérez-Uribe, and A. Stauffer, "A Phylogenetic, Ontogenetic, and Epigenetic view of Bio-Inspired Hardware Systems," *IEEE Trans. Evolutionary Computation,* vol. 1, no. 1, pp. 83-97, Apr. 1997.
[12] M. Sipper, M. Goeke, D. Mange, A. Stauffer, E. Sanchez, and M. Tomassini, "The Firefly Machine: Online Evolware," *Proc. 1997 IEEE Int'l Conf. Evolutionary Computation (ICEC'97),* pp. 181-186. 1997.
[13] M. Sipper, *Evolution of Parallel Cellular Machines: The Cellular Programming Approach.* Heidelberg: Springer-Verlag, 1997.
[14] D. Mange, E. Sanchez, A. Stauffer, G. Tempesti, P. Marchal, and C. Piguet, "Embryonics: A New Methodology for Designing Field-Programmable Gate Arrays with Self-Repair and Self-Replicating Properties," *IEEE Trans. VLSI Systems,* vol. 6, no. 3, pp. 387-399, Sept. 1998.

Eduardo Sanchez received a diploma in electrical engineering from the Universidad del Valle, Cali, Colombia, in 1975, and a PhD from the Swiss Federal Institute of Technology in 1985. Since 1977, he has been with the Department of Computer Science at the Swiss Federal Institute of Technology, Lausanne, where he is currently a professor in the Logic Systems Laboratory, engaged in teaching and research. His chief interests include computer architecture, VLIW processors, reconfigurable logic, and evolvable hardware. Dr. Sanchez was co-organizer of the inaugural workshop in the field of bio-inspired hardware systems, the proceedings of which are titled *Towards Evolvable Hardware* (Springer-Verlag, 1996). He is a member of the IEEE.

Moshe Sipper received a BA in computer science from the Technion-Israel Institute of Technology, and an MSc and a PhD from Tel Aviv University. He is a senior researcher in the Logic Systems Laboratory at the Swiss Federal Institute of Technology, Lausanne. His chief interests involve the application of biological principles to artificial systems, including evolutionary computation, cellular computing, bio-inspired systems, evolving hardware, complex adaptive systems, artificial life, and neural networks. Dr. Sipper has published close to 70 papers in these areas, as well as the book *Evolution of Parallel Cellular Machines: The Cellular Programming Approach* (Springer-Verlag, 1997). He was program chairman of the Second International Conference on Evolvable Systems: From Biology to Hardware (ICES 98) held in Lausanne, Switzerland, in September 1998. He is a senior member of the IEEE.

Jacques-Olivier Haenni received a diploma in computer engineering from the Swiss Federal Institute of Technology, Lausanne, in 1997. Since then, he has been a PhD candidate at the Logic Systems Laboratory of the Swiss Federal Institute of Technology. His research interests include computer architecture, reconfigurable computing, and co-design.

Jean-Luc Beuchat received a diploma in computer engineering from the Swiss Federal Institute of Technology, Lausanne, in 1997. Since then, he has been a PhD candidate at the Logic Systems Laboratory of the Swiss Federal Institute of Technology, working on the digital implementation of reconfigurable neuroprocessors. His research interests include neural networks, field-programmable devices, reconfigurable systems, and on-line arithmetic.

André Stauffer recieved a diploma in electrical engineering and a PhD degree from the Swiss Federal Institute of Technology, Lausanne. He is a senior lecturer in the Department of Computer Science at the Swiss Federal Institute of Technology. He spent one year as a visiting scientist at the IBM T.J. Watson Research Center, Yorktown Heights, New York. In addition to digital design, his research interests include circuit reconfiguration and bio-inspired systems. Dr. Stauffer also collaborates with the Centre Suisse d'Electronique et de Microtechnique SA in Neuchâtel, Switzerland. He was co-organizer of a special session entitled "Toward Evolware" held as part of the IEEE International Conference on Evolutionary Computation (ICEC '97). He is a member of the IEEE.

Andrés Pérez-Uribe received a diploma from the Universidad del Valle, Cali, Colombia, in 1993. From 1994 to 1996, he held a Swiss government fellowship and is currently a PhD candidate in the Department of Computer Science, Swiss Federal Institute of Technology, Lausanne. Since 1994, he has been with the Logic Systems Laboratory at the Swiss Federal Institute of Technology, working on the digital implementation of neural networks with adaptable topologies, in collaboration with the Centre Suisse d'Electronique et de Microtechnique SA (CSEM). His research interests include artificial neural networks, field-programmable devices, evolutionary techniques, and complex and bio-inspired systems. He was a member of the steering committee and secretary of the Second International Conference on Evolvable Systems: From Biology to Hardware (ICES 98) held in Lausanne, Switzerland, in September 1998. He is a student member of the IEEE.