Runtime decision of hardware or software execution on a heterogeneous reconfigurable platform

Vlad-Mihai Sima, Koen Bertels
Computer Engineering
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Abstract

In this paper, we present a runtime optimization targeting the speedup of applications running on a reconfigurable platform supporting the MOLEN programming paradigm. More specifically, for functions that have an execution time dependent on parameters, we propose an online adaptive decision algorithm to determine if the gain of running that function in hardware outweighs the overhead of transferring the parameters, managing the start and stop of the execution and obtaining the result. Our approach is dynamic in the sense it does not rely on compile time information.The algorithm is applied on a real video codec for which a function is implemented in hardware and we show improvements as big as 24% percent can be obtained for the specific kernel. We also determine the overhead and execution time ranges in which this optimisation is usefull and what other factors can influence it. [1]

1. Introduction

Due to the increasing heterogeneity of computer system and applications, the hardware and software designers develop new approaches that use more efficiently the available, limited, resources. A wide range of such problems can be solved in a fast and efficient manner by Reconfigurable Computing which combines the flexibility of a GPP (general purpose processor) with the speed of the (reconfigurable) hardware. One issue with such complex system is to decide the mapping between the tasks that have to be performed and the available hardware. Within a single application context, this can be solved by the compiler. However as soon as multiple applications compete for the same resources, the compiler cannot solve this.

In this paper, we propose an online decision algorithm called **AMAP** (adaptive mapping algorithm) that, taking into account particularities of the function and the history of the execution decides which implementation should use for the execution of that instance. One main novelty of the algorithm is that it takes this decision as late as possible - before each run - so it can make better decisions than a compile time algorithm. Also the profile information is stored and will be used when taking the decision for the next function call.

1. This research has been funded by the hArtes project EU-IST-035143, the Morpheus project EU-IST-027342 and the Rcosy Progress project DES-6392

The paper is organized as follows: in Section 2 we briefly present the Molen programming paradigm for reconfigurable architectures and related work. Next, we give a motivational example and also define the exact problem. A detailed description of the runtime algorithm is presented in Section 4. The results of the algorithm are shown in Section 5. In Section 6, we present conclusions and outline new research directions.

2. Background and related work

The **MOLEN programming paradigm** [1] is a paradigm that offers an abstraction of the available resources to the programmer, together with a model of interaction between the components. Using a 'one time' architectural or operating system extension the Molen programming paradigm allows for a virtually infinite number of new hardware operations to be executed on the reconfigurable hardware.

The work done in hardware software partitioning considered until now static partitioning done at compile time with the objective to minimize the total cost or minimize the cost while satisfying one constraint [2]. Various algorithms have been used to solve this problem like simulated annealing [3] [4], integer linear programming [5], mixed integer linear programming [6], knapsack problem [7] and genetic algorithms [8].

Different other problems were considered when doing the partitioning, like: area allocation [5], granularity selection [9] and scheduling [4] [8].

One common characteristic of all these approaches is that they rely on the fact that the profile information and execution trace are available at compile time and they optimize just for one specific set of cases ([5] [4] [9]).

From the runtime and operating system point of view, the work was focused on online scheduling for task that are already mapped to hardware [10]. Using a cache and software dispatch was proposed in [11] but the software dispatch was used when the contention was too high.

Online hardware software partitioning for image processing was procepoosed in [12] but there, as in other works, the algorithm used 'performance profiles' that had to be computed at compile time. A similar problem is described in [13], where the problem is defined in terms of an application set and various profile information known before the application starts. In [14] multiple applications are considered and an algorithm is given to select the most efficient set of functions taking into account function speed and area constraints. Once the selection is made, the decision is changed only when the application gets into a new execution stage.

Considering the previous work, we propose a new method of improving the hardware software partitioning by taking the final decision at runtime based on system execution history, and not on predetermined execution times. The decision is taken for each function execution and not for all executions. This is useful for coarse granularity (task/function level) and for functions which execution time is dependent on the parameters.

3. Problem definition

When optimizing an application for a reconfigurable platform one of the most important steps is the hardware/software partitioning. The role of this phase is to decide which functions/tasks will be implemented in hardware and which in software. The decision is taken based on multiple attributes depending on the partitioning algorithm, for example: estimated execution time, profile information, hardware area available and data dependency between tasks. One disadvantage of this approach is that, if the attribute depends on the function parameters, just the average for multiple execution is considered. In real life, for the same task some of the attributes (like execution time) can have multiple values based on the parameters of the tasks. Assuming there will be just one value implies some optimizations possibilities are lost.

Consider the hardware software partitioner decided that function f should be implemented in hardware. The total execution time in hardware is represented by the following formula:

$$t_{hw\_exec} = t_{setup} + t_{exec} + t_{return} \qquad (1)$$

The time t_{setup} represents the time needed for parameter transfer, memory transfer and the start of the hardware. This should include any overhead introduced for example by the operating system or the hardware control unit. The value t_{return} includes the time needed to retrieve the result, copy the data if necessary and stop the hardware. The t_{exec} time is the time in which the hardware unit processes the data and provides the results.

If the times in Equation 1 are independent of the parameters, the total execution time t_{hw_exec} can be computed at compile time and the decision of using the hardware or the software implementation is a compile time decision. On the other hand if any of the above times depend on the parameter values the decision that is taken at compile time could be suboptimal for some cases.

```
int satd(char *pix1, int i_pix1,
         char *pix2, int i_pix2,
         int i_width, int i_height)
{
  int x, y;
  int result;

  for( y = 0; y < i_height; y += 4 )
 {
    for( x = 0; x < i_width; x += 4 )
    {
      ... computations ...
    }
    pix1 += 4 * i_pix1;
    pix2 += 4 * i_pix2;

  }
  return result;
}
```

Figure 1. Motivational example from x264 application

As an example of such a case we give the function in Figure 1. It is obvious that the execution time depends on the parameters *i_width* and *i_height* as these two parameters control the number of iterations. The memory access pattern is determined by *i_pix1* and *i_pix2*. We measured the execution time both when running on the GPP and when running on the FPGA and we obtained the results in Table 1. We can see that for this function multiple execution times are possible depending on the parameters and even more the 'speedup' (ratio between the software and hardware execution time plus overhead) is not constant. A static, compile time, hardware software partitioner could use just the average of the execution times or the average speedup and in this way miss optimizations.

Problem statement: When both a software and a hardware implementation for a specific function are available, decide, taking into account the overheads, which of the

Table 1. Parameters and execution time for satd call (P1,P2,P3,P4 are the integer parameters of satd)

Parameters				Execution times μ s			Speedup
P1	P2	P3	P4	Overhead	t_{sw}	t_{hw}	
16	16	32	16	5	31	12.9	1.73
4	4	32	16	4.6	2.55	1.9	0.39
8	8	32	16	4.65	7.86	4.2	0.88

two instances is better to execute. The overheads are also affected by the particularities of the architecture like the time needed to transfer parameters, the time needed to start/stop the execution of the hardware function (t_{setup}) and the time needed to retrieve the result (t_{return}). The target is to make this decision at run time, as at compile time the parameters value will not be available.

One advantage of this approach is that the algorithm decision is taken based on the current state of the system. One example of an unexpected state of the system is the start of a different application. Another example is a DMA memory transfer that can cause a spike in the bus use so transferring memory could become a problem and using the software version that uses cached data can become more efficient. This is something it can't be known and it is very hard to predict at compile time.

4. Conditional hardware execution for Molen

As instrumentation and profiling in a real environment are difficult and error prone, we propose a solution that will react dynamically, when the application runs, to the changing conditions. We assume the designer of the application has already a set of candidates for hardware execution. For this set of candidates, exact profiling information would be needed in order to be able to take a decision at compile time. This is not always possible because of two reasons: the behavior of the candidate functions can be changed depending on the parameters and the running conditions might change because of event external to the application, like multiple application running or power constraints that affect the system. The main idea is to save the values of the parameters for each function call together with the execution time in software and in hardware, so next time a function is called with the same parameters an estimation can be done on whatever it is more efficient to run the hardware version or the software version.

The overall structure of the algorithm is depicted in Figure 2.

The algorithm will be able to react to changing conditions that can appear in a reconfigurable system. Also, it can directly take into account the reconfiguration overhead as it measures the needed time for executing a hardware function. In this way, even if it is not aware of a configuration caching mechanism it will detect at runtime that one of the configurations is cached (so it can be configured much faster) and use for the other hardware functions the software version (as reconfiguring would take too much time).

4.1. Runtime Profile Data Module

The main purpose of the runtime profile data module is to offer information about past invocations with the same parameters for a specific function - we will call this metrics.

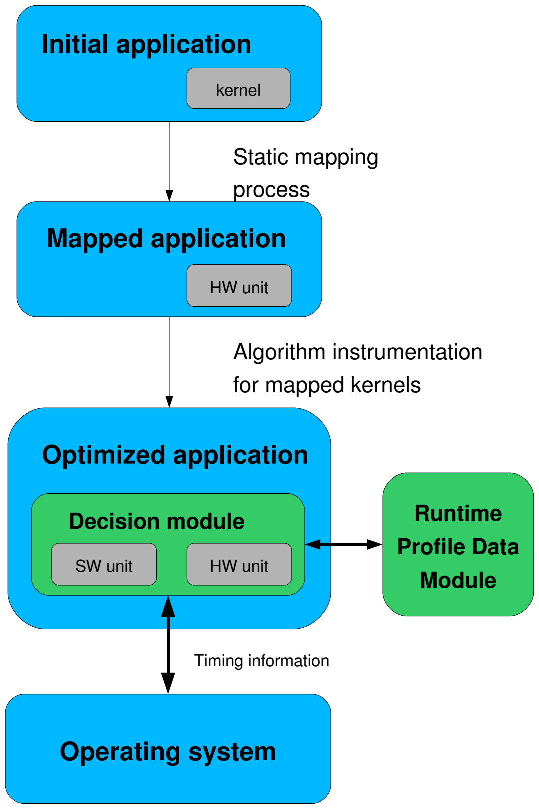

Figure 2. Algorithm overall structure

Our algorithm needs all the values of the parameters, so we need a data structure in which we can insert and search fast, but we are not interested in deleting elements. Also, as all processing happens at runtime we need a simple data structure to limit the overhead imposed by it. Taking this into account, we consider a red-black tree as the supporting data structure. The complexity of the search and insert operation are logarithmic so we consider the binary search tree a good choice.

For a one parameter function the tree is straightforward as each node will contain the values of that parameter and the associated metrics. In case the function has more parameters the binary search tree becomes a compositions of binary search trees, where just the nodes corresponding to the last parameter have associated metrics. Let's assume we have a two parameter function. Given the parameters in parameters in Table. 2 the resulting tree is depicted in Figure. 3. When the function is called we do the following steps to identify the node containing the information about past invocations:

```
t = tree root
p = first parameter
do {
```

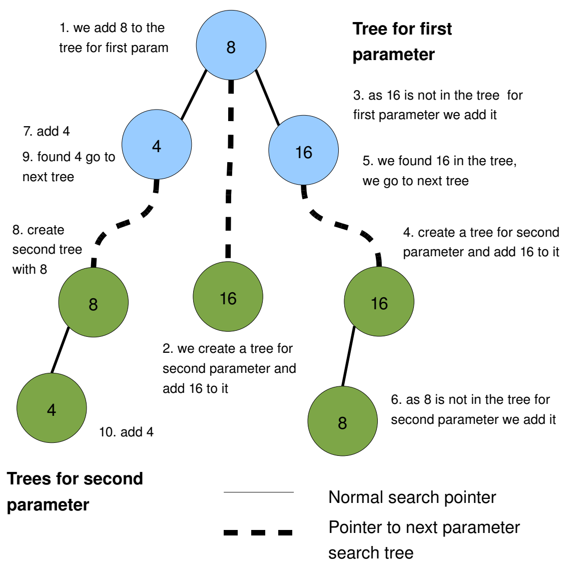
1. we add 8 to the tree for first param
8
Tree for first parameter
3. as 16 is not in the tree for first parameter we add it
7. add 4
9. found 4 go to next tree
4
16
5. we found 16 in the tree, we go to next tree
8. create second tree with 8
4. create a tree for second parameter and add 16 to it
8
16
16
2. we create a tree for second parameter and add 16 to it
4
8
6. as 8 is not in the tree for second parameter we add it
10. add 4
Trees for second parameter
Normal search pointer
Pointer to next parameter search tree

Figure 3. Search tree for 2 parameter function

```
  n = find value of p in t
  if(n is not found) {
    n = add p to t
  }
  t = next parameter tree from t
  p = next parameter
} while(p <= last parameter)
```

For each of the nodes corresponding to the second parameter, there is a metrics structure associated (omitted to simplify the figure).

The metrics we will use are:

- hardware execution time and the number of executions
- software execution time and the number of executions.

We chose this metrics as the focus of the algorithm is to improve the total execution time. Different metrics could be used in case another objective would have been chosen, for example effective area occupied in case the objective would be to increase the total utilization rate of the FPGA.

The decision function uses the metrics to decide if for this sequence of parameters it is better to use the software or the hardware function.

One important issue with the profile data module is that it could take too much space of the processor cache and that would degrade performance. The solution is to allocate the data in a contiguous block that is a multiple of the size of the cache line, and limit the increase to a certain value. When the limit is reached, the cache module will not be able to add new nodes to the tree. The solution we propose is to add the metrics to the ones of the last found node (the closest to the one that should be added). We will analyze in the results section the effect of this limitations over the algorithm efficiency.

Table 2. Parameter values for satd call

Parameters		Steps in figure
height	width	
8	16	1,2
16	16	3,4
16	8	5,6
4	8	7,8
4	4	9,10

```
m = get_profile(non pointer parameters);
if ( decision(m) is sw ) {
  t = time(call_sw_f());
  set_sw_time(m,t);
} else {
  t = time(call_hw_f());
  set_hw_time(m,t);
}
```

Figure 4. AMAP decision module

4.2. Decision module

The runtime part of the algorithm outline is presented in Figure. 4. For each function call that is taken into account by the algorithm, the code will be inserted in place of the call and a specific cache will be created at program start. The *call_hw* function must contain everything needed by the hardware call, like memory and parameter transfers. The functions *set_sw_time* and *set_hw_time* are the functions that update the profile data module metrics after the call. The function *get_profile* is used to retrieve the metrics structure from the profile data module, while the function *decision* represents the logic in the decision module.

The decision function does a comparison of the times needed in hardware and in software for a specific node in the profile data module and the the one which has a smaller execution time.

5. Results

In this section, we present the estimated results of applying the algorithm on the x264 video codec.

The x264 video codec is the state of the art in video compressing and it requires a lot of computing power, which involves it is challenging to use it in embedded environments. One of the most time consuming kernels of the application (around 30% of the total execution time) is a function - *satd_wxh* - which computes the sum of absolute differences. In Table 1 we give examples of parameters with which is called the *satd_wxh* function, togheter with the times needed for the execution, for each parameter set.

Our hardware platform is a Xilinx Virtex-4 ML410 which is based on the Xilinx XC4VFX60 FPGA. The MOLEN programming paradigm is implemented using the APU unit of the PowerPC and an on chip memory, which is accessed through the DCR bus. The design contains also a Flash memory reader used as external memory, and an internal 256 MB DDR2 memory. For the x264 application we implemented and tested *satd_wxh* kernel using the DWARV tool [15]. The system runs at 200 MHz and the hardware designs are clocked at 100 MHz. We couldn't execute the whole application on the board as the PowerPC is not able to access the internal FPGA memory through it's data cache. Even if we are able to place all the needed data in the internal FPGA memory, work is needed to allow the data cache to work with that memory.

All the data was collected from actually running the applications with real input. As the memory transfer are known to be suboptimal we measured the operating system overhead, the hardware and software execution and estimated the result based on these numbers.

By applying the algorithm, we can select the best case for any parameter set. We applied the algorithm on various reference videos such that the trace of the execution changes based on the data - so our function will receive different parameters for different videos. The results are listed in table 3. The *HW* column represent the total execution time of the kernel in case all the calls would be executed on the reconfigurable fabric compared to the software execution time. The result obtained by applying our algorithm are presented in the column *AMAP*. The last column represent the 'overhead' introduced by the algorithm, respectively the percent of the total execution time spent in the decision and cache module.

We can see from the table that always using the hardware can degrade performance. This can happen if the overhead, represented by the system call and starting/stopping the hardware unit - is comparable (or greater) to the total time spent inside the kernel - for example for hall, claire and miss-america. By applying our algorithm the improvement ranges from 43% to 5% with an average of 15%.

As mentioned in Section 4.1 one important aspect is the memory footprint of the data stored about parameters and execution times. To test how the algorithm behaves in such cases we set various limits and run simulations with the gathered data. The results are presented in Figure 5. The first thing that can be observed in the graph is that the size of the memory footprint will affect the behavior of the algorithm. The second important observation is that for a size of 10 there is a cut off point - after which the improvement is marginal. The big differences in execution time per frame between the data sets can be explained by the fact the algorithm results depend on the amount of 'motion' that happens in the video.

To test the behavior of our algorithm in an extreme situation we considered increasingly high overheads (multiplying by 2, 3 etc the measured overhead). The results can be seen in Figure 6. We compare with the execution 'completely in software'. The bold line represents the average for the data. As expected, the algorithm will use more and more the software version. After some point, the overhead makes running the hardware completely inefficient (in our graph, around 5x overhead), and the algorithm will mostly use the software. The decrease in performance (after 5x, everything that is the above 100% line is a decrease in performance) is because the algorithm has to profile at least some executions in order to determine that the overhead is significant and it would be inneficient to execute the function in hardware. Still, it will gracefully adapt to the new conditions and limit the performance decrease to less than 3% compared to pure software execution for an overhead increase of 8x.

Table 3. Total kernel execution time in hardware compared to total kernel execution tim in software

Video	HW	AMAP	AMAP execution
akiyo	82.01%	76.08%	4.05%
carphone	106.45%	87.22%	6.50%
claire	134.97%	91.87%	7.25%
coastguard	91.25%	86.71%	4.98%
container	90.18%	79.96%	4.86%
foreman	98.69%	85.36%	5.68%
hall	113.64%	88.60%	7.14%
miss-america	121.65%	91.47%	8.10%
mobile	88.95%	82.25%	4.71%
news	88.85%	80.29%	4.77%
salesman	93.54%	80.75%	4.74%
silent	93.50%	84.37%	5.16%
suzie	104.42%	86.67%	6.37%
Average	100.62%	84.74%	5.72%

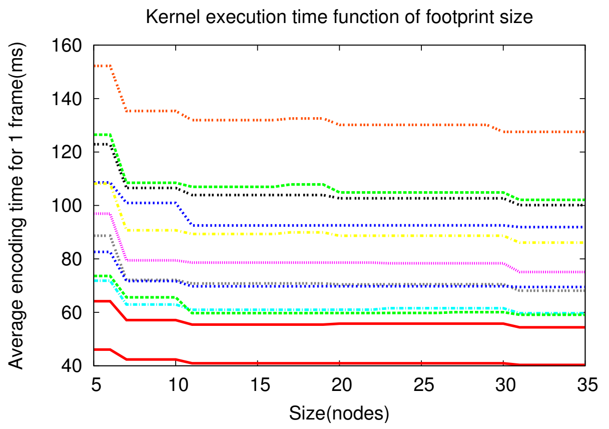

Figure 5. Algorithm performance for different cache sizes (each line represents a different video)

6. Conclusions

In this paper, we proposed a runtime algorithm which, using profiling, selects between the software and hardware execution for a specific function based on the parameter values with which the function is called. Our experiments

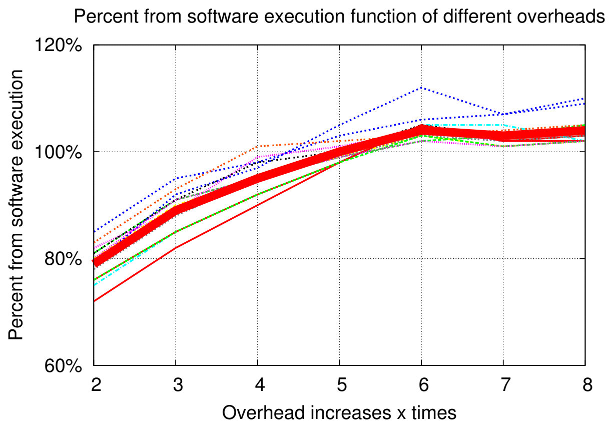

Figure 6. Algorithm performance for different overheads (each line represents a different video, bold line represents average)

show that it can give a significant improvement in a dynamic system, where it is hard at compile time to predict all the parameters of the system. The algorithms can be seen as an extension to traditional compile time hardware software mapping, which ignores the dynamic behavior of the tasks. We also studied the impact of the size of the profile data stored in the tree and examined the effect of various overheads on the performance.

As future work we can analyze policies for purging the cache and study methods to better capture the profile time in a dynamic environment. Also, the algorithm could be included as a hardware unit to minimize the overhead imposed by it's execution.

References

[1] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The molen polymorphic processor," *IEEE Trans. Comput.*, vol. 53, no. 11, pp. 1363–1375, 2004.

[2] P. Arató, Z. Ádám Mann, and A. Orbán, "Algorithmic aspects of hardware/software partitioning," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 10, no. 1, pp. 136–156, 2005.

[3] S. Banerjee and N. Dutt, "Efficient search space exploration for hw-sw partitioning," in *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM, 2004, pp. 122–127.

[4] B. Miramond and J.-M. Delosme, "Design space exploration for dynamically reconfigurable architectures," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 366–371.

[5] E. M. Panainte, K. Bertels, and S. Vassiliadis, "Compiler-driven fpga-area allocation for reconfigurable computing," in *DATE '06: Proceedings of the conference on Design, automation and test in Europe*. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2006, pp. 369–374.

[6] A. C. Nagaraj Shenoy and P. Banerjee, "An algorithm for synthesis of large time-constrained heterogeneous adaptive systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 6, no. 2, pp. 207–225, 2001.

[7] W. Jigang and T. Srikanthan, "Algorithmic aspects of area-efficient hardware/software partitioning," *J. Supercomput.*, vol. 38, no. 3, pp. 223–235, 2006.

[8] J. Harkin, T. M. McGinnity, and L. P. Maguire, "Modeling and optimizing run-time reconfiguration using evolutionary computation," *Trans. on Embedded Computing Sys.*, vol. 3, no. 4, pp. 661–685, 2004.

[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 Trans. Very Large Scale Integr. Syst.*, vol. 9, no. 2, pp. 273–290, 2001.

[10] H. Walder and M. Platzner, "Online scheduling for block-partitioned reconfigurable devices," in *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2003, p. 10290.

[11] M. Dales, "Managing a reconfigurable processor in a general purpose workstation environment," in *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2003, p. 10980.

[12] H. Quinn, L. A. S. King, M. Leeser, and W. Meleis, "Runtime assignment of reconfigurable hardware components for image processing pipelines," in *FCCM '03: Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2003, p. 173.

[13] C. Huang and F. Vahid, "Dynamic coprocessor management for fpga-enhanced compute platforms," in *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*. New York, NY, USA: ACM, 2008, pp. 71–78.

[14] W. Fu and K. Compton, "An execution environment for reconfigurable computing," in *FCCM '05: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 149–158.

[15] Y. D. Yankova, G. Kuzmanov, K. Bertels, G. N. Gaydadjiev, J. Lu, and S. Vassiliadis, "Dwarv: Delftworkbench automated reconfigurable vhdl generator," in *In Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL07)*, August 2007, pp. 697–701.