A hybrid genetic algorithm for constrained hardware-software partitioning

Pierre-André Mudry Guillaume Zufferey Gianluca Tempesti

École Polytechnique Fédérale de Lausanne (EPFL)
Cellular Architectures Research Group, Station 14
CH-1015 Lausanne
pierre-andre.mudry@epfl.ch

Abstract

In this article, we propose a novel partitioning method for hardware-software codesign based on a genetic algorithm that has been enhanced for this specific task. Given a high-level program and an area constraint, our software considers different granularities levels to discover the most interesting blocks to be implemented in ad hoc *functional units that can then be used as new instructions in a* Move *processor. Various optimizations are conducted to obtain a clean, very fast (in the order of a few seconds) and efficient partitioning on programs ranging from a few to several hundreds of lines of code.*

1 Introduction and motivation

The codesign of complex digital systems has been successfully used since the early 90s. This approach of designing systems from the hardware and the software standpoints at the same time is now widely accepted in the industry where cheap, reliable and fast systems are needed.

Such systems are usually built around a core processor containing hardware modules that can be tailored for a specific application and can then exploit the synergism of hardware and software. This ”tailoring” corresponds to the *codesign* of the system and can be divided in several subtasks [7]: partitioning, co-synthesis, co-verification and co-simulation.

In this article, we will focus on the partitioning task, which can be stated as follows: starting from a program and a certain number of time and size constraints, the partitioning task consists in determining which parts of the program are the best candidates to be implemented in hardware in order to minimize the execution time and match the constraints. Several different methods to solve this task have been developed: Gupta and De Micheli start with a full hardware implementation [6] whilst Ernst *et al.* [5] use profiling results in their Cosyma environment to determine with a simulated annealing algorithm which blocks to move to hardware. Vahid *et al.* [18] use clustering together with a binary-constraint search to minimize hardware size while meeting constraints. Others have proposed approaches like fuzzy logic [2], genetic algorithms [15], hierarchical clustering [11] or tabu search [4] to resolve this task.

We chose to work with a genetic algorithm (GA) because of the complex, NP-complete [14], nature of the partitioning task. In fact, GAs are very good heuristics to find solutions to complex optimization problems. Although some attempts to use genetic algorithms have been shown to be less efficient than other search methods for hardware-software codesign in [19], we propose here an improved genetic algorithm that is able to solve this difficult task in a very efficient manner.

Starting from the tree representation of a program, this new algorithm builds a solution realizing the best compromise between raw performance gain and hardware area increase. In other words, we try to find the most interesting parts to be implemented in hardware, given a limited amount of resources. The novelty of our approach lies in the several optimizations passes applied to the intermediary results of a standard GA, which permits to avoid the most common pitfalls associated with GAs, such as being trapped in local minima. Thus we will show that our algorithm is robust and performs well on relatively large programs by converging to nearly optimal solutions.

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 classical GA approach. Afterwards, we present some experimental results that 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 minor changes.

The *Move* architecture, which belongs to the class of transport triggered architectures (TTA), presents some interesting characteristics and a soft-core processor based on this concept has been previously developed in our group [16] to explore various bio-inspired paradigms.

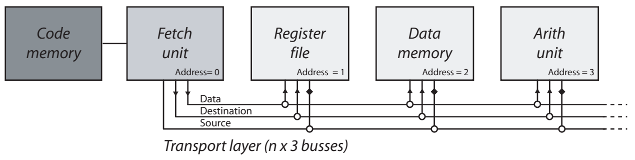


Figure 1. General architecture of a *TTA* processor.

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`.

Several arguments in favor of TTAs have been proposed by Corporaal [3] and Hoogerbrugge [10]:

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

A consequence of the TTA structure is that the internal architecture of the processor can be described as a *memory map* which associates the different possible operations with the address of the corresponding functional units. This feature along with the partitioning algorithm allows us to 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.

3 A basic genetic algorithm for partitioning

Because of the versatility of *Move* processors, automatic partitioning is interesting. In fact, the partitioning can automatically determine which parts of the code of a given program are the best candidate to be implemented as FUs.

We describe in this section the basic GA that serves as a basis for our partitioning method and that will be enhanced in section 4 where we will describe the specific improvements we have introduced. Fig. 2 depicts the flow diagram of this basic algorithm, which 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.

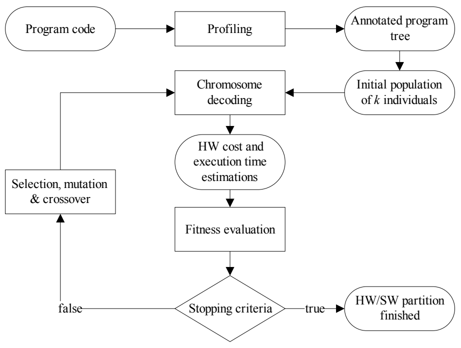


Figure 2. General flow diagram of our genetic algorithm.

3.1 Programming language and profiling

We could have used assembly 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 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 which supports all the classical declarative language constructs in a syntax resembling C. Several limitations have however been imposed for this programming language: pointers are not supported, recursion is forbidden and no typing exists (all values are treated as 32 bits integers). As a result, only fixed-point or integer calculations can be conducted.

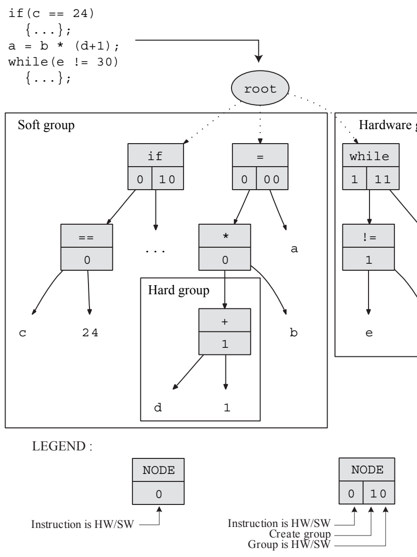
These simplifications permitted us to focus on the codesign partitioning problem without having to cope with unrelated complications. However, these limitations could be lifted in a future release of our partitioner.

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 estimation on how many times each line is executed for a large number of realistic input vectors. This step serves as a good estimate of the general program execution scheme and will permit 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 provided source code. It then generates the corresponding program tree which will then constitute the main data structure it will work with (Fig. 3). From this structure, it builds the *genome* of the program which consists of an array of boolean values. It 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. 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. The first

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



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

Figure 3. Creation of groups according to the genome.

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.

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 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 mutation rates are defined for the following cases:

1. A new functional unit is created;
2. An existing functional unit is destroyed;
3. A new group of statements is created or two groups are merged.

Using different mutation rates for the creation and the destruction of functional units is 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 subtrees. 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 [8] or [17]. The method we chose to use is based on a very fine characterization of each hardware elementary building block of the hardware platform targeted (in the current implementation, a Virtex® II FPGA). These blocks correspond to simple logical and arithmetical operations (AND, OR, $+, \ldots$), which can then be arranged together to elaborate more complex operations that form new FUs in the *Move* processor.

To estimate the size and timing of the FUs, we used the Synplify Pro® synthesis solution coupled, in some cases, with the Xilinx® place-and-route tools, to determine different performance and area metrics for each basic block[2].

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, pipelined dividers. Area estimators were built using the same principles. Using these parameters, determining size and time of each subtree is then relatively straightforward because only two different cases have to be considered:

1. For software subtrees, the estimation is done recursively over the nodes of the program tree, adding at each step the appropriate execution time and potential hardware unit[3].
2. For hardware subtrees, the computation is a bit more complex because it depends on the position of the considered subtree: if it is located at the root of a group, it constitutes a new FU and more computation is needed.

Of course, for each functional unit that has to be created, the costs of the instructions and registers required for moving and storing the variables used in this new units have to be added. Moreover, if this unit is used several times, its hardware size has to be counted only once.

3.5 Fitness evaluation

The objective of the GA is to get the partitioning with the smallest execution time whilst remaining smaller than a given 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

[2]Of course, this characterization would have to be redone for each different hardware platform targeted.

[3]e.g. the first time an add instruction is encountered, an add FU must be added to compose the minimal processor to execute this program.

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.

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.

4 An improved, hybrid genetic algorithm

All the approaches described in the introductory section work at a specific *granularity level*[4] 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 can not be used in other contexts. However, more recent work [9] 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 adapt the GA to considerably improve its performance.

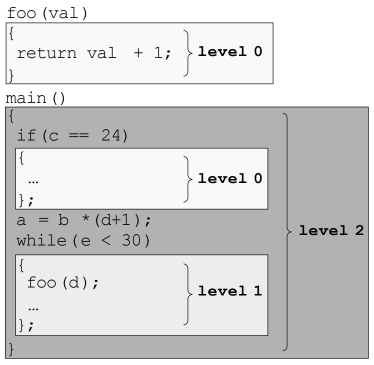
4.1 Levelling the representation via hierarchical clustering

One problem of the basic GA described above lies in the fact that it implicitly favours the implementation in hardware of nodes close to the root. In fact, when a node is changed to hardware its whole subtree is also changed and the genes corresponding to 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 subtree.

The solution we propose resides in the decomposition of the program tree into different levels that correspond to *blocks* in the program[5], as depicted on Fig. 4. 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 blocks 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.

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

[5]Series of instructions delimited by brackets



```
foo(val)
{
 return val  + 1;    } level 0
}
main()
{
 if(c == 24)
 {
  …                  } level 0
 };
 a = b *(d+1);                   } level 2
 while(e < 30)
 {
  foo(d);            } level 1
  …
 };
}
```

Figure 4. 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 nodes. By examining the problem at different levels we obtain different granularities for the partitioning. With a single algorithm, we cover levels ranging from instruction level to process level (c.f. [9] for a definition of these terms). This 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

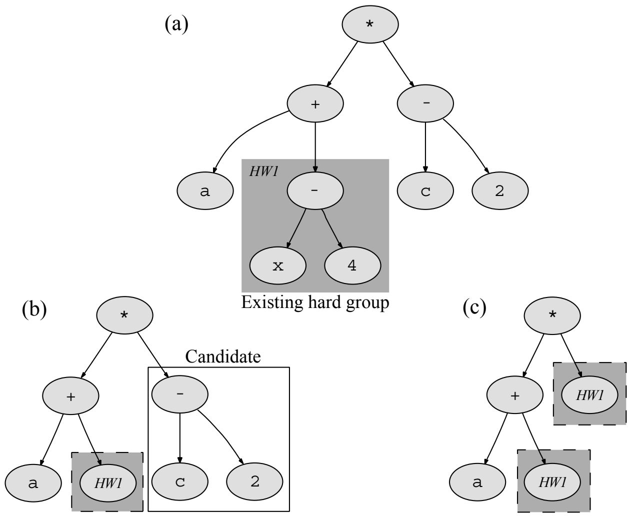


Figure 5. Candidates for pattern-matching removal.

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, while one FU might not be interesting when used once, it would become so when reused several times because the 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. 5. Reusability is then 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 step was added to remove blocks that were discovered during evolution but that were not useful for the partition.

5 Experimental results

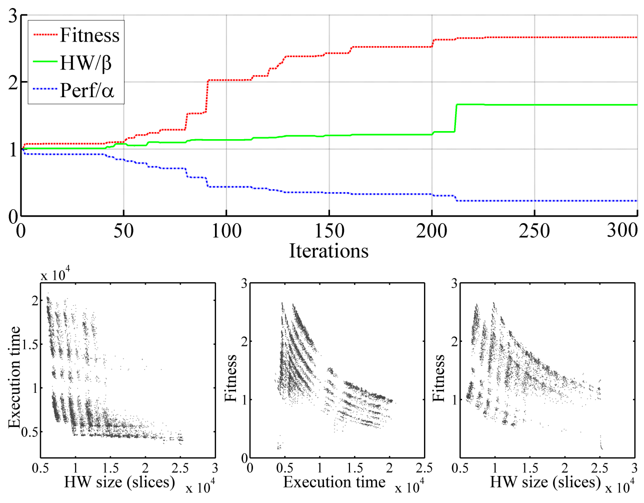


Figure 6. Exploration during the evolution.

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 kinds 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. Fig. 6 depicts the

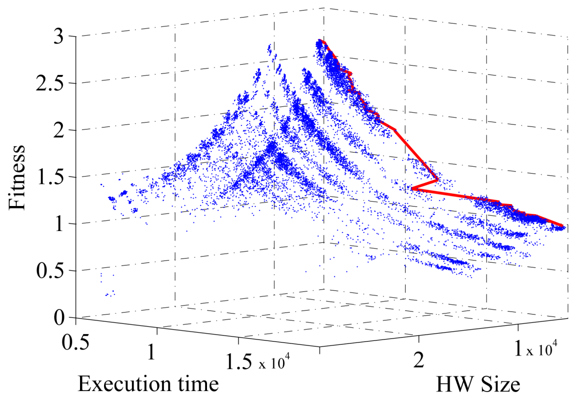


Figure 7. Best individual trace along with the explored fitness landscape.

evolution, using 40 iterations per level, of 25 individuals for the `FACT` program. Fig. 7 shows the coverage of the fitness landscape during evolution along with the best individual trace for the same program. We can see that the exploration space is well covered during evolution. Fig. 8 sums up the experiments that have been conducted to test our algorithm. Each figure in the table represents the mean of 500 runs. The performance differences between randomly generated programs and real programs can be explained by the lack of structure of random programs. Moreover, it is particularly interesting to note that all the results were obtained in the order of 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)

Program name	Genes [bits]	Max HW inc. [slices]	Est. HW inc. [slices]	Estimated speedup	Run time [ms]
`FACT`	571	∞	65.17 %	4.41	3447
		20%	19.75%	3.00	3750
		10 %	9.66 %	2.38	3864
`DCT`	212	∞	71.37 %	2.77	547
`RND100`	100	∞	1.4 %	1.23	250
`RND200`	200	∞	1 %	1.08	734

Figure 8. Evolution results on various programs (mean value of 500 runs).

have already been identified in [12] to be a major difficulty in 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, constrained problems in a very limited amount of time. By using several techniques, we reduced the search space and made it manageable by a GA. Our method, albeit tailored for a specific kind of processor architecture, remains general and could be used for almost every embedded systems architecture.

This work was done in the context of the development of an automatic software suite for bio-inspired systems generation. The results presented in this paper, as well as those of others groups, who have shown that HW/SW partitioning can be successfully used for FPGA soft-cores [13], encourage us to pursue our research in order to address the unresolved issues of our system: for example, while 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. We are also exploring the possibility of automatically generating HDL code for the extracted hardware blocks, a tool that would allow us to verify our approach on a larger set of problems and on real 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. *IEEE Micro*, 17(3):62–70, 1997.

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

[4] 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.

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

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

[7] 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.

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

[9] 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.

[10] J. Hoogerbrugge and H. Corporaal. Transport-triggering vs. operation-triggering. In *Proceedings 5th International Conference Compiler Construction, CC '94*, pages 435–449, January 1994.

[11] 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, Washington, DC, USA, 1996. IEEE Computer Society.

[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] 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, Washington, DC, USA, 2005. IEEE Computer Society.

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

[15] 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, Washington, DC, USA, 1998. IEEE Computer Society.

[16] 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.

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

[18] 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.

[19] 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.