Developing Applications for Polymorphic Processors : The Delft Workbench

Koen Bertels, *Member, IEEE,* Stamatis Vassiliadis, *Fellow, IEEE,* Elena Moscu Panainte, Yana Yankova, Carlo Galuzzi, Ricardo Chaves, Georgi Kuzmanov, *Member, IEEE,*

***Abstract*— Reconfigurable computing bears a great promise as it provides the flexibility of software design as well as the substantially better performance of hardware based execution. In order for this technology to really catch on, the necessary tools have to be developed that provide an integrated and (semi) automated development platform. Such a development platform should bridge as much as possible the differences that exist between software and hardware development for which distinct techniques and approaches are required. This article presents the current state of the Delft workbench that allows the developer of any kind of application to develop new or port existing applications to a reconfigurable platform.**

***Index Terms*— Reconfigurable hardware, hardware software interface, compiler**

I. Introduction

Reconfigurable computing has gained increasing attention from the industry over the last couple of years as it constitutes a very interesting marriage between the performance of hardware and the flexibility of software. Reconfigurable fabrics such as FPGA's can be used as stand-alone processors or in combination with general purpose processors (GPP). In this article, we focus on the latter use where the reconfigurable device will contain application specific logic which previously had to be implemented in either dedicated hardware or only in software. The functions executed on the reconfigurable fabric can be changed (at runtime or at compile time) in function of the application at hand. However, for this technology to really be adopted on a large scale, a number of important gaps have to bridged of which some are considered to be difficult. One of those challenges is the need for a machine organization that provides a generic way in which different components such as a general purpose processor and various reconfigurable devices can be combined in a transparent way.[1] Another challenge is that we need the necessary tools to transform (existing or new) applications in such a way that we can ultimately use the reconfigurable computing units. We need such tools because application development in this context is no longer a pure software writing effort but assumes substantial hardware design capabilities. The resulting or envisioned hardware is application specific and the term used to denote it is Application Specific Instruction Set processors (ASIP). Where traditional general purpose processors have a fixed instruction set, ASIP's have a flexible, extendible and application specific instruction set which can even change at run time. So for application developers to use this technology, (semi-) automatic support in designing the hardware and to generate the required executable code should be available. In this article, we present the current state of the Delft workbench project that targets the required toolset and which is based on the Molen Programming Paradigm. In short, the design flow, shown in Figure 1, consists of the following steps. Starting point is an application written in, for instance, C. Using a decision model, the application will be profiled to identify those parts that could be implemented on the reconfigurable fabric. Using a C2C-compiler, the original application then needs to be transformed as the accelerated parts will have to be replaced with the appropriate calls to the reconfigurable fabric. Once those transformations have been applied, the extracted functions have to be implemented in hardware. To this purpose, they are translated into VHDL. This can be done either automatically, manually or using an available IP-core from a library. Once the configuration and execution code have been implemented, the retargetable compiler can then generate the appropriate binaries after which the modified application can be executed on the heterogeneous platform. A feedback loop ensures that certain choices can be evaluated and modified after which the same design steps can be repeated. The remainder of the paper will discuss each of those steps in more detail and is structured as follows. We first introduce the Molen programming paradigm which assumes a particular machine organisation that combines a general purpose processor (GPP) with one or more custom computing units (CCU). We then introduce the different steps of the application design process starting at the profiling phase and then moving to transforming the application and defining the new instructions. Compilation and hardware generation are the final steps after which the applications can be executed. For each design step, we present a small motivational example. We conclude the paper by presenting a case study that concerns the implementation of the AES-encryption algorithm and presents the programmer's interface to use such heterogeneous platforms.

[1]We will use the terms reconfigurable devices, custom computing units and FPGA interchangeably in this article. The terms kernel and code segments are also used as synonyms. We define them as any combination of instructions or a a combination of clusters of instructions that will be considered a candidate for hardware acceleration.

II. The Molen Programming Paradigm

The Molen programming paradigm is a sequential consistency paradigm in which an application can be partitioned in such a way that certain parts can run (in parallel) on the reconfigurable fabric and other parts run on the general purpose processor. This paradigm assumes the Molen

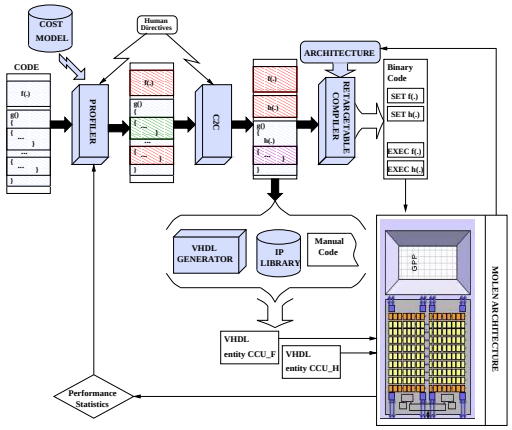

Fig. 1: The Delft Workbench Design Flow

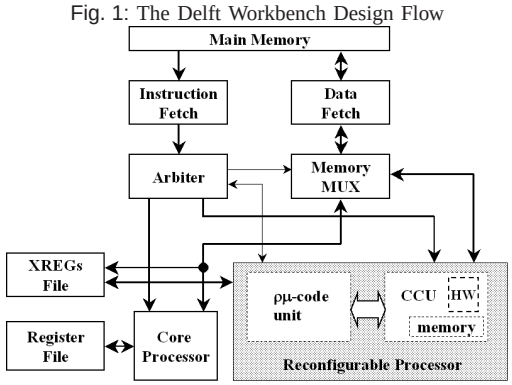

Fig. 2: The Molen machine organization

organisation which defines the interaction between a general purpose processor and the custom computing units (CCU), implemented on the FPGA [1]. It consists of a one time instruction set extension that allows the implementation of an arbitrary number of CCU's. The paradigm allows for parallel and concurrent hardware execution and is intended (currently) for single program execution. For a given ISA, a one time architectural extension (based on the co-processor architectural paradigm) comprising 4 instructions (for the minimal poloymorphic (π)ISA) suffices to provide an almost arbitrary number of operations that can be performed on the reconfigurable hardware. The four basic instructions needed are **set**, **execute**, **movtx** and **movfx**. By implementing the first two instructions (**set**/**execute**) an hardware implementation can be loaded and executed in the reconfigurable processor. The **movtx** and **movfx** instructions are needed to provide the communications between the reconfigurable hardware and the general-purpose processor (GPP). The Molen machine organization that supports the Molen programming paradigm is described in Figure 2. The two main components in the Molen machine organization are the 'Core Processor', which is a GPP and the 'Reconfigurable Processor' (RP). Instructions are issued to either processors by the 'Arbiter' by means of a partial decoding of the instructions received from the instruction fetch unit. The support for the SET/EXECUTE instructions required in the Molen programming paradigm is based on *reconfigurable microcode*. The reconfigurable microcode is used to emulate both the configuration of the CCU and the execution of implementations configured on the CCU.

III. Components of the Delft Workbench

A. Profiling

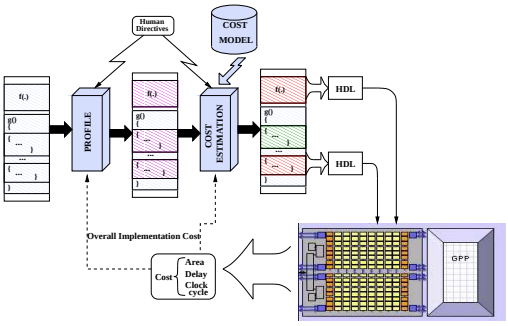

Fig. 3: Profiling the application

As shown in Figure 3, profiling consists of identifying those parts of an application that could be mapped on the reconfigurable hardware. The goal is to determine as early as possible in the design stage whether certain parts not only provide the necessary speedup but can also be implemented given the limited reconfigurable or other hardware resources. Making this analysis is called partitioning. The goal is thus to end up with certain parts that will be accelerated as a CCU and the remaining parts of the application will be executed on a regular general purpose processor. A 'part' of the application can be a whole function or procedure but it can also be any cluster of instructions that is scattered throughout the application. The partitioning itself will be determined in view of a particular objective such as increased performance, reduced power consumption or a smaller footprint.[2] Even though such a primary objective is the ultimate decision criterion, more aspects come into play. The code that will be accelerated must not only fit on the available area of the reconfigurable chip, the cycle time should not be affected too much. This cycle time can go down as the code segment executed on the reconfigurable device grows in size, resulting in an overall slower execution. So one of the tradeoffs the designer has to make is to choose between larger accelerated code segments and higher performance improvement but then also more area and longer cycle time. These different trade offs that have to be taken into account, combined with the large number of possible code segments, make the design space very large. This is why researchers have proposed various kinds of automatic support.

Where some approaches do compile time profiling, most estimation schemes in literature have focused on estimation based on synthesis-like techniques, simulation, or simple summation approaches. If it is known what hardware operators will (probably) be used, the profiler can use various equations to predict the area consumed. For instance, in [2], quadratic relations are used for multipliers and linear ones for addition and subtraction. However, this information is missing at the beginning of the design process where we are merely looking at the application and trying to figure out what

[2]For the remainder of the paper, we will refer exclusively to performance improvement but the approach holds mutatis mutandis for other objective functions.

the computational hotspots are. For instance, when looking at MPEG2, a profiler could for instance identify, through trace analysis, that the functions Sum of Absolute Differences (SAD), 2 dimensional Discrete Cosine Transformation (DCT) and its inverse (IDCT) consume together around 65% of the total MPEG2 execution time. Implementing them on hardware could result in a significant speedup (see e.g. [3] for details) as can be computed using

$$n_{molen} \simeq n_{soft} - n_f + n_{call} * cost \quad (1)$$

with $cost = x_{set} + y_{exec}$ where

- n_{molen} represents the total number of GPP cycles spend in the application;
- n_{soft} represents the total number of cycles when completed executed by the GPP;
- n_f represents the total number of cycles spent in a function f;
- n_{call} represents the number of calls to function f in the application;
- x_{set} represents the number of GPP cycles required for one configuration of the FPGA for the function f;
- y_{exec} represents the number of GPP cycles required for one execution on the FPGA for function f;

Performance improvement means that the total number of GPP cycles needed by the Molen processor to execute the application - n_{molen} - is less than the number of cycles for a pure GPP-based execution - n_{soft}. Using equation 1, $n_{molen} < n_{soft}$ when the cost of using the reconfigurable function expressed by the number of GPP cycles required to perform the SET and EXECUTE instructions, is less than n_f/n_{call}. Where this simple model allows us to determine whether a function could be beneficially implemented as a CCU, it has two main disadvantages: first, it assumes that complete functions will be mapped on the hardware. It may however be useful that only a part of a function is mapped or that a part of a function is combined with a part of another function and then mapped on hardware. The second drawback is that it does not make any kind of prediction of the hardware resources that will be required. Where the first problem could be (partially) addressed by code rewriting, the second problem needs to be resolved in a different way.[3] What is required is a means to relate the software function to the hardware. One possible approach which is similar to [2], is to build a quantitative model on the basis of a large data set of various kernels, belonging to different application domains, having different sizes and for which a VHDL-implementation is available. Each kernel could then be characterized using several software metrics such as McCabe's cyclomatic number and Halsteads length, volume, and effort. Using the VHDL-implementation, the kernels can be synthesized for a specific reconfigurable platform, and kernel wise executed to collect run-time information. From this synthesis and execution, we could collect information such as area, latency and speedup. Using standard regression techniques, the relationship between the software metrics and hardware characteristics could be estimated and consequently used for prediction purposes. However, several attempts for constructing such models have been tried but none so far provide sufficiently accurate predictions.

[3]This part, as well as the loop optimizations to be discussed in the next section, is still under development and only the basic ideas are described here.

B. Graph Transformations

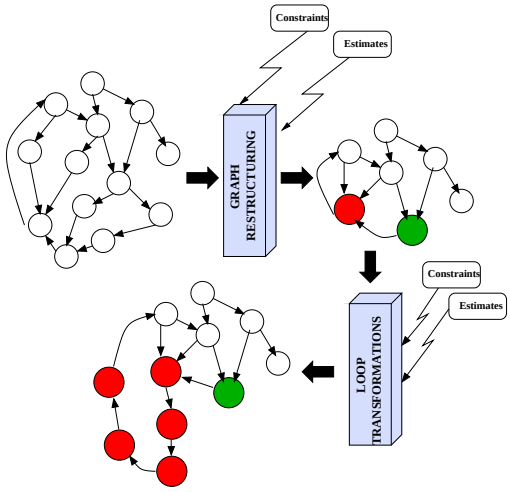

Fig. 4: Transforming the original application

Even though the profiler made a first assessment of what kernels could be mapped on reconfigurable hardware, whether in the end this will be done depends on a number of subsequent analyzes and transformations. An important next step, depicted in Figure 4, is to determine how the proposed code segments can be clustered. These clusters, if selected, will become the new instruction in the instruction set. Speedup can be obtained by exploiting available parallelism. This aspect therefore needs to be carefully analyzed. An important source of parallelism are loops. As loops are an indispensable control structure in any application and as some of the clusters may either contain loops or be part of a loop body, we need to see how the loops can be transformed in order to achieve the envisioned performance improvement.

Graph Restructuring The process of identifying the clusters involves finding certain patterns in the data flow graph (DFG).[4] The starting point in this clustering process are what are called MISO's, clusters having **M**ultiple **I**nputs and a **S**ingle **O**utput. Even though the limitation to a single output is restrictive, research has mainly focused on this kind because they have the interesting property of convexity.[5] This will guarantee a feasible and proper scheduling by the compiler. However, as will be explained below, we can overcome this limitation by combining multiple MISOs into Multiple Inputs Multiple Outputs (MIMO).

[4]A data flow graph is a graph in which each node represents an operation. Going from one node to another means going from one operation to the next. When control flow information is also available, we speak of control data flow graphs(CDFG).

[5]A subgraph G^* of a graph G is convex if there does not exist a path from any of the nodes of the subgraph to the output node of G^* that does not belong to the subgraph G^*.

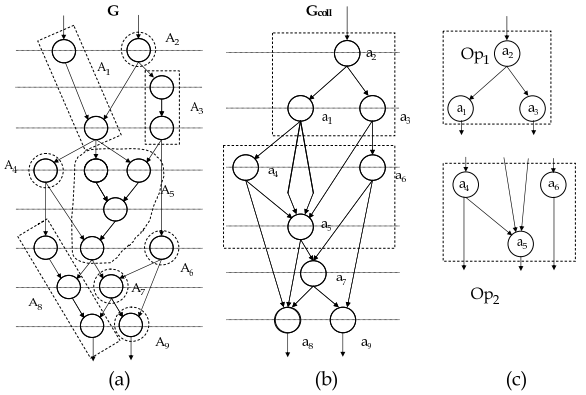

Fig. 5: Convex MIMO instructions generation: (a) the original graph G is partitioned in maximal MISO clusters; (b) the clusters are collapsed in single nodes which have the same behavior ($A_i \mapsto a_i$); (c) the collapsed nodes are combined per levels to obtain convex MIMO instructions and can be connected as Op_1 or disconnected as Op_2. The horizontal lines in (a) and (b) represent the level of the nodes.

As depicted in Figure 5a-b-c, growing the cluster is done in several stages. In step (a), MISO-clusters are combined with each other resulting in maximally sized MISO sub-graphs, called MaxMISO's. The motivation for generating MaxMISO's is that we do not want to have many small clusters scattered all over the program which will create too much overhead and only marginal, if any, performance improvement. In step (b), these MaxMISO's are then collapsed in single nodes that represent the potentially new instructions. Finally, in step (c), we finally will select two operations, Op_1 and Op_2 using some decision criteria, that will be mapped onto hardware as newly available instructions.

Loop Transformations Once the kernels have been clustered and collapsed on a single node, we can now turn to optimizations that allow to fully exploit the newly identified instructions. An important source of performance improvement are loops for which various optimisations exist, such as loop tiling, loop fusion, loop unrolling and software pipelining. As the impact of the different optimisations is still an open research question and beyond the scope of this article, we will illustrate some of the issues using an example.

The following loop contains a call to a reconfigurable function. The question we want to answer is whether the function that will run on the reconfigurable fabric needs to be modified as to include the whole loop or whether we apply loop unrolling in a partial or complete way.

```
for I = 1 to n
    for J = 1 to n
        a[I,J] = call_FPGA_OP1(A[I-1,J], A[I-1,J+1])
    endfor
endfor
```

When it has been chosen to apply full loop unrolling for the inner loop, the code becomes

```
for I = 1 to n
    a[I,1] = call_FPGA_OP1(A[I-1,1],A[I-1,2])
    a[I,2] = call_FPGA_OP1(A[I-1,2],A[I-1,3])

    a[I,n] = call_FPGA_OP1(A[I-1,n],A[I-1,n])
endfor
```

Where determining the unrolling factor may be relatively simple if the loop is executed on a GPP, it may not be so evident in the context of hardware accelerators. Every unrolled instruction will occupy additional area and can therefore not be stretched unlimitedly. The unrolling may also increase the cycle time as the hardware function will take more time to execute as it grows in size. One way to counter this is to introduce pipelining. Therefore, the number of stages in the pipeline depends on the unrolling factor. There may also be issues regarding the required vs available bandwidth if more memory accesses are needed to feed the accelerator.

C. Retargetable Compilation

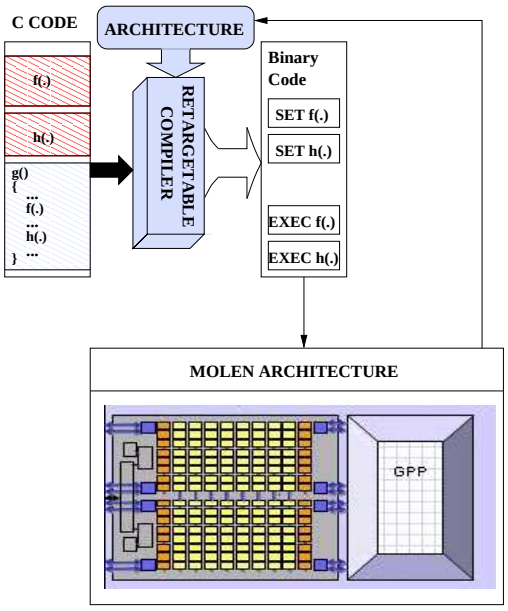

Fig. 6: Compiling the modified application

As can be seen from Figure 1, the design flow now forks into a compiler part and a VHDL generation part. As far as the compiler is concerned, and represented by Figure 6, the target architecture has now been augmented with the newly identified instructions and the compiler needs to be able to exploit these new features. As explained before, the Molen programming paradigm requires for each new hardware function a SET and EXECUTE instruction that will respectively configure and execute the FPGA based CCU. It is the compiler's task to determine when to schedule the SET and EXECUTE instructions and where to place them on the FPGA, denoted as spatio-temporal compilation.

The 'temporal' refers to the point in the execution flow at which the configuration of the CCU is started. Given the huge latency, the compiler should be able to hide this configuration time such that the CCU is correctly configured when its EXECUTE statement is read. The 'spatial' aspect refers to the available area on the FPGA. Each CCU occupies a certain amount of the area and again it is the compiler's task to determine how many CCU's will fit at any given time on the FPGA. To address these spatio-temporal constraints, we introduce advanced instruction scheduling and area allocation algorithms. These algorithms minimize the number of

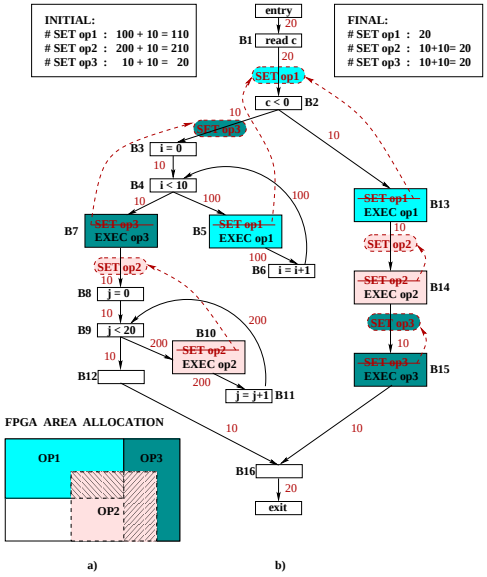

Fig. 7: Intraprocedural Instruction Scheduling

executed hardware reconfiguration instructions based on the application's specific features while taking into account the available area. Additionally, the compiler is capable to decide not to schedule the function as a CCU but rather to schedule the pure software based execution when the reconfiguration latency could not be reduced or when the required area was not available. The proposed scheduling algorithms are applied at both the intraprocedural and interprocedural level.

For the intraprocedural instruction scheduling algorithm, we present the following motivational example. Figure 7(b) shows the control-flow graph of a procedure, when *op1*, *op2* and *op3* operations are performed on the reconfigurable hardware and they are placed on the FPGA as presented in Figure 7(a). The numbers associated with each edge of the graph represent the execution frequency of the edge. One first observation is the redundant repetitive execution of *SET op1* at B5 in the loop B4-B5-B6. Operation *op1* is configured 100 times in B5 and 10 times in B13. By moving *SET op1* instruction on the (B1, B2)-edge, we can reduce the number of configuration calls for *op1* to 20 and make redundant the *SET op1* instruction at node B13. The hardware configuration for *op2* at B10 cannot be moved earlier than node B7 as it will change the hardware configuration for *op3* that must be performed at B7. There are no redundant configurations for *op3*, thus the hardware execution of *op3* has to be preceded each time by the hardware configuration. If the compiler detects that the hardware configuration for *op3* consumes all the performance gain produced by the hardware execution of *op3*, the scheduler can switch to its software execution on the GPP.

D. VHDL Generation

Where the first fork in Figure 1 concerned the compiler scheduling of the SET and EXECUTE instruction, the second fork involves the hardware generation for the kernels that have been transformed into new instructions. If these hardware descriptions are not available as IP-cores, we have to generate them either automatically or manually. To illustrate the idea of the translation of a computation, expressed in a high-level language (HLL), to a hardware model, consider the example shown in Figure 8. The presented C code (Figure 8a) is the $fmult$ function from the G721 encoder as implemented in the MediaBench benchmark suite [4]. The first step in the hardware generation process is to transform the input source into an internal representation that reflects the control and data dependencies in the algorithm. The most widely used representation is a hierarchical control- and data-flow graph (HCDFG) (Figure 8b, Figure 8d). The hierarchy levels in the HCDFG reflect the enclosure level of the statements. The nodes of this graph are basic blocks or compound nodes, corresponding to statements or called functions. The edges of the graph represent the control and data dependencies.[6] The basic blocks nodes refer to data-flow graphs (DFGs) that describe the data-dependecies within the corresponding basic block (Figure 8c). From the HCDFG the VHDL-compiler has to infer a hardware model for the computation. The arithmetic and logic operations within the basic blocks are mapped to pure logic (Figure 8c- e). When a control structure such as the `for-loop` in `quan` is found, a controller has to be generated (Figure 8f). In the automated hardware generation, controllers are usually described as finite-state machines.

After the computation model is derived, the actual generation of the hardware description is performed. This description is expressed in a hardware description language. There are several HDLs, but only two of them are widely supported by the EDA-vendors both for simulation and synthesis. These languages are VHDL and Verilog, and they are standardized by IEEE (IEEE Std 1076-2002 [5], IEEE Std 1076.6-2004 [6] and IEEE Std 1364-2005 [7], respectively). These languages are relatively new as the first VHDL standard was issued in 1987 and the first Verilog standard in 1995.

The standardization and the efforts of EDA-vendors providing quality synthesis tools have contributed to the growing acceptance of these languages as means for hardware design. Over the last couple of years, a lot of research effort has been invested in the automated HDL generation. Several commercial tools that generate hardware from HLL input also appeared (eg. Catapult-C [8], XPRES Complier [9]). Even though they provide some kind of automation, the whole process is strongly directed by the designer. Direct input is required for applying certain low-level optimizations as well as for the actual mapping process. In other words, the goal of these tools is to save code-writing time and to use them one still needs hardware-design knowledge.

The straightforward mapping of the different HLL constructs should not be considered an open issue (as illustrated in Figure 8). However, the quality of the generated hardware description is the main hurdle which is mainly due to the gap that exists between the essentially sequential code and the hardware computation model which is basically concurrent and data flow oriented. As an example, refer to the `for`-loop in Figure 8a. There are no inter-iteration data dependencies,

[6] In the figure, the basic blocks are presented with solid boxes and the compound nodes - with dashed boxes. The data edges are presented with dashed arrows and the control edges - with solid arrows.

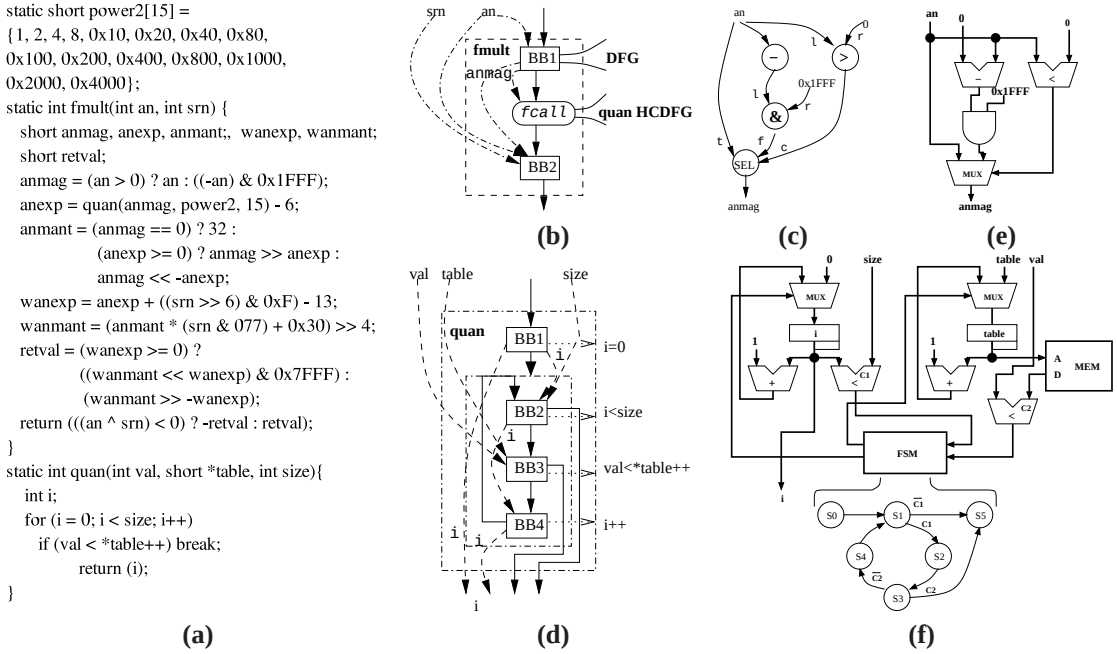

Fig. 8: C to HDL translation process: **(a)** `fmult` C Code; **(b)** `fmult` HCDFG; **(c)** `fmult BB1` DFG; **(d)** `quan` HCDFG; **(e)** `fmult BB1` schematics; **(e)** `quan` schematics

hence the loop can be fully unrolled (within the particular calling context, the upper bound is 15) and all comparisons can be implemented in parallel. Furthermore, as the `power2` table has only 15 constant values, the element references can be replaced with the corresponding values and the memory accesses can be completely removed. These simple optimizations would produce much faster design at the expense of a small increase of the area.

IV. Validation

To validate the MOLEN computational paradigm and to assess the current available toolset, algorithms taken from various domains have been implemented for which significant speedups have been reported. In the following, we will not elaborate on all the stages of the design process but rather emphasize the programmer's interface. We will show how easy it is for application developers to use the Molen polymorphic processor. As an example, assume that a developer needs to embed encryption in an existing application but because of real time constraints, a pure software based solution is not an option and the decision has been made to use an FPGA based engine of the symmetrical encryption algorithm AES [10]. The developer wants answers to the following questions: (i) how can an FPGA-based accelerator be integrated in existing applications? (ii) how is the modified application being executed on the Molen platform, and (iii) what are the speedups obtained? The reported results were obtained using a XILINX Virtex II Pro, which embeds a PowerPC as a GPP running at 300 MHz, with a main data memory running at 100 MHz.

Symmetric encryption algorithm - AES

The AES [11] is the new NIST standard chosen to replace DES, it uses the Rijndael encryption algorithm with cryptography keys of 128, 192, 256 bits, the 128 bit key being the most commonly used. It is beyond the scope of this article to present in detail the AES algorithm but the algorithm manipulates data input block, disposed in a 4 by 4 bytes matrix. The encryption and decryption computation consists of byte substitution, bit permutation and arithmetic operations in finite fields, more specifically, addition and multiplications in the Galois Field 2^8. Each set of operations is repeated 10, 12 or 14 times depending on the size of the key (128, 192, 256 bits respectively). This critical part of the ciphering process was implemented as a CCU whereas certain operations, such as key expansion, are executed on the GPP.

Programmer's Interface In order to use this CCU, the existing software code is compiled by the MOLEN compiler as described previously. For the compiler to be able to make the appropriate scheduling and area allocation, a simple, pragma-based annotation of the original source code is required as well as some information on the area occupation, configuration latency and the number of parameters.[7] Where the original function call to the AES-method was given by *void AES(Key, Data, Mode)*, the pragma annotation required for the Molen compiler is

```
Function declaration:
 #pragma call_fpga AESCipher
 void AES(Key, Data, Mode)
```

and the function usage remains unchanged, namely *AES(Key, Data, Mode)*.

Thus, for the application developer to use any reconfigurable coprocessor is transparent: it is called as if the function was implemented in software. Only the $pragma$ notation has to be added in order to inform the compiler which functions are implemented in hardware. This capability allows reconfigurable

[7] This information is contained in an architecture description file.

TABLE I: MOLEN with AES

	Hardware		Software		
Bits	Cycles	(Mbps) ThrPut	Cycles	(Mbps) ThrPut	Kernel SpeedUp
128	646	59	24216	1.59	43
4k	4366	281	738952	1.66	169
128k	31246	1258	23610504	1.67	751

coprocessors to be used in any existing application with minor modifications to the source code.

Polymorphic Program Execution In order to understand how the application is executed on the Molen platform (see Figure 2), we briefly present its execution flow. Like in any regular application, the instructions come from memory but rather than being decoded by the GPP, they are first (partially) decoded by the arbiter. The arbiter decides whether it concerns an instruction for the GPP or for any of the available CCU's, in our case the AES-CCU. The exchange registers are used to transfer parameters to and from the CCU. The data to be (de-)ciphered by the AES-CCU, is retrieved directly from the main data memory. If a SET instruction is found, the corresponding configuration code is executed. In case an EXECUTE-instruction is seen, the arbiter signals the CCU with the $Start$ flag. By receiving this signal, the control unit in the CCU for the AES-core starts initializing the core, by reading the memory pointers from the XREG which are used to address the main data memory. This initialization process includes the reading of an initialization vector and retrieving the expanded key from main data memory. Afterwards, the control enters a loop where each of the 128 bit data blocks is read, encrypted (or decrypted) and stored back into memory. Note that the AES-CCU only has to read the expanded key only when it is altered. The implementation of this system requires 1130 slices and 12 BRAMs. The maximum frequency of 100 MHz is imposed by the main data memory and not by the MOLEN processor.

Performance Improvement Table I presents the throughput and speedup results for a 128-bit key, using the pure software implementation and the Molen processor with the AES CCU. The speedups obtained for the AES-core are measured at kernel and not application level and limited unrolling is used. The impact of the initialization overhead is evident when comparing the speedups obtained for small and large data blocks, including the transfer of the (1408-bit) expanded key which needs to be done only once. A speedup of 43 is achieved when only one 128-bit data block is encrypted against a speedup of 751 for a file with 128 kbits.[8]

V. Conclusion and Future Research

In this article, we have sketched the different steps involved in developing applications that will run on a heterogeneous platform such as the Molen polymorphic processor. Specific about designing applications for such platforms is the blend of hardware and software development. As both require different techniques and skills, tools are required to provide sufficient support to fill in the gaps as much as possible. We have pointed out the different steps required to develop a new or modify an existing application that can be executed on such platforms. In the example given, we have emphasized the simplicity of using the Molen processor for existing applications if CCU's are available as IP-cores. The presented example also clearly illustrates the power of the approach yielding kernel speedups of up to 750 times. Whether reconfigurable computing will be largely adopted by the industry is highly dependent on the availability of workbenches as the one described here.

[8] Evaluation prototypes are available at http://ce.et.tudelft.nl/MOLEN

References

[1] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The Molen polymorphic processor," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363–1375, November 2004.

[2] R. R. D. Kulkarni, W. Najjar and F. Kurdahi, "Fast area estimation to support compiler optimizations in fpga-based reconfigurable systems," in *IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, April 2002.

[3] E. M. Panainte, K. Bertels, and S. Vassiliadis, "Multimedia reconfigurable hardware design space exploration," in *Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2004)*, November 2004, pp. 398–403.

[4] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems," in *International Symposium on Microarchitecture*, 1997, pp. 330–335. [Online]. Available: citeseer.ist.psu.edu/lee97mediabench.html

[5] (2002) 1076 ieee standard vhdl language reference manual, ieee std 1076-2002. [Online]. Available: http://ieeexplore.ieee.org/servlet/opac?punumber=7863

[6] (2004) Ieee standard for vhdl register transfer level (rtl) synthesis, ieee std 1076.6-2004. [Online]. Available: http://ieeexplore.ieee.org/servlet/opac?punumber=9308

[7] (2006) Ieee standard for verilog hardware description language, ieee std 1364 -2005. [Online]. Available: http://ieeexplore.ieee.org/servlet/opac?punumber=10779

[8] C-based design. [Online]. Available: http://www.mentor.com/products/c-based_design/

[9] Xpres compiler. [Online]. Available: http://www.tensilica.com/products/xpres.htm

[10] R. Chaves, G. Kuzmanov, S. Vassiliadis, and L. A. Sousa, "Reconfigurable memory based AES co-processor," in *Proceedings of the 13th Reconfigurable Architectures Workshop (RAW 2006)*, April 2006, p. 192.

[11] NIST, "Announcing the advanced encryption standard (AES), FIPS 197," National Institute of Standards and Technology, Tech. Rep., November 2001.