A Dynamically Constrained Genetic Algorithm For Hardware-software Partitioning

Pierre-André Mudry   Guillaume Zufferey   Gianluca Tempesti

École Polytechnique Fédérale de Lausanne
Cellular Architectures Research Group
Station 14, 1015 - Lausanne, Switzerland
Email : pierre-andre.mudry@epfl.ch

ABSTRACT

In this article, we describe the application of an enhanced genetic algorithm to the problem of hardware-software codesign. Starting from a source code written in a high-level language our algorithm determines, using a dynamically-weighted fitness function, the most interesting code parts of the program to be implemented in hardware, given a limited amount of resources, in order to achieve the greatest overall execution speedup. The novelty of our approach resides in the tremendous reduction of the search space obtained by specific optimizations passes that are conducted on each generation. Moreover, by considering different granularities during the evolution process, very fast and effective convergence (in the order of a few seconds) can thus be attained. The partitioning obtained can then be used to build the different functional units of a processor well suited for a large customization, thanks to its architecture that uses only one instruction, *Move*

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids; C.0 [**Computer Systems Organization**]: General—*systems specification methodology*

General Terms

Algorithms, design

Keywords

Constrained Hardware–Software partitioning, TTA processor, genetic algorithm

1. INTRODUCTION AND MOTIVATIONS

As very efficient heuristics, genetic algorithms (GAs) have been widely used to solve complex optimization problems. However, when the search space to be explored becomes very large, this technique becomes unapplicable or, at least, inefficient. This is the case when GAs are applied to the *partitioning* problem, which is one of the tasks required for the hardware-software codesign of embedded systems.

Consisting in the realization, at the same time, of the hardware and the software layers of an embedded system, codesign has been used since the early 90s and is now a technique widely spread in the industry. This design methodology permits to exploit the different synergies of hardware and software that can be obtained for a particular embedded 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 consist in different tasks, as defined in [8]: partitioning, co-synthesis, co-verification and co-simulation.

In this article, we will focus on the complex, NP-complete [15], partitioning problem that can be defined as follows: starting from a program to be implemented on a digital system and given a certain execution time and/or size constraints, the partitioning task consists in the determination of which parts of the algorithm have to be implemented in hardware in order to satisfy the given constraints.

As this problem is not new, several methods have been proposed in the past to solve it: Gupta and De Micheli start with a full hardware implementation [7], whilst Ernst *et al.* [6] use profiling results in their Cosyma environment to determine with a simulated annealing algorithm which blocks to move to hardware. Vahid *et al.* [20] use clustering together with a binary-constrained search to minimize hardware size while meeting constraints. Others have proposed approaches like fuzzy logic [2], genetic algorithms [4][17], hierarchical clustering [12] or tabu search [5] to solve this task.

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 such as simulated annealing to solve the partitioning task [21][22], they can be hybridized to take into account the particularities of the problem and solve it efficiently. The improved genetic algorithm we propose starts from a software tree representation and progressively builds a partition of the problem by looking for the best compromise between raw performance and hardware area increase. In other words, it tries to find the most interesting parts of the input program to be implemented in hardware, given a limited amount of resources.

The novelty of our solution resides in the multiple optimization steps applied on the population at each generation along with a dynamically-weighted fitness function. Thus, we obtain an hybridized algorithm that explores only the most interesting parts of the solution space and, when good candidates are found, refines them as much as possible to extract their potential.

This paper is organized as follows: in the next section we briefly present the TTA processor architecture that serves as a target platform for our algorithm. The following section is dedicated to the formulation of the problem in the context of a genetic algorithm and section 4 describes the specific enhancements that are applied to the standard GA approach. Afterwards, we present some experimental results which show the efficiency of our approach. Finally, section 6 concludes this article and introduces future work.

2. THE TTA PARADIGM

We have developed our new partitioning method in the context of the *Move* processor paradigm [1] [3] which will be briefly introduced here. However, our approach remains general and could be used for different processor architectures and various reconfigurable systems with only minor changes.

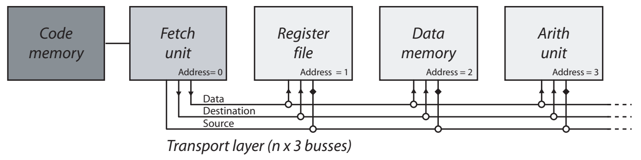


Figure 1: General architecture of a *TTA* processor.

The *Move* architecture, which belongs to the class of transport triggered architectures (TTA), presents some interesting characteristics. This family of architectures was originally intended for the design of application-specific dataflow processors (processors where the instructions define the flow of data, rather than the operations to be executed).

In many respects, the overall structure of a TTA-based system is fairly conventional: data and instructions are fetched to the processor from the main memory using standard mechanisms (caches, memory management units, etc. . .) and are decoded 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 together by one or more *transport busses*. All computation is carried out by the functional units (examples of such units can be adders, multipliers, register files, etc.) and the role of the instructions is simply to move data to and from the FUs in the order required to implement the desired operations. Since all the functional units are uniformly accessed through input and output registers, instruction decoding is reduced to its simplest expression, as only one instruction is needed: `move`.

TTA move instructions trigger operations that in fact correspond to normal RISC instructions. For example, a RISC `add` instruction specifies two operands and, most of the time, a result 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 functional unit and used wherever needed.

The *Move* approach, in and for itself, does not imply high performance, but several arguments in favor of TTAs have been proposed [3][11]:

- The register file traffic is reduced because the results can be moved directly from one FU to another;
- Fine-grained instruction level parallelism (ILP) is achievable through VLIW encoded instructions;
- Data moves are determined at compile time, which could be used to reduce power consumption;

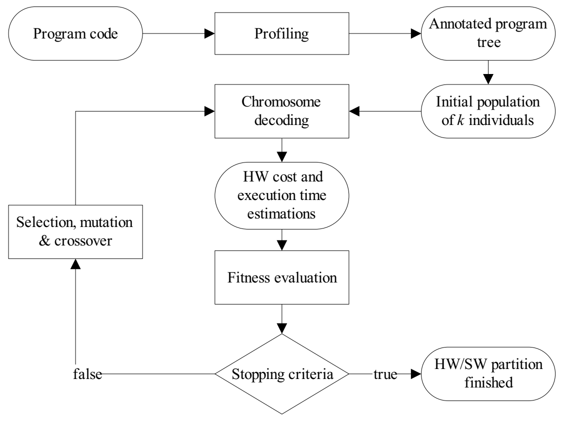


Figure 2: General flow diagram of our genetic algorithm.

- New *instructions*, in the form of functional units (FU), can be added easily.

The latter advantage, along with the fact that the architecture handles the functional units as *"black boxes"*, i.e. without any inherent knowledge of their functionality, implies that the internal architecture of the processor can be described as a *memory map* which associates the different possible operations with the addresses of the corresponding functional units.

This feature, coupled with the algorithm described in this paper, introduce 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. A soft-core processor based on this concept has been previously developed in [18] to explore various bio-inspired paradigms. Among other things, this architecture also has been identified as a good candidate for building *ontogenetic* processors [18], that is, processors that could self-assemble from basic building blocks according toa small set of instructions.

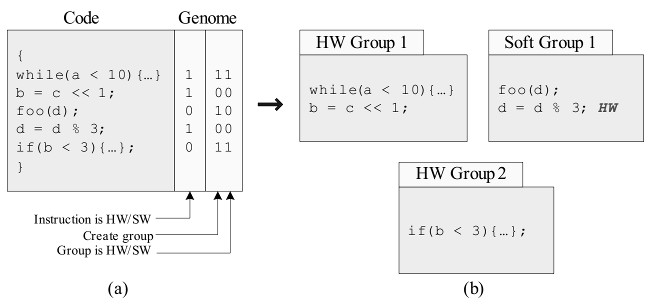
Because of the versatility of *Move* processors, automatic partitioning becomes indeed very interesting for the synthesis of ontogenetic, application-specific processors: the partitioning can automatically determine which parts of the code of a given program are the best candidates to be implemented as FUs that can then be inserted in the memory map of the processor.

3. A BASIC GENETIC ALGORITHM FOR PARTITIONING

We describe in this section the basic GA that serves as a basis for our partitioning method and that will be be enhanced in section 4 where the specific improvements we have introduced will be described. The basic algorithm, whose flow diagram is depicted on Fig. 2, works as follows: starting from a program written in a specific language resembling C, a syntactic tree is built and then analyzed by the GA which then produces a valid, optimized partition. The various parameters of the GA can be specified on the graphical user interface that has been designed, like every other software described here, in *Java*.

3.1 Programming language and profiling

Assembly could have been used as an input for our algorithm but the general structure of a *Move* assembly program is difficult to capture because every instruction is considered only as a data



```
{
while(a < 10){…}   1  11
b = c << 1;        1  00
foo(d);            0  10
d = d % 3;         1  00
if(b < 3){…};      0  11
}
```

```
HW Group 1:
while(a < 10){…}
b = c << 1;

Soft Group 1:
foo(d);
d = d % 3; HW

HW Group 2:
if(b < 3){…};
```

Figure 3: Genome encoding.

displacement, introducing a great deal of complexity in the representation of the program's functionality. Thus, the programs to be evolved by the GA are written in a simplified programming language that supports all the classical declarative language constructs in a syntax resembling C. Several limitations have however been imposed to this programming language:

1. Pointers are not supported;
2. Recursion is forbidden;
3. No typing exists (all values are treated as 32 bits integers). As a result, only fixed-point or integer calculations can be conducted.

These simplifications permitted us to focus on the codesign partitioning problem without having to cope with unrelated complications. However, it should be noted that these limitations could be lifted in a future release of our partitioner.

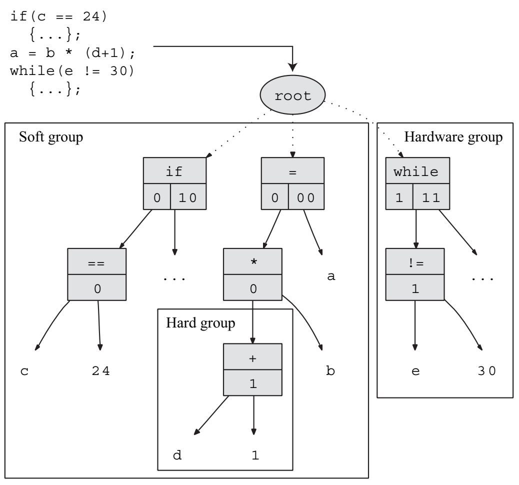
Prior to being used as an input for the partitioner, the code needs to be *annotated* with code coverage information. To perform this task, we use standard profiling tools on a *Java* equivalent version of the program. This step provides an estimation of how many times each line is executed for a large number of realistic input vectors. With the data obtained, the general program execution scheme can be estimated, which will allow the GA to evaluate the most interesting kernels to be moved to hardware.

3.2 Genome encoding

Our algorithm starts by analyzing the syntax of the annotated source code. It then generates the corresponding program tree, which will then constitute the main data structure the algorithm will work with. From this structure, it builds the *genome* of the program, which consists of an array of boolean values. This array is constructed by associating to each node of the tree a boolean value indicating if the subtree attached to this node is implemented in hardware (Fig. 3, column a). Since we also want to regroup instructions together to form new FUs, to each statement[1] correspond two additional boolean values that permit the creation of groups of adjacent instructions (Fig. 3, column b). The first value indicates if a new group has to be created and, in that case, the second value indicates if the whole group has to be implemented in hardware (i.e. to create a new FU).

The complete genome of the program is then formed by the concatenation of the genomes of the single nodes. An example of a program tree with its associated genome is represented on Fig. 4, which depicts the different possible groupings and the representation of the data the algorithm works with.

[1]Statements are assignments, *for*, *while*, *if*, function calls…



```
if(c == 24)
  {...};
a = b * (d+1);
while(e != 30)
  {...};
```

Figure 4: Creation of groups according to the genome.

3.3 Genetic operators

3.3.1 Selection

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. In order to ensure a larger population diversity, part of the new population is not obtained by reproduction but by random generation, allowing a larger exploration of the search space.

3.3.2 Mutation

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

1. A new functional unit is created;
2. An existing functional unit is destroyed. The former hardware group is then implemented in 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.

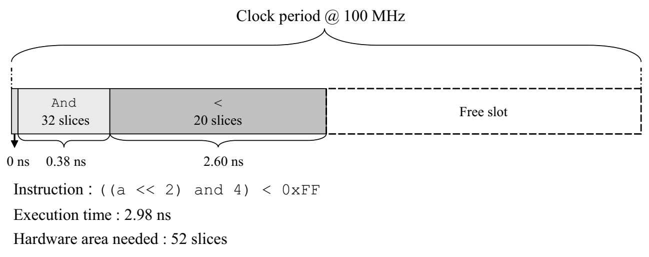
3.3.3 Crossover

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.

3.4 Determining hardware size and execution time

Computing hardware size and execution time is one of the key aspects of the algorithm, as it defines the fitness of an individual. Different techniques exist to determine these values, for example in [9] or in [19]. The method we chose to use is based on a very fine characterization of each hardware elementary building block of the targeted hardware platform. In the current implementation we use a Virtex® II field-programmable gate array (*FPGA*), which is a programmable chip containing logic elements that can be configured to act like processors or other digital circuits.

The characterization of each of these building blocks that conduct very simple logical and arithmetic operations (`AND`, `OR`, $+$, ...) allows then to arrange them together to elaborate more complex operations that form new FUs in the *Move* processor. For example, it is possible to reduce the execution of several software instructions to only one clock cycle by chaining them in hardware as depicted on Fig. 5 (note that the shift operation used in the example is "free" (no slices[2] are used) in hardware because only wires are required to achieve the same result). This simple example shows the principles of how the basic blocks are chained and how hardware size and execution time are predicted.



Instruction : `((a << 2) and 4) < 0xFF`
Execution time : 2.98 ns
Hardware area needed : 52 slices

Figure 5: Hardware time and size estimation principle of a software instruction.

The basic blocks' size and timing metrics have been determined using the Synplify Pro® synthesis solution coupled, in some cases, with the Xilinx® place-and-route tools. Thus, we have obtained the number of slices of the FPGA required to implement each block and the length of the critical path of each basic block. Because this characterization mostly depends on the architecture targeted and on the software used, it has to be redone for each different hardware platform targeted.

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 for example. 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 program 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: if it is located at the root of a group, it constitutes a new FU and some computation is needed. In fact, the time to move the data to the new FU and the size of the registers required for the storage of the local variables have to be taken into account. Moreover, as every FU is connected to the rest of the processor using a standard bus interface, its cost also has to be considered. Finally, if this unit is used several times, its hardware size has to be counted only once: to determine if the generated FU is new, its sub-trees are compared to the ones belonging the pool of the already available FUs.

[2]Slices are the fundamental elements of the FPGA. They characterize the how much space for logic is available on a given circuit. The name and implementation of these elements differ from one vendor to one another.

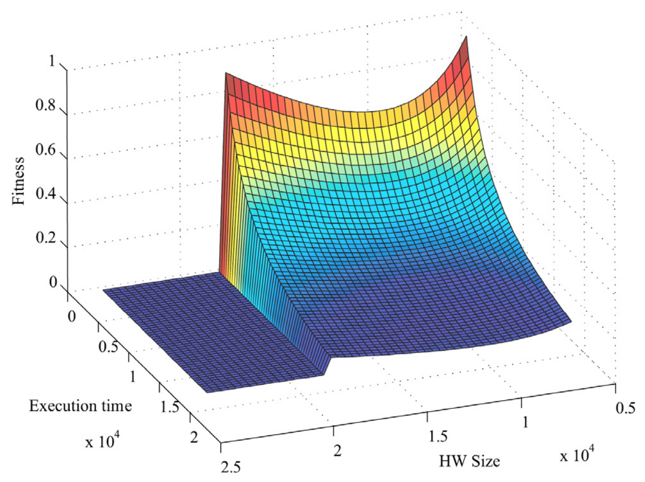


Figure 6: Ideal fitness landscape shape.

3.5 Fitness evaluation

3.5.1 *A static fitness function*

The objective of the GA is to get the partitioning with the smallest execution time whilst remaining smaller than an area constraint. To achieve this, the fitness function used to estimate each individual needs to have high values for the candidates that balance well the compromise between hardware area and execution speed. Because we made the assumption that the basic solution for the partitioning problem relies on a whole software implementation (that is, using only a simple processor that contains the minimum of hardware required to execute the program to be partitioned), we use a relative fitness function. This means that this simple processor, whose hardware size is β, has a fitness of 1 and the fitness of the discovered solutions are expressed in terms of this trivial solution. We also define α, the time to execute the given program on this trivial processor. For an individual having a size s and requiring a time t to be executed, the following fitness function can then be defined:

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

where $hwLimit$ is the maximum hardware size allowed to implement the processor with the new FUs defined by the partitioning algorithm.

The first ratio appearing in the top equation corresponds to the speedup obtained with this individual and the second ratio corresponds to its hardware size increase. Therefore, the following behaviour can be achieved: when the speed increase obtained during one step of the evolution is relatively bigger than the hardware increase needed to obtain this new performance, the fitness increases. In other words, the hardware investment for obtaining better performance has to be small enough to be retained.

3.5.2 *A dynamic fitness function*

One drawback of the static fitness function is that it does not necessarily use the entire available hardware. As this property might be desirable, particularly when a given amount of hardware is available and would be lost if not used, we introduce here a dynamically weighted fitness function that can cope with such situations. In fact, we have seen that the static fitness function increases only when the hardware investment is balanced by a sufficient speedup.

To go further, our idea is to push evolution towards solutions that use more hardware by modifying the balance between hardware size and speedup in the fitness function. This change has to be done only when a relatively good solution has been found, as we do not want the algorithm to be biased towards solutions with a large hardware cost at the beginning of the evolution.

To achieve this goal, a new dynamic parameter is added to the static fitness function and permits more expensive blocks to be used as good solutions are found. For an individual having an hardware size of s, we first compute the adaptive factor k using the following equation:

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

We then compute the individual fitness using that adaptive factor in the a refined fitness function:

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

where α, β, and $hwLimit$ have the same meaning as in the static function. Thus, we obtain the fitness landscape shown on Fig. 6, which clearly shows the decrease of the fitness when a given $hwLimit$ (on the example given, about 19000) is exceeded. The figure also clearly shows the influence of the k factor which is responsible for the peak appearing near the $hwLimit$.

4. AN HYBRID GENETIC ALGORITHM

All the approaches described in the introductory section work at a specific *granularity level*[3] that does not change during the codesign process, that is, these partitioners work well only for certain types of inputs (task graphs for example) but cannot be used in other contexts. However, more recent work [10] has introduced techniques that can cope with different granularities during the partitioning. Because of the enormous search space that a real-world application generates, it is difficult for a generic GA such as the one we just presented to be competitive against state-of-the-art partitioning algorithms. However, we will show in the rest of this section that it is possible to hybridize (in the sense of [16]) the presented GA to considerably improve its performance.

4.1 Leveling the representation via hierarchical clustering

One problem of the basic GA described above lies in the fact that it implicitly favors the implementation in hardware of nodes close to the root. In fact, when a node is changed to hardware its whole sub-tree is also changed and the genes corresponding to the subnodes 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 resides in the decomposition of the program tree into different levels that correspond to *blocks* in the program[4], as depicted on Fig. 7. Function calls have the level of the called function's block and a block has level $n + 1$ if the highest level of the block or function calls it contains is n, the deepest blocks being at level 0 by definition. 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.

[3]Function level, control level, dataflow level, instruction level...

[4]Series of instructions delimited by brackets

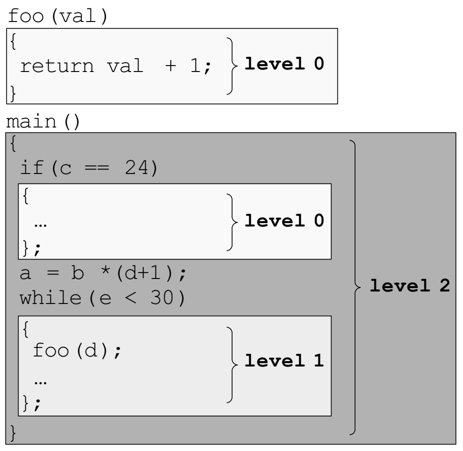


Figure 7: Levels definition.

The GA is recursively applied to each level, starting with the deepest ones ($n = 0$). 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. By examining the problem at different levels we obtain different granularities for the partitioning. As a result, with a single algorithm, we cover levels ranging from instruction level to process level (cf. [10] for a definition of these terms). 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. By doing so, the search time is greatly reduced while preserving the global quality of the solution.

4.2 Pattern-matching optimization

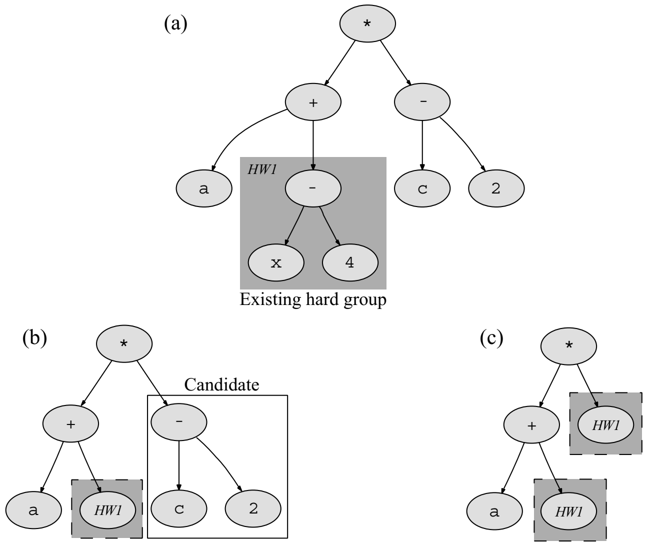


Figure 8: Candidates for pattern-matching removal.

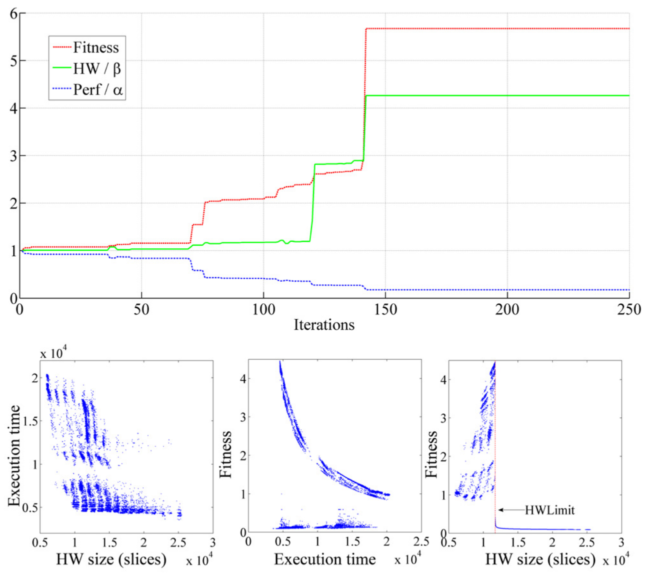


Figure 9: Exploration during the evolution.

A very hard challenge for evolution is to find *reusable* functional units that can be employed at different locations in a program. Two different reasons explain this difficulty, the first being that even if a block could be used elsewhere within the tree, the GA has to find it only by random mutations. The second reason is that it is possible that, although one FU might not be interesting when used once, it would become so when reused several times because the initial hardware investment has to be made only once.

To help the 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. This situation is depicted on Fig. 8: starting from an implementation using one FU (Fig. 8.a), this step searches for candidates sub-trees that show a structure similar to the existing FU. A perfect match is not required: variables values, for example, are passed as parameters to the FU and can differ (Fig. 8.b). Finally, the software sub-tree is simply replaced by a *call* to that FU (Fig. 8.c). Reusability is thus greatly improved because only one occurrence of a block has to be found, the others being given by this new step.

4.3 Non-optimal block pruning

Another help is given to the algorithm by cleaning the best individual of each generation. This is done by removing all the non-optimal hardware blocks from the genome. These blocks are detected by computing, for each block or group of similar blocks, the fitness of the individual when that part is implemented in software. If the latter is bigger or equal than the original fitness, it means that the considered block does not increase or could even decrease the fitness and is therefore useless. The genome is thus changed so that the part in question is no longer implemented as a functional unit.

This particular step, could be considered as a *cleaning pass*, was added to remove blocks that were discovered during evolution but that were not useful for the partition.

5. EXPERIMENTAL RESULTS

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, which factorizes large integer in prime numbers. The last kind of programs tested are random generated programs with different genome sizes. The quality of our results 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 slices in the VIRTEX-II 3000 that have to be added to the software-only solution to obtain the final partition.

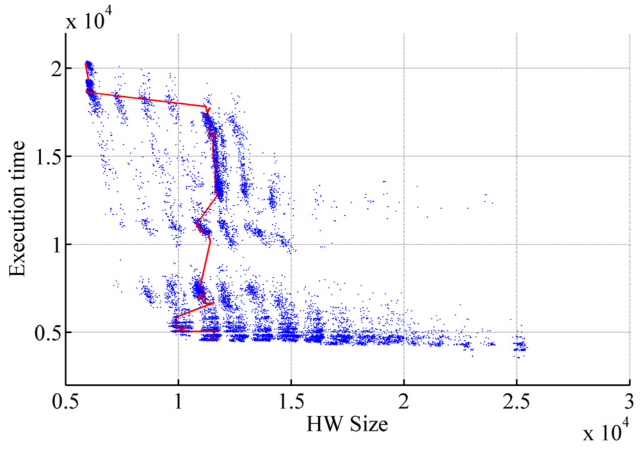


Figure 10: Best individual trace along with the explored fitness landscape

Fig. 9 depicts the evolution, using 40 iterations per level, of 30 individuals for the `FACT` program. A maximum hardware increase of 20% has been specified. We can see that the exploration space is well covered during evolution. Fig. 10 shows the coverage of the fitness landscape during evolution along with the best individual trace for the same program.

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 11: Evolution results on various programs (mean value of 500 runs).

Figure 11 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 not minutes or hours as it is usually the case when GAs are involved and that the algorithm converged to very efficient solutions during that time.

Unfortunately, even if the domain is the source of a rich 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 [13] to be a major difficulty against direct comparisons.

6. CONCLUSIONS AND FUTURE WORK

In this article we described an implementation of a new partitioning method using an *hybrid GA* that is able to solve relatively large and constrained problems in a very limited amount of time. However, albeit our method is tailored for a specific kind of processor architecture, it remains general and could be used for almost every embedded system architecture with only minor changes.

This work was done in the context of the development of an automatic software suite for bio-inspired systems generation in which *Move* processors would be used as ontogenetic processors that could be assembled from different buildings blocks. In this paper, we presented a method to automatically generate such blocks (i.e. FUs).

The usage of a dynamically-weighted fitness function introduced some flexibility in the GA and permitted to closely meet the constraints whilst maintaining an interesting performance. By using several optimization passes, we reduced the search space and made it manageable by a GA. Moreover, the granularity of the partitioning is determined dynamically rather than fixed before execution thanks to hierarchical clustering. The different levels determined by this technique constitute thus problems of growing complexity that can be handled more easily by the algorithm.

The results presented here, as well as those of others groups, who have shown that HW/SW partitioning can be successfully used for FPGA soft-cores [14], encourage us to pursue our research in order to address the unresolved issues of our system: for example, although the language in which the problem has to be specified remains simple, we are currently working on an automatic converter for C which would give us the opportunity to directly test our method on well-known benchmarking suites.

Future work within the project calls for two main axes of research. On one hand it would be interesting to introduce energy as a parameter for the fitness function in order to optimize the power-consumption of the desired embedded circuit. On the other hand, we are also exploring the possibility of automatically generating the HDL code corresponding to the extracted hardware blocks, a tool that would allow us to verify our approach on a larger set of problems and also on real hardware.

7. 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, April 2001.

[2] V. Catania, M. Malgeri, and M. Russo. Applying fuzzy logic to codesign partitioning. *IEEE Micro*, 17(3):62–70, 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 Transactions 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, 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. Henkel and R. Ernst. An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(2):273–289, April 2001.

[11] J. Hoogerbrugge and H. Corporaal. Transport-triggering vs. operation-triggering. In *Proceedings 5th International Conference Compiler Construction*, pages 435–449, 1994.

[12] J. Hou and W. Wolf. Process partitioning for distributed embedded systems. In *CODES '96: Proceedings of the 4th International Workshop on Hardware/Software Co-Design*, page 70. IEEE Computer Society, 1996.

[13] 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.

[14] R. Lysecky and F. Vahid. A study of the speedups and competitiveness of FPGA soft processor cores using dynamic hardware/software partitioning. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 18–23. IEEE Computer Society, 2005.

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

[16] J.-M. Renders and H. Bersini. Hybridizing genetic algorithms with hill-climbing methods forglobal optimization: two possible ways. In *Proc. of the First IEEE Conference on Evolutionary Computation*, volume 1, pages 312–317, June 1994.

[17] V. Srinivasan, S. Radhakrishnan, and R. Vemuri. Hardware/software partitioning with integrated hardware design space exploration. In *DATE '98: Proceedings of the conference on Design, automation and test in Europe*, pages 28–35. IEEE Computer Society, 1998.

[18] 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. IEEE Computer Society Press, June 2005.

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

[20] 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.

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

[22] 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.