Hardware/software coevolution of genome programs and cellular processors

Gianluca Tempesti, Pierre-André Mudry, Guillaume Zufferey
Cellular Architectures Research Group
Ecole Polytechnique Fédérale de Lausanne (EPFL)
EPFL-IC-GRTEM, Station 14, 1015 Lausanne, Switzerland
gianluca.tempesti@epfl.ch

Abstract

The application of evolutionary techniques to the design of custom processing elements bears a strong relation to the natural process that led to the co-evolution of cells and genomes in biological organisms. As such, it is an interesting avenue for an effective application of evolutionary approaches in the domain of hardware design.

The architecture of conventional non-configurable processors, however, is ill-adapted to this kind of approach, as evolution can operate exclusively on the software (the genome) and not on the hardware that executes it, leading to scalability issues that seem very difficult to overcome.

Building on a family of configurable processors we developed in the past years, in this article we introduce a design methodology that allows the architecture of the processor to co-evolve together with the code to be executed.

1. Introduction

The analogy between biological cells and processors (or processing elements) in silicon is fairly current in the design of bio-inspired computing systems, particularly through the implicit assumption that programs are equivalent to genomes (e.g, in the genetic programming paradigm). With few exceptions, however, these efforts have focused either on off-the-shelf processors or hand-designed custom ones.

The former provide researchers with design environments and software tools, but cannot use dedicated bio-inspired hardware and need to move much of the inherent complexity to software, introducing scalability issues that have not been solved to date. On the other hand, custom processors can exploit specific mechanisms but need to be designed by hand, a process that absorbs time and resources better spent on the exploration of bio-inspired solutions.

Due to these issues, bio-inspired processors are rarely used as a tool, hindering in turn research aimed at exploiting in hardware bio-inspired processes such as cell specialization (i.e., the capability of cells to adapt their structure to their task) or cellular evolution (i.e., the co-evolution of the cell structure and of the genome of the organism).

The work presented here describes a step in the development of a set of processor architectures dedicated to bio-inspired systems and of a design environment that simplifies their conception and use. We will show how the architecture of the processors and the code they execute (the genome) can be co-evolved, a process that considerably simplifies the software and increases its evolvability and scalability.

2. Background

Bio-inspiration imposes a set of non-conventional constraints on the design of processors meant to behave as cells within an artificial organism. In this section, we will present how we draw inspiration from biology for system design and then analyze how the processor architecture we have been using can address the requirements of the approach.

2.1. Embryonics

In the Embryonics project [13], we have been studying how to transpose some of the mechanisms and properties involved in the development of complex organisms to the design of digital hardware. Our approach involves a self-contained mapping between the world of multi-cellular organisms in biology and that of silicon, based on levels of complexity ranging from the population to the molecule.

Within this mapping, we define an artificial organism as a parallel array of cells, where each cell is a simple processor that stores the description of the operation of every cell in the organism as a program (the *genome*). This inherent redundancy is compensated by the added capabilities of the system, such as growth [14] and self-repair [18].

The operation of multi-cellular organisms relies, among other things, on the specialization of the cells to a finite set of specific operations, implying that their *physical structure* is adapted to its function (e.g., a skin cell is physically

Proceedings of the First NASA/ESA Conference on Adaptive Hardware and Systems (AHS'06)
0-7695-2614-4/06 $20.00 © 2006 IEEE

different from a liver cell). Structural differences notwithstanding, the same program (genome) controls the operation of all cells. To maintain the analogy with digital processors, we must achieve a similar degree of adaptation.

A first step in this direction was to define our cells as *reconfigurable* processing elements, realized by programmable logic and structurally adapted to the task. For a given application, all cells are structurally identical and contain the same program (and can thus be seen as *stem cells* [16]), but different parts of the program and of the structure are activated depending on the cell's position in the organism, implementing specialization.

2.2. MOVE processors

Our cellular processors require then an architecture that is substantially different from conventional general-purpose processor architectures: it must be possible to adapt the cell structure to the application to exploit the programmability of application specific systems and it must be possible to adapt the topology of the system to the application to take advantage of the features of the ontogenetic approach [19].

To achieve this kind of adaptability within an array of processors, we exploited the *Move* or *TTA* (Transport-Triggered Architecture) paradigm [1][3], originally developed for the design of application-specific dataflow processors (i.e., processors where the instructions define the flow of data, rather than the operation to be executed).

In some respects, the overall structure of a TTA-based system is fairly conventional: data and instructions are fetched from the main memory using standard mechanisms (caches, memory management units, etc.) and are decoded more or less as in conventional processors. The basic differences lay in the architecture of the processor itself, and hence in the instruction set.

Rather than being structured, as is usual, around a more or less serial pipeline, a *Move* processor (Fig. 1) relies on a set of *functional units* (FUs) connected by one or more *transport busses*. All computation is carried out by the functional units (examples of FUs can be adders, multipliers, register files, etc.) and the instructions simply move data to and from the FUs in the order required to implement the desired operations. As all FUs are uniformly accessed through I/O registers, instruction decoding is reduced to its simplest expression, as only one instruction is needed: `move`.

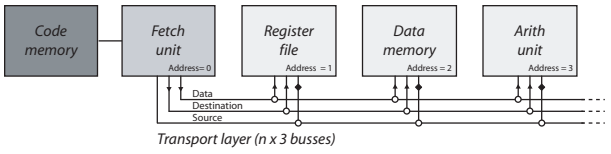

Figure 1. Architecture of a TTA processor.

TTA move instructions trigger operations that in fact correspond to normal RISC instructions. For example, a RISC `add` instruction specifies two operands and, usually, a destination register. The *Move* paradigm requires a slightly different approach to obtain the same result: instead of using a specific `add` instruction, the program moves the two operands to the input registers of a functional unit that implements the add operation. The result can then be retrieved from the output register of the FU and used where needed.

The *Move* approach, in and for itself, does not imply high performance, but several arguments in favor of TTAs have been proposed [3][10]. Most notable of these in our context is the capability to easily add new *instructions* in the form of functional units. This feature, along with the fact that the functional units are handled as *"black boxes"*, i.e. without any inherent knowledge of their functionality, implies that the architecture of the processor can be described as a *memory map* which associates the different possible operations with the addresses of the corresponding FUs.

This capability, coupled with an algorithm such as the one described in this paper, introduces in the system an interesting amount of flexibility by specializing the instruction set (i.e., with *ad-hoc* functional units) to the application while keeping the overall structure of the processor (fetch and decode unit, bus structure, etc.) unchanged.

In biological terms, this ability corresponds to specializing the internal structure and operation of the cells (organelles, metabolic pathways, protein coding, etc.) while keeping the structure and operation of the genome (DNA bases, genome chemistry and access, etc.) unchanged, and as such seems perfectly suited to the requirements of an approach that seeks to draw inspiration from biology. It also opens the way, as shown in this article, to an efficient co-evolution of the genome and of the cell architecture.

3. Objectives and related literature

The work presented in this paper is part of a broader research effort aimed at developing a complete design environment for bio-inspired digital systems. The goal of the project is to allow researchers to rapidly design a cellular system starting from a high-level language description of an application and to implement it within a reconfigurable co-processing unit for highly-parallel computation.

The environment will eventually handle the design of complete multi-cellular networks and include features to control communication and growth within the network. This article, however, focuses on the synthesis and optimization of a single processor within the network, i.e., of a cell and of the *gene* that it will have to express within the organism. Developed specifically to address the design of application-specific processors, the *Move* architecture is ideally suited for this kind of *hardware/software codesign*.

Consisting of the design of the hardware and the software layers of a system at the same time, codesign has appeared the early 90s and is widely spread in the industry. This approach exploits the synergies of hardware and software in an application-specific system. Such systems are usually built around a core processor that can be connected to hardware modules tailored for a specific application. This "tailoring" corresponds to the codesign of the system and can be divided into different subtasks, as defined in [8]: partitioning, co-synthesis, co-verification and co-simulation.

In this paper, we focus on the complex, NP-complete [15] partitioning problem, defined as follows: starting from a program to be implemented on a system and given some execution time and/or size constraints, partitioning consists in determining which parts of the program have to be implemented in hardware in order to satisfy the constraints.

Several methods have been proposed in the past to solve this problem: Gupta and De Micheli start with a full hardware implementation [7], while Ernst *et al.* [6] use profiling results to determine with a simulated annealing algorithm which blocks to move to hardware. Vahid *et al.* [21] use clustering together with a binary-constrained search to minimize hardware size while meeting constraints. Other approaches include fuzzy logic [2], GAs [4][17], hierarchical clustering [11] or tabu search [5].

In this article, we will show that despite the fact that standard GAs have been shown in the past to be less efficient than other techniques to solve the partitioning task [22][23], they can be hybridized to take into account domain-specific knowledge and solve it much more efficiently.

Because of the versatility of *Move* processors, automatic partitioning becomes very interesting for the synthesis of bio-inspired, application-specific processorsm as it can be used to determine which parts of a given program are the best candidates to be implemented as FUs in the processor.

From a biological and evolutionary perspective, this approach has several advantages: it models rather accurately the kind of cellular specialization that occurs during the development of an individual (as required for embryonic systems), it can be integrated into a design flow (a crucial advantage for research), and, more importantly for this article, it increases the evolvability of the system by providing a well-defined target for genetic algorithms (the evolution of the cell as the basic building block of complex organisms).

4. A genetic algorithm for partitioning

To tackle the partitioning problem in our processors we opted for a genetic algorithm that selects the parts of the program code which can be advantageously transformed into functional units for our processors. Some domain-specific ameliorations have been introduced to guide the algorithm in its search and increase its performance.

Obviously, this approach is not a requirement for using *Move* processors and indeed their features fit the needs of bio-inspired systems independently of any evolutionary technique. However, genetic algorithms in this context not only maintain a strong analogy to nature (where cells did indeed evolve along with the genome) but are also a very efficient method to tackle a difficult problem (partitioning).

4.1. Overview and genome encoding

The partitioner we used (Fig. 2) works on programs written in a simplified programming language that supports all the classical declarative language constructs in a C-like syntax. Several limitations (which could eventually be lifted) have however been imposed to this language: pointers are not supported, recursion is forbidden, and no typing exists (all values are treated as 32-bit integers).

Prior to being used in the algorithm, the code is *annotated* with coverage information using standard profiling tools on a *Java*-equivalent program. This step provides an estimation of how many times each line is executed for a large number of realistic input vectors, allowing the GA to find the most interesting kernels to be moved to hardware.

The algorithm then analyzes the syntax of the annotated source code and generates the corresponding *program tree*, which will then be the main data structure in the algorithm. From this tree, the genome is constructed by associating to each node of the tree a boolean value indicating if the node's subtree is implemented in hardware (Fig. 3).

As we also want to regroup instructions together to form new FUs, to each statement (assignments, *for*, *while*, *if*, function calls...) correspond two additional boolean values that permit the creation of groups of adjacent instructions: the first value indicates if a new group has to be created and, if so, the second value indicates if the whole group has to be implemented in hardware (i.e. to create a new FU).

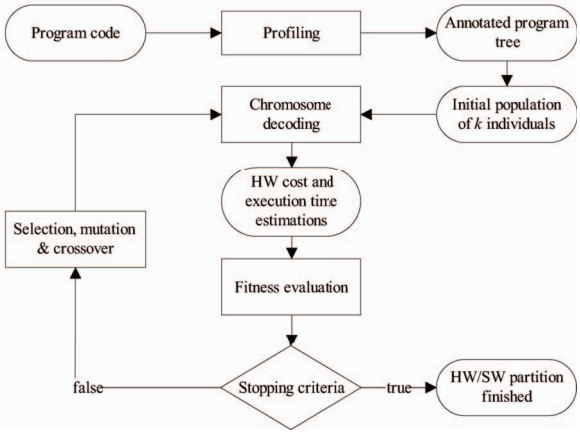

Figure 2. Flow diagram of the algorithm.

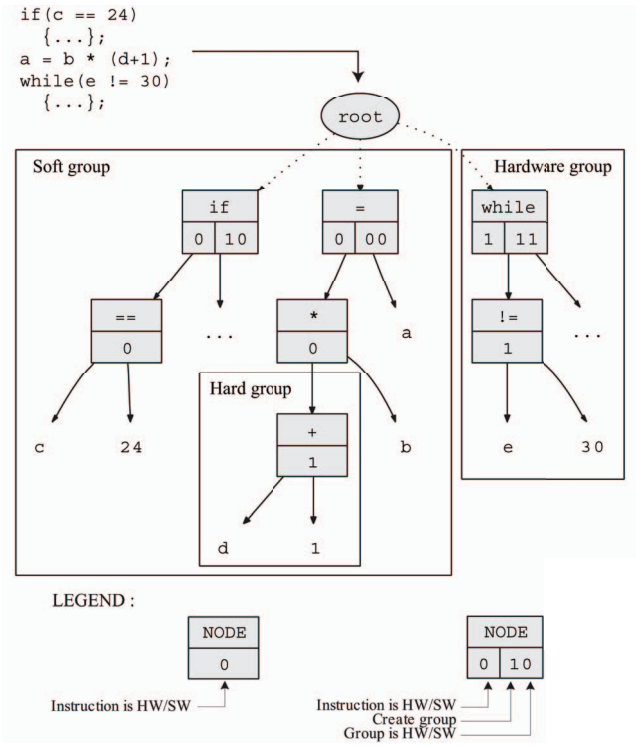

Figure 3. Genome encoding.

The complete genome of the program could then be formed by the concatenation of the values of the nodes. This naive encoding, however, introduces a strong bias by implicitly favoring the implementation in hardware of nodes close to the root. In fact, when the GA changes a node to hardware, its whole sub-tree is also changed and the genes of the sub-nodes are no longer affected by the evolutionary process. If this occurs for an individual that has a good fitness, the evolution may stay trapped in a local maximum, because it will never explore the possibility of using smaller functional units within that hardware sub-tree.

The solution we propose (Fig. 4) resides in the decomposition of the program tree into different levels that correspond to *blocks* in the program (series of instructions delimited by brackets). These levels represent interesting points of separation because they often correspond to the most computationally intensive parts of the programs (e.g. loops) that are good candidates for being implemented in new FUs.

The GA is then recursively applied to each level, starting with the deepest ones (level 0). To pass information between each level, the genome of the best individual evolved at each level is stored. A mutated version of this genome is then used for each new individual created at the next level.

This approach permits to construct the solution progressively by trying to find the optimal solution of each level. It gives priority to nodes close to the leaves to express themselves, and thus good solutions will not be hidden by higher level groups. This specific optimization also dramatically reduces the search space of the algorithm as it only has to work on small trees representing different levels of complexity in the program.

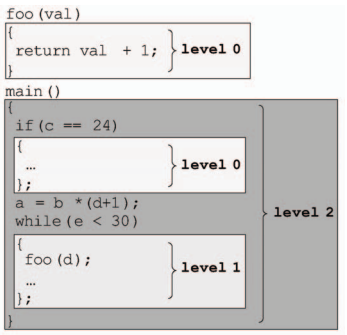

Figure 4. Levels definition.

4.2. Genetic operators and optimizations

The GA starts with a basic population composed of random individuals. For each new generation, individuals are chosen for reproduction using rank-based selection with elitism. To ensure a larger population diversity, part of the new population is not obtained by reproduction but by random generation. For each generation, the standard genetic operators are applied, together with some domain-specific operations that increase the performance of the algorithm.

Crossover is applied by randomly choosing a node in each parent's tree and by exchanging the corresponding sub-trees. This corresponds to a double-point crossover and it is used to enhance the genetic diversity of the population.

A mutation consists of inverting the binary value of a gene. However, as a mutation can affect the partitioning differently, depending on where it happens, different mutation rates are defined for the following cases:

1. A new functional unit is created.
2. An existing functional unit is destroyed and the corresponding group reverts to software.
3. A new group of statements is created or two groups are merged together.

Using different mutation rates for the creation and the destruction of functional units can be very useful. For example, increasing the probability of destruction introduces a bias towards fewer FUs.

In addition to these standard operators, two additional operations are performed on each new individual: *pattern matching* and *non-optimal block pruning*.

To optimize partitioning, it is crucial to find *reusable* functional units that can be used at different locations in a program. This task is however very difficult for the standard operators, and to help evolution to find such blocks, a *pattern matching* step has been added: every time a piece of code is transformed in hardware, similar pieces are searched in the whole program tree and mutated to become hardware as well. Reusability is then greatly improved because the standard operators need to find only one occurrence of a block, the others being given by this new step.

The GA is further aided by *pruning* the best individual of each generation. This step consists of removing all non-optimal hardware blocks from the genome. These blocks are detected by computing the fitness of the individual when each block or group of similar blocks is implemented in software. If the considered block does not increase or decreases the fitness, the genome is changed so that the part in question is no longer implemented as a functional unit.

4.3. Fitness computation

The fitness of an individual is derived by computing hardware size and execution time. Different techniques exist to determine these values (e.g., [9][20]). The method we adopted is based on a very fine characterization of elementary hardware building blocks (blocks that conduct very simple logical and arithmetic operations such as `AND`, `OR`, $+$, ...) on the target platform. These building blocks can then be joined to create more complex operations that form new FUs and their characterization is used to determine the size and timing values for the FUs.

While our approach can be applied to any programmable hardware platform, this characterization obviously depends on the target platform. In the current implementation we use a Xilinx Virtex™FPGA. The size (number of slices) and timing metrics of the basic blocks have been determined using the Synplify Pro™synthesis solution coupled, in some cases, with the Xilinx place-and-route tools.

This very detailed characterization permitted us to take into account a wide range of timings, from sub-cycle estimates for combinational operators to multi-cycle, high latency operators such as pipelined dividers. Area estimators were built using the same principles. Using these parameters, determining size and time for each sub-tree is then relatively straightforward because only two different cases have to be considered:

1. For software sub-trees, the estimation is done recursively over the nodes of the tree, adding at each step the appropriate execution time and potential hardware unit (e.g. the first time an `add` instruction is encountered, an `add` FU must be added to compose the minimal processor necessary to execute this program).
2. For hardware sub-trees, the computation is a bit more complex because it depends on the position of the considered sub-tree. For example, if it represents a new FU, some computation is needed to take into account factors such as the time to move the data to the new FU, the size of the bus interface, and the size of the registers required for the storage of the local variables.

The objective of the GA is to find the partitioning with the smallest execution time given an area constraint. Assuming that the basic solution to the problem is a software implementation on a simple processor with minimal hardware, we use a relative fitness function. This simple processor, whose hardware size is β, has then a fitness of 1 and the fitness of the discovered solutions are expressed in terms of this trivial solution. We also define α as the time to execute the given program on this trivial processor.

Moreover, as our goal is to use all available hardware, we biased evolution towards solutions that use more hardware. This bias must be active only when a relatively good solution has been found, as we do not want evolution to be biased towards solutions with a large hardware cost at the start. To achieve this, a dynamic parameter is added to the fitness function to let larger blocks be used when good solutions are found. For an individual of hardware size s, we first compute the adaptive factor k:

$$k = \frac{hwLimit - s}{hwLimit}$$

where $hwLimit$ is the maximum hardware size for the processor with the new FUs defined by the algorithm.

For an individual having a size s and an execution time t, the fitness function f can then be defined as:

$$f(s,t) = \begin{cases} \frac{\alpha}{t} \cdot (k \cdot \frac{\beta}{s} - k + 1) & \text{If } s \leq hwLimit \\ (\log(s - hwLimit) + 1)^{-1} & \text{otherwise} \end{cases}$$

This function assigns a greater fitness to individuals that balance well the area/speed compromise: when the speed increase obtained during one step of the evolution is relatively bigger than the hardware increase it requires, the fitness increases. The parameter k implies that the weight of the hardware increase in the computation of the fitness drops as the individual approaches $hwLimit$.

5. Results

In the context of the design of bio-inspired systems, it is fundamental to develop approaches that, on one hand, can be easily exploited and parametrized to allow their use as research tools and, on the other hand, are efficient and can scale up to real-world applications. In this section, we present the system interface, illustrate its operation on a simple example, and show some performance measures.

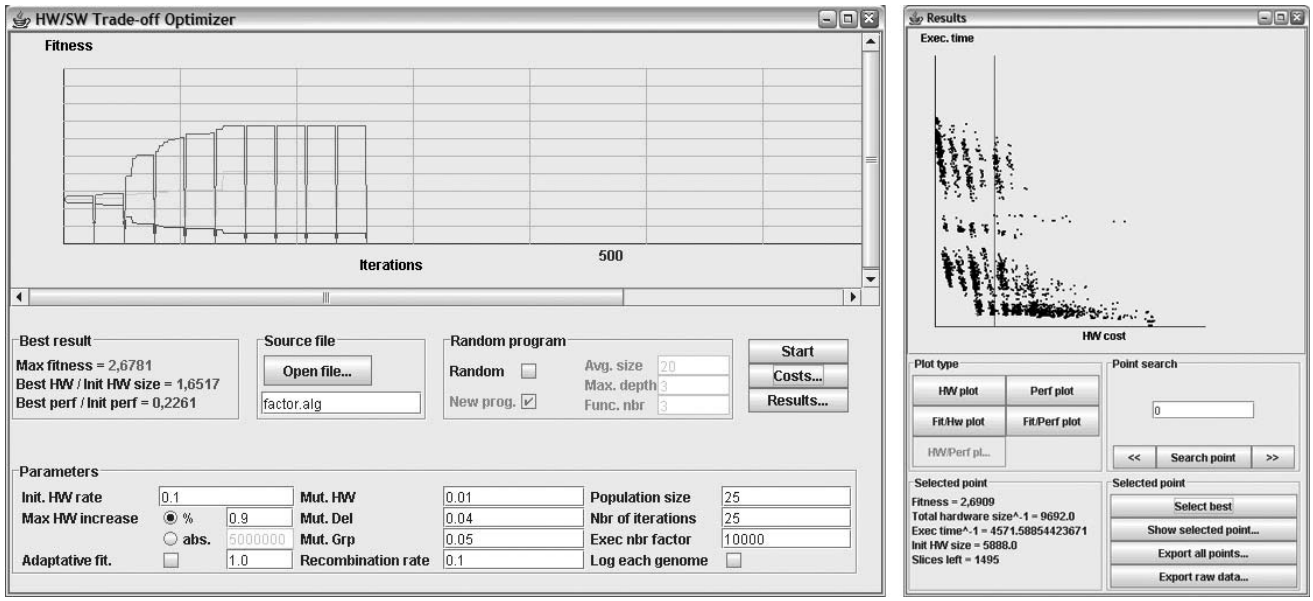

Figure 5. The two main windows of the user interface: execution (left) and results (right).

5.1. User interface

The above GA partitions quite efficiently programs of useful size, as we shall see. However, the algorithm itself could obviously be ameliorated by exploiting the latest evolutionary techniques and especially by identifying the correct parameters for the system. To this end, we designed a graphical user interface (written in *Java*, like all other software in the system) based on two main windows (Fig. 5).

The execution window allows the user to set the conventional and the domain-specific parameters of the algorithm. Beside the ”standard” mutation and recombination rates, population size, and number of iterations, the key parameters (bottom left) are the hardware rate of the initial population, the maximum hardware increase with respect to the minimal processor (defined either as a percentage or as an absolute number), and whether an adaptive fitness is used (and, if not, the value of k). In addition, a sub-window (not shown) allows the user to change the hardware costs of the basic blocks used to evaluate the size of the processor.

The middle row of commands in the window determines the program to which the algorithm will be applied (loaded from a source file or generated randomly to allow faster benchmarking) and displays the current results of the evolution as numbers that correspond to the graphical display at the top of the window. These results show the fitness achieved by the best individual in the current population and the hardware and performance increases for this individual.

Once the evolutionary run is completed, the results window can display various representations of the output of the system. Firstly, the performance of the whole evolutionary search can be explored visually using various two-dimensional graphs of the optimization space from different points of view, a feature that allows the user to estimate the impact of the different parameters of the evolutionary process. Another option allows the user to export the data generated for external analysis, plotting, or reuse.

Finally, every single evolved individual can also be analyzed and displayed. The analysis shows the basic parameters of a given solution. The display shows the final code after optimization, showing the discovered FUs in their utilization context. Using this view, it is for example possible to see where the blocks have been used in the code.

5.2. Operation

Fig. 6 shows the effects of the algorithm on a set of functions of the `FACT` program, which factorizes large integers in prime numbers. The left column shows the original code, annotated by the profiler with the estimated number of times each line is executed. The right-hand column shows the effect of the partitioning algorithm on these same functions.

The main result of the operation is the creation of a set of dedicated FUs, indicated by the `HW` label. A set of functions (`HW(1)` to `HW(4)`) represent code that has been transformed into hardware. These functions usually represent dedicated instructions that are used often in the program and are annotated with the performance increase they allow, their execution time, the size of hardware required for their implementation, and how often they are used within the code (note that, in the figure, these numbers correspond to the entire program and not just to the functions shown).

ORIGINAL CODE:

```
function isPrime(u) #28211
{
  if(u == 2) #28211
  {
    return 1; #5000
  }
  max = bin_sqrt(u); #28211
  if(u % 2 != 0 && u > 2) #23211
  {
    for(i = 3; i < max; i + 2) #105795
    {
      if((u % i) == 0) #95302
      {
        return 0; #7719
      }
    }
    return true; #10493
  }
  return false; #4999
}

function log2(u) #164924
{
  x = 0; #164924
  while(u != 0) #1168952
  {
    u = u >> 1; #1004028
    x = x + 1; #1004028
  }
  return x; #164924
}

function isSquare(u) #94777
{
  low = 0; #94777
  high = 0; #94777
  x = 0; #94777
  if(u == 1) #94777
  {
    return true; #12315
  }
  low = (log2(u) >> 1) - 1; #82462
  high = (log2(u) >> 1) + 1; #82462
  low = pow(2, low); #82462
  high = pow(2, high); #82462
  while(true) #360086
  {
    x = (low + high) >> 1; #360086
    if(low > high) #360086
    {
      return false; #69115
    }
    if((x * x) < u) #290971
    {
      low = x + 1; #129104
    }
    else
    {
      if((x*x) > u) #161867
      {
        high = x - 1; #148520
      }
      else
      {
        return true: #13347
      }
    }
  }
}
```

FINAL CODE:

```
function isPrime (u)
{
  HW(1)
  HW(3)
  if (u % 2 != 0 & u > 2)
    {
      for (i = 3;i < max;i + 2)
        {
          if (u % i == 0)
            {
              return 0;   } }
    return 1;   }
  return 0;   }

[HW] [HW] - function log2 (u)
{
  x = 0;
  while (u != 0)
    {
      u = u >> 1;
      x = x + 1;   }
  return x;   }

function isSquare (u)
{
  low = 0;
  high = 0;
  x = 0;
  HW(1)
  low = log2(u) >> 1 - 1;
  high = log2(u) >> 1 + 1;
  low = pow(2, low);
  high = pow(2, high);
  while (1)
    {
      x = low + high >> 1;
      HW(4)
      if (x * x < u)
        {
          low = HW(2);   }
      else
        {
          if (x * x > u)
            {
              high = x - 1;   }
          else
            {
              return 1;   } } } }

HW(1) (Perf. gain : 1.0002143413697018)
        (Exec: 20.0, HW: 49.0, Used: 3)
if (v == 0)
  {
  return 1;   }

HW(2) (Perf. gain : 0.997791210375102)
        (Exec: 2.0, HW: 38.0, Used: 3)
i + 1

HW(3) (Perf. gain : 1.824315957433441)
        (Exec: 44.0, HW: 329.0, Used: 3)
max = bin_sqrt(u);

HW(4) (Perf. gain : 1.0013923806061944)
        (Exec: 81.0. HW: 72.0. Used: 1)
if (low > high)
  {
    return 0;   }
```

Figure 6. Sample results of partitioning.

In this case, the evolutionary algorithm has also decided to transform an entire function (`log2`) in hardware and to build a dedicated FU that implements directly the entire function. This kind of operation is usually applied to small functions that are executed a large numbers of times in the program (as defined by the profiler).

Program name	Genes [bits]	Max HW inc. [slices]	Est. HW inc. [slices]	Estimated speedup	Run time [ms]
FACT	571	∞ 20% 10 %	213 % 19.92% 9.87%	5.69 2.92 1.81	3250 2821 4100
DCT	212	∞	73.73 %	2.77	547
RND100	100	∞	1.4 %	1.23	250
RND200	200	∞	1 %	1.08	734

Figure 7. Evolution results on various programs (mean value of 500 runs).

The results of the evolutionary run that produced the partition shown are significant and can be quantified by means of the estimated *speedup* and *hardware increase*. The speedup is computed by comparing the software-only solution to the final partition and the hardware increase represents the number of *Virtex* slices added to the software-only solution to obtain the final partition. Given a maximum hardware increase of 10% (i.e., the final processor had to be at most 10% larger than the minimal processor, and in fact was 8% larger in the individual shown), in this example evolution was able to find a partition that provides an estimated speedup of 227% in the execution time of the program.

5.3. Performance

To show the efficiency of our partitioning method we tested it on two benchmark programs and several randomly-generated ones. The size of the applications tested lies between 60 lines for the `DCT` program, which is an integer direct cosine transform, and 300 lines of code for the `FACT` program. The last kind of programs tested are random generated programs with different genome sizes.

Fig. 7 sums up the experiments that have been conducted to test our algorithm. Each figure in the table represents the mean of 500 runs. It is particularly interesting to note that all the results were obtained in the order of a few seconds and that the algorithm converged to very efficient solutions during that time.

Unfortunately, even if the domain is a rich source of literature, a direct comparison of our approach to others seems very difficult. Indeed, the large differences that exist in the various design environments and the lack of common benchmarking techniques (which can be explained by the different inputs of HW/SW partitioners that may exist) have already been identified in [12] to be a major difficulty against direct comparisons.

Obviously, the scalability of our approach should be verified on larger programs. However, it should also be kept in mind that our cellular arrays are meant to operate as configurable co-processing units attached to more conventional processors and as such target applications based on a

highly-parallel execution of computationally intensive kernels of code, such as those found in scientific or multimedia applications. These kernels, while sometimes very complex, rarely require extremely large code: in this context, to be effective in real-world applications our algorithm only has to scale up to a relatively small program size.

6. Conclusions and future work

In this article we presented an approach to the co-evolution of program code and processor architecture. Studied to be included within a design environment for bio-inspired systems, the approach reflects rather accurately the co-evolution of cells and genomes in biological systems and is quite efficient for the design of custom processors.

In particular, we presented the results we obtained by applying an evolutionary algorithm to the co-design of a *Move* processor and of the code it has to execute. These results, valuable in themselves, must also be regarded in the context of the development of a research tool. Elements such as a versatile user interface to allow the exploration of evolutionary parameters and the annotation of results to increase their interpretability represent then important outcomes of our research.

The next phase of our research aims at integrating the algorithm into a working design environment, relaxing some of the restrictions currently imposed on the programs to be co-evolved and streamlining the process of generating the VHDL code that implements the cellular processors. In a second stage, we will examine the complex issues associated with the design of complex arrays of communicating processors to identify other areas where evolution can be usefully exploited for the design of bio-inspired hardware.

References

[1] M. Arnold and H. Corporaal. Designing domain-specific processors. In *Proceedings of the 9th International Workshop on Hardware/Software Codesign*, pages 61–66, Copenhagen, April 2001.

[2] V. Catania, M. Malgeri, and M. Russo. Applying fuzzy logic to codesign partitioning. In *IEEE Micro*, 1997.

[3] H. Corporaal. *Microprocessor Architectures : from VLIW to TTA*. Wiley and Sons, 1997.

[4] R. P. Dick and N. K. Jha. MOGAC: a multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems. *IEEE Transacations on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):920–935, October 1998.

[5] P. Eles, K. Kuchcinski, Z. Peng, and A. Doboli. System level hardware/software partioning based on simulated annealing and tabu search. *Design Automation for Embedded Systems*, 2:5–32, 1997.

[6] R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for microcontrollers. In *IEEE. Design & Test of Computers*, pages 64–75, December 1993.

[7] R. Gupta and G. D. Micheli. System-level synthesis using re-programmable components. In *Proc. European Design Automation Conference*, pages 2–7, August 1992.

[8] J. Harkin, T. M. McGinnity, and L. Maguire. Genetic algorithm driven hardware-software partitioning for dynamically reconfigurable embedded systems. *Microprocessors and Microsystems*, 25(5):263–274, August 2001.

[9] J. Henkel and R. Ernst. High-level estimation techniques for usage in hardware/software co-design. In *ASP-DAC*, pages 353–360, 1998.

[10] J. Hoogerbrugge and H. Corporaal. Transport-triggering vs. operation-triggering. In *International Conference on Compiler Construction*, Edinburgh, 1994.

[11] J. Hou and W. Wolf. Process partitioning for distributed embedded systems. In *Fourth International Workshop on Hardware/Software codesign*, pages 70–76, March 1996.

[12] M. López-Vallejo and J. C. López. On the hardware-software partitioning problem: System modeling and partitioning techniques. *ACM Transactions on Design Automation of Electronic Systems*, 8(3), July 2003.

[13] D. Mange, M. Sipper, A. Stauffer, and G. Tempesti. Towards robust integrated circuits: The embryonics approach. *Proceedings of the IEEE*, 88(4):516–541, 2000.

[14] D. Mange, A. Stauffer, E. Petraglio, and G. Tempesti. Embryonic machines that divide and differentiate. In *Proc. 1st Int. Workshop on Biologically Inspired Approaches to Advanced Information Technology (BioADIT04)*, 2004.

[15] H. Oudghiri and B. Kaminska. Global weighted scheduling and allocation algorithms. In *European Conference on Design Automation*, pages 491–495, March 1992.

[16] H. Pearson. The regeneration gap. *Nature*, (414):388, 2001.

[17] V. Srinivasan, S. Radhakrishnan, and R. Vemuri. Hardware software partitioning with integrated hardware design space exploration. pages 28–35, Paris, France, 1998.

[18] G. Tempesti, D. Mange, and A. Stauffer. A robust multiplexer-based fpga inspired by biological systems. *Journal of Systems Architecture*, 43(10):719–733, 1997.

[19] G. Tempesti, P.-A. Mudry, and R. Hoffmann. A move processor for bio-inspired systems. In *NASA/DoD Conference on Evolvable Hardware (EH05)*, pages 262–271, Los Alamitos, June 2005. IEEE Computer Society Press.

[20] F. Vahid and D. Gajski. Incremental hardware estimation during hardware/software functional partitioning. *IEEE Transactions on VLSI Systems*, 3(3):459–464, September 1995.

[21] F. Vahid, J. Gong, and D. Gajski. A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning. In *Proc. EURODAC*, pages 214–219, 1994.

[22] T. Wiangtong. *Hardware/Software Partitioning And Scheduling For Reconfigurable Systems*. PhD thesis, Imperial College London, February 2004.

[23] T. Wiangtong, P. Y. Cheung, and W. Luk. Comparing three heuristic search methods for functional partitioning in hardware-software codesign. *Design Automation for Embedded Systems*, 6(4):425–449, July 2002.