Automatic hardware generation for the Molen reconfigurable architecture: a G721 case study

Dimitris Theodoropoulos *Yana Yankova* *Georgi Kuzmanov* *Koen Bertels*

Computer Engineering Lab, Delft University of Technology,
P.O. Box 5031, 2600 GA Delft,The Netherlands
email: D.Theodoropoulos@tudelft.nl

Abstract— **The advantages of the reconfigurable technology in terms of performance have been widely recognized. However, programming reconfigurable systems and designing hardware accelerators for them is not a trivial task. The Molen paradigm provides an easy to use approach to couple a General Purpose Processor (GPP) with custom designed reconfigurable accelerators both at program level and at hardware design level. In this case study, we illustrate the entire design flow to demonstrate how one can use the Delft-Workbench Automated Reconfigurable VHDL Generator (DWARV) tool, the Molen compiler and the Molen reconfigurable co-processor to accelerate a C application code in hardware. As a case study application, the G721 audio encoder is used. The implementation platform is a Xilinx VirtexII Pro XC2VP30-7 FPGA, which integrates two PowerPC 405 processors. The experimental results obtained after employing the described design flow suggest an overall application speedup of 2.7 times over a pure software implementation.**

Keywords— **Reconfigurable processors, FPGA, Molen paradigm, Automatic HDL generation**

I. Introduction

Over the last decade, Field Programmable Gate Arrays (FPGAs) became very popular, due to their capability of combining software flexibility with hardware performance. In addition, although today's microprocessor chips are becoming faster and integrate more than one CPU cores, still there are application areas (signal processing, multimedia) that can benefit from hardware accelerators. However, designing these accelerators and exploiting the FPGA advantages, requires the developer to understand in detail the software and hardware design concept. Therefore, tools and workbenches that assist the designers in the non-trivial development process are necessary.

One such tool chain is the Delft Workbench [2]. It is a semi-automatic tool platform targeting the Molen polymorphic organization [8], [6] and supporting the Molen programming paradigm [7]. The main idea is to automate as much as possible the design exploration and the final development process. More specifically the Delft Workbench research emphasizes on:

- Code profiling and cost modeling
- High level application graph transformation
- Re-targetable compilation
- VHDL generation

In this case study, we demonstrate how an application can be mapped to a polymorphic processor, using the tools in the Delft Workbench chain. More specifically, the G721 audio encoder application is first profiled and then accelerated, by mapping its most computational parts in hardware. An overall application speedup of 2.7 times over software execution is observed. Moreover, the designs providing this speedup were generated within seconds.

The remainder of the paper is organized as follows: Section II briefly describes the Molen organization, the DWARV toolset [9] and the Molen Compiler [5]. Section III depicts in detail the entire process of how one can use this framework to accelerate a software application in hardware, using as a case study the G721 audio encoder. The obtained results are also presented in Section III. Section IV concludes the paper.

II. Background

In this section, we briefly describe the Molen organization and the tools that are used during the design process.

A. The Molen organization

The Molen paradigm is used to speedup an application's execution by implementing its most critical functions as hardware accelerators, referred to as Custom Computing Units (CCUs). The MOLEN organization is depicted in Fig. 1. The main parts are the core processor (PowerPC in this case study), the reconfigurable co-processor (RP) and the Arbiter. The GPP's Instruction Set Architecture (ISA) is extended, in order to control the hardware accelerators (Reconfigurable Instructions - RI). The Arbiter fetches the application's instructions from the main memory. It partially decodes each one of them and checks whether it belongs to the standard or to the extended ISA and arbitrates them to the corresponding processor.

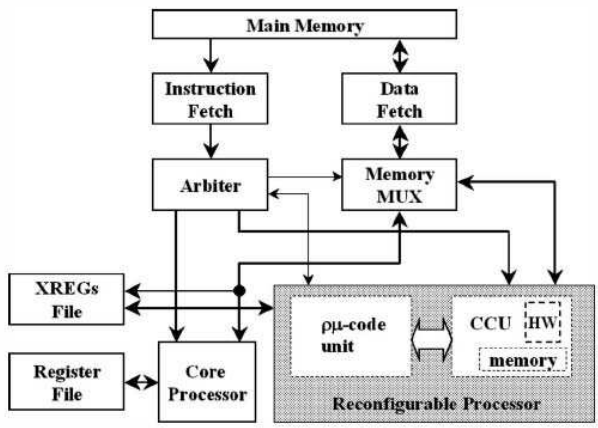

Fig. 1. The Molen organization.

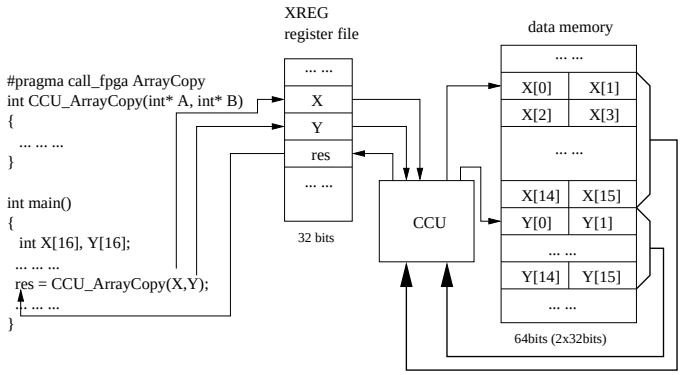

Fig. 2. Exchanging data between the GGP and the CCU.

The data transfer between the GPP and the RP is performed through exchange registers (XREG), organized in a register file. Fig. 2 gives an example of data exchange through the XREGs. The pragma annotation is for the Molen compiler to recognize which function will be implemented in hardware. During the program compilation, the memory addresses of *X* and *Y* are stored to the XREGs defined by the user before compilation. The GPP starts the program execution. When the Arbiter detects an RI it will send the appropriate reconfigurable microcode ($\rho\mu$-code) address to the $\rho\mu$-code unit. The latter contains the ρ-control store memory for $\rho\mu$-code storage. The $\rho\mu$-code unit is responsible for reading the microcode address and subsequently fetching the respective CCU's $\rho\mu$-code. This code initiates the execution of the addressed CCU. In the same time, the arbiter stalls the GPP. The CCU then reads the XREGs, which point to the address of *X* and *Y* in the data memory. When the CCU completes its computations, the returned value *res* is stored back to an XREG defined by the user prior to the compilation. The Arbiter detects that the CCU has finished and resumes the GPP. As the program execution is continued, the returned value is read and used.

The above example suggests that for every function implemented in hardware, as many input XREGs as the function's input parameters are used. If there is also a return value, an additional output XREG is allocated. As mentioned before, the XREGs allocation is specified just prior to the application compilation.

B. Design Flow

In this case study, the process of mapping an application to the polymorphic machine organization, described in the previous section, is divided into seven steps:

1. Profiling
2. Decision Making on HDL Generation
3. VHDL Generation (DWARV)
4. CCU Module Assembling
5. Compilation (MOLEN Compiler)
6. Synthesis and Bitstream Generation
7. Execution

Each of the above steps is described in more details in the case study context of Section III.

Two tools of the DelftWorkbench, namely the DWARV tool set and the MOLEN compiler, are used in the mapping process. The automated hardware generation in Step 3 is performed by the DWARV tool set, described in more details in Section II-C. The The MOLEN compiler, presented in Section II-D, performs the compilation in Step 5.

C. DelftWorkbench Automated Reconfigurable VHDL Generator (DWARV) Toolset

The DWARV organization is depicted in Fig. 3. The input is C code with pragma annotations that specify the code parts to be implemented in hardware. The C file goes through the DFG (Data Flow Graph) Builder, which translates it to a Hierarchical Data Flow Graph (HDFG). This graph consists of simple and compound nodes. The first represent arithmetic and logic operations, plus register and memory transfers. The latter are `for`-loops in the input code. The edges of the graph represent the data dependencies and the precedence operations order. The output of the DFG Builder is a serialized in a binary format HDFG for each annotated function in the input code.

In the "Hardware Description" file, the user can specify the following:

i. XREG file and data memory word size
ii. XREG file and data memory address size
iii. XREG file and data memory access time in clock cycles, assuming a 10 ns period
iv. big or little endian format
v. burst mode for both data memory and XREGs
vi. C data type size

vii. The address of the region in the XREG file, allocated for the I/O parameters of the translated function

The HDFG file is then passed as an input to the VHDL generator. The latter performs As Soon As Possible (ASAP) scheduling on the input graph considering the memory and XREG file latency. The generated VHDL design is FSM based and utilizes the CCU interface. The latter allows mapping of the generated designs as Molen co-processors.

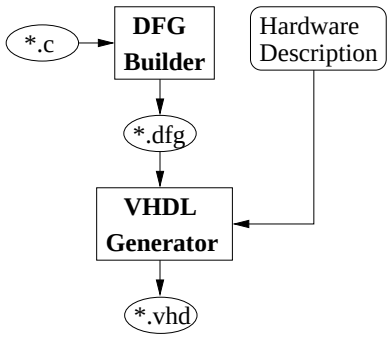

Fig. 3. The DWARV toolset.

D. The Molen Reconfigurable Compiler

Fig. 4 illustrates the Molen compiler interfaces. The input of the compiler is pragma-annotate C code and a hardware configuration file.The pragma annotation identifies the functions that are selected for hardware implementation. The configuration file contains the XREG addresses for the input and output function parameters. In addition, the SET and EXECUTE $\rho\mu$-code addresses for each function are specified.

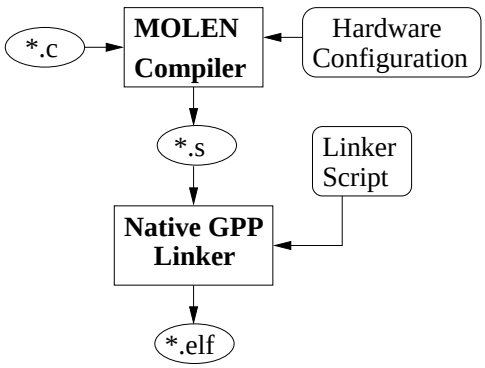

Fig. 4. The Molen compiler overview.

The compiler replaces the invocations of the pragma-annotated functions with the corresponding polymorphic instructions using the addresses specified in the configuration file. It also generates the necessary instructions for moving the parameters and the return value to and from the XREGs.

The output of the compiler is an assembly file that is processed by the GPP native linker. As an additional input the linker accepts a linker script, where the architecture parameters such as memory allocation for instruction and data are specified. The output is an *.elf file. This is a binary data file that contains an executable CPU code image.

III. G721 Case Study

In this section the entire process of mapping an application to a polymorphic processor is illustrated. While analyzing each step, it is also described how it is applied to the case study application. The obtained results after the application execution are also reported.

A. Experimental Setup

The G721 audio encoder from the MediaBench benchmark suite [4] is selected as a case study application. The prototyping platform is an offline version of the MOLEN polymorphic processor prototype. The prototype is implemented on Xilinx Virtex II Pro XUP board, xc2vp30-7ff896 device. The general purpose processor is PowerPC 405 (PPC), operating at 300MHz. The prototype allows implementation of up to 64 CCUs with operating frequency of 100MHz. The synthesis of the generated CCUs and the subsequent integration with the MOLEN prototype is performed within the Xilinx ISE 8.1 design environment. The compilation of the application, the generation of the CCU designs, and the synthesis and assembling of the designs are performed under Fedora Core 4 (2.6.14-1) Linux on AMD Athlon 64 Processor 3200+ with 2GB RAM.

The evaluation of the DelftWorkbench tool chain is based on two criteria. The first criterion is the runtime of the tools in the tool chain. These times are measured with the Linux `time` utility and they are reported in seconds. The second criterion is the achieved performance improvement of the case study application. This improvement is measured by executing the pure software and the hybrid software/hardware implementation of the case study application. The run times of the application for both implementations are measured using the internal PowerPC timer and reported in PowerPC cycles. The used timer is a counter that increments at 300MHz frequency. For each measurement, the timer base register is initialized with 0 and read at the end of the execution.

B. Design Flow

Fig. 5 depicts the flow of mapping an application to the polymorphic processor. This process is divided into seven steps, explained below.

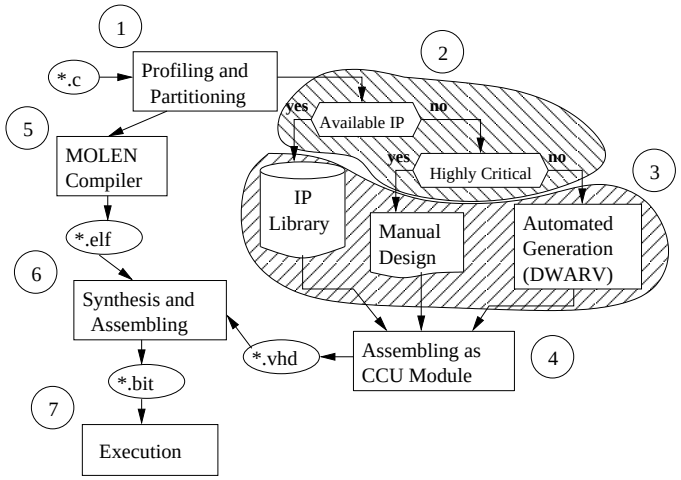

Fig. 5. The entire design flow.

STEP 1. Profiling: At the first step, the C application is profiled and the most time consuming functions are identified. In the current experiments, the profiling is performed at two stages. First, the application is profiled with the GNU gprof [3] dynamic profiler under Montavista Linux 2.4.20 on Xilinx ML 310 board (PowerPC 405). As a result of this profiling an ordered list of the execution percentage for each function is generated. The second stage of the profiling is carried on the execution platform. At this stage, the actual execution time percentage of the top most kernels is derived. To compute this percentage, the application and the kernels execution times are measured, using the PPC timer, in separate runs to avoid skewing of the results due to timer overhead.

For the considered application, the dynamic profiling identified two critical functions, namely *fmult* and *update*. In order to measure the execution times of these functions on the execution platform, a wrapper function for each kernel was created, as shown in the example below:

```
/* kernel body moved to a new
   function with all global
   data passed as parameters */
static
int fmult_in(int an, int srn,
             short power2[])
{
... ... ...
}

/* original function used as
   a wrapper */
int fmult(int an, int srn)
{
   int res;
   /* timer base register (TBR)
      initialization */
   init_timer();
   res = fmult_in(an, srn, power2);
   /* read and print the  TBR value*/
   read_timer();
   return res;
}
```

The wrapper is created with the original function signature. The body of the original function is moved to a function with modified signature. In the wrapper function the kernel function is called. Prior to the kernel call, a timer initialization function is invoked (`init_timer()`). Immediately after the kernel call, a routine (`read_timer`) that reads the timer value and prints it to the standard output is placed.

The wrapping of the kernels serves one more goal. Namely, in this way the application is made compliant to the requirement of the tool chain that all data accessed from the hardware accelerated function have to be passed as parameters to this function. In other words, no global data may be accessed from within the accelerated functions. In the example above, the original `fmult` function accesses the static global array `power2`. In the restructured code, this array is passed as a parameter to the kernel from the wrapper function.

After the application is restructured, it is compiled and executed on the prototyping platform. The obtained profile is shown in Table I. The second column of the table shows how many times the corresponding function was invoked. The third column reports the computed percentage of the application execution time spent in the respective function. As it can be observed from the presented example above, the recorded execution time includes also the time necessary to pass the function parameters and to read the return value. The forth column of the table shows the maximum overall application speedup that can be achieved according to the Amdahl's law [1]. The last row of the table reports the sum of the kernels' percentage and the theoretical application speedup if both kernels are executed in zero cycles.

TABLE I
G721 APPLICATION PROFILE

Function name	**Times called**	**Percentage**	**Speedup Limit**
fmult_in	124,080	53.62%	2.16
update_in	15,510	27.57%	1.38
combined	-	81.19%	5.32

STEP 2. Decision Making on HDL Generation: After the most critical kernels are identified, it is decided how the corresponding hardware implementation will be derived. Three possible approaches are considered. If the identified kernel function is available as an IP library core,

it is instantiated from there. Otherwise, the corresponding design is either developed manually or generated by DWARV. The decision for manual or automated generation is based on how critical is the selected kernel and on the current stage of the application design cycle. If the application development process is still in the design space exploration phase, automated generation is preferred as it is much faster. If the development process is in the final (release) stage, manual implementation is considered if the previous design stages have shown that the particular kernel is highly critical and the automatically generated design cannot meet the design requirements.

STEP 3. VHDL Generation: Prior to the automated VHDL generation, the selected functions for hardware generation are annotated with *#pragma to_dfg*. In addition, the allocated for each function XREGs are specified in the DWARV hardware configuration file.

For the kernels in the case study application, the following XREGs allocation is performed: for *update_in* function the XREGs at address 0x56 to 0x68, and for *fmult_in* function the XREGs at address 0x69 to 0x6C. This allocation is specified in the `config.v` configuration file as shown below:

```
** fmult_in
xreg_start_address = 069
** update_in
xreg_start_address = 056
```

It can be observed that only the first address of the allocated registers range is specified. The tools in the DelftWorkbench tool chain require all function parameters to be mapped to a continuous section of the XREG file. Therefore, it is not necessary the addresses of the registers allocated for the rest of the parameters to be specified. Another observation that should be made, is that there is no explicit allocation of an XREG for the return value of the *fmult_in* function. The reason is that DWARV currently maps the return value to the XREG that immediately follows the input parameters' XREGs. In the particular case, the return value will be mapped to address 0x6C in the XREG file.

After the necessary configuration options are specified, the designs for the annotated functions are generated. The DWARV execution times are given in the first two rows of Table II. As both functions are located in one C file, the corresponding `*.dfg` files are generated in a single run of the DFG Builder. The DFG Builder generates separate `*.dfg` file for each annotated function, which are processed in separate runs of the VHDL Generator. Therefore, the reported VHDL Generator time in the table is the sum of the times recorded for the processing of `fmult_in.dfg` and `update_in.dfg` files.

TABLE II
COMPILATION AND SYNTHESIS TIMES

Tool name	Run time, sec
DFG Builder	14.515
VHDL Generator	0.697
XST	263.341
Assembling	869.169

STEP 4. CCU Assembling: After the designs for each kernel are derived, the corresponding $\rho\mu$-code addresses are determined and stored in the $\rho\mu$-code memory. Fig. 6 shows each accelerated function's CCU connection with the XREGs, data memory, $\rho\mu$-code memory and the Arbiter. The $\rho\mu$-code memory currently is implemented as a lookup table that gives the correspondence between the CCU address encoded in the `EXECUTE` instruction and the corresponding CCU code. The encoded address serves as an index in the $\rho\mu$-code memory. The $\rho\mu$-code stored at that address is used to select through the de-multiplexer the corresponding CCU.

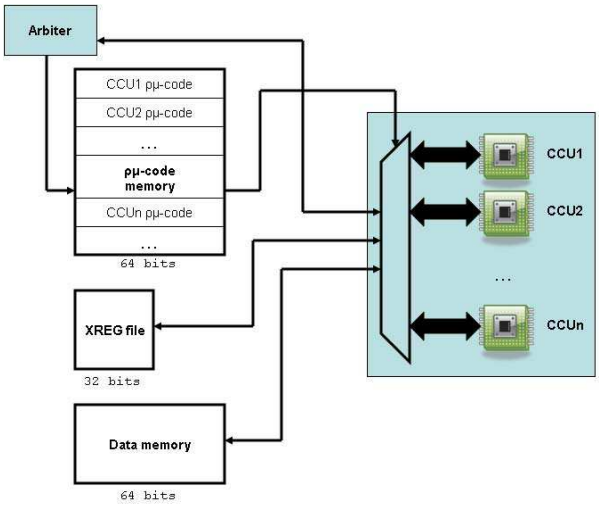

Fig. 6. VHDL design assembly and connection with the Molen infrastructure.

The de-multiplexer is described in a pre-prepared VHDL design. This design is instantiated for each application and serves as a top module in the final VHDL design. The instantiation comprises of instantiation of the CCUs designs and specification of the invocation code in the contained de-multiplexer template. This is illustrated in the example below:

```
architecture Behavioral of CCU is
component FMult
Port(
...);
end component;
```

```
component Update
Port(
...);
end component;
...
process
begin
...
 if start_op='1' then
    check_s<='1';
 end if;

 if check_s='1' then
   --initiate fmult execution
   if (MIR(1)='1') then
      start_op_FMult_s<='1';
   end if;
   if (MIR(0)='1') then
     start_op_update_s<='1';
   end if;
   check_s<='0';
 end if;
...
end process;
...
FMult_i: FMult port map (
   ... ...
   start_op => start_op_FMult_s,
   ... ... );

Update_i: Update port map (
   ... ...
   start_op => start_op_Update_s,
   ... ... );
...
end Behavioral;
```

The generated designs for *fmult_in* and *update_in* are instantiated as `FMult` and `Update` components. The *update_in* function is mapped to address 1 and the *fmult_in* function is mapped to address 2. Currently, one-bit positional encoding of the address is used as a $\rho\mu$-code. Therefore, if the first bit of the `MIR` word is set to 1, the *update* kernel is initiated. If the second bit is set to 1, the *fmult* kernel is invoked.

STEP 5. Compilation: Prior to the compilation, the selected for hardware implementation functions are annotated with *#pragma call_fpga operation_name*. The allocated XREGs for the function parameters and the defined $\rho\mu$-code execution addresses are specified in the configuration file. The argument of the pragma is a string identifier that relates the annotated function with the corresponding entry in the compiler configuration file. An example of the annotations and the configuration file entries for the case study application are shown below:

```
/* annotation */
#pragma call_fpga fmult
static
int fmult_in(int an, int srn,
             short power2[])
{
   ... ...
}

/* configuration file */
Pragma Name fmult
Input XR 69
Output XR 6C
EXEC address 2
```

As it can be seen from the example, the pragma argument can differ from the function name. The allocated previously XREG addresses for the function's parameters and return value are specified in the `XRin` and `XRout` fields, respectively. Again, the input parameters mapping is specified only with the starting address of the allocated section in the XREG file. Nevertheless, unlike the DWARV's configuration file, the allocated for the return value XREG address has to be explicitly specified.

After all necessary configuration parameters are set, the application is compiled. In the current experiments, an online version of the MOLEN compiler was used. Hence, it was not possible to distinguish between compilation and network transfer time. Therefore, compilation time results are not presented.

STEP 6. Synthesis and Bitstream Generation: After the necessary VHDL designs and the elf file are generated, they are merged with the MOLEN prototype and the final bitstream is generated. For that purpose, third-party CAD[1] tools are used. In the current experiments, the designs are synthesized and assembled in Xilinx ISE 8.1 design environment.

The assembling is performed at two steps. First, the designs of the de-multiplexer and the CCUs are synthesized. The synthesis estimation of the necessary resources and the execution frequency for each kernel, as well as for the entire CCU design are reported in Table III. It can be observed, that although automatically generated, the designs occupy only 30% of the available area. This allows their simultaneous implementation in the hardware. In addition, the estimated frequency shows that the timing constraints (100MHz CCU operating frequency) are not violated. Hence, it is possible the automatically generated designs to be used as hardware accelerators.

The time, necessary for the designs to be synthesized is reported in the third row of Table II. It can be observed that the time, necessary the designs to be generated is only a fraction ($\sim 6\%$) of the time, necessary for their synthesis.

[1]Computer Aided Design

TABLE III
SYNTHESIS ESTIMATES

Kernel	Slices	Flip Flop	LUT	MULT18X18	Frequency, MHz
update	2779 (20%)	2939 (10%)	4298 (15%)	0	169.926
fmult	1029 (7%)	835 (3%)	1769 (6%)	2 (1%)	129.659
combined	4142 (30%)	4260 (15%)	6709 (24%)	2 (1%)	126.062
Device Capacity	13696	27392	27392	136	N/A

This shows that the DWARV toolset is able to generate fast designs that are within the resource and timing limits.

After the designs are synthesized, they are merged with the MOLEN infrastructure. The adopted modular implementation scheme is depicted in Fig. 7. *CCU* is the design that integrates all the accelerators, *CLK_GEN* is the clock generation module, *$\rho\mu$-code memory* is the memory for storing the $\rho\mu$-code and *MOLEN* is the pre-synthesized Molen infrastructure. After the assembling of the separate modules, the generated `*.elf` file is stored in the prototype's program memory and the final bitstream is generated. The time, necessary for the modules to be assembled is shown in the last row of Table II.

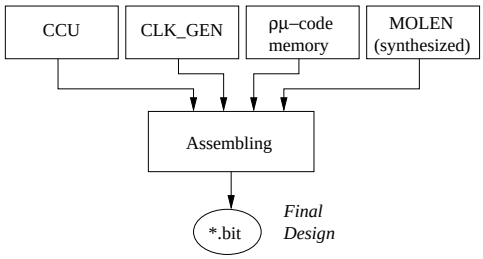

Fig. 7. Modular implementation and final design generation.

STEP 7. Execution: After the bitstream is derived, the application is executed on the prototyping platform. The recorded execution times are reported in Table IV. These times are measured in PowerPC (PPC) cycles in the same way as described in Step 1. The third row of the table reports the obtained overall application speedup. The last row shows how close is the achieved speedup to the theoretical limit as constituted by the Amdahl's law (see Table I). The obtained speedup is only 50% of the theoretical maximum. Nevertheless, the hardware kernels that provided this speedup were generated within seconds. This allows for fast evaluation whether the hardware implementation of a given function would be beneficial. If the performance improvement provided by the generated designs is not sufficient, manual implementation of the corresponding kernels can be considered.

TABLE IV
APPLICATION PERFORMANCE IMPROVEMENT.

Software Execution, PPC Cycles	387,402,570
Hardware Execution, PPC Cycles	143,084,730
Speedup	2.71
Efficiency	50.93%

IV. CONCLUSIONS

In this paper, the automated mapping of an application to a polymorphic platform was illustrated. The MOLEN modular design allowed easy integration of the custom accelerators. The evaluated tools provided support on each step in the design cycle. The MOLEN compiler allowed standard compilation flow of the application. The hardware/software interface code was automatically inserted and the details remained invisible for the user. The DWARV toolset generated designs that were within the resource and timing limits. The experimental results suggested that the generated designs provided overall application speedup of 2.7 times. The obtained speedup was modest compare to the speedup that could be achieved with manually crafted designs. Nevertheless, the DWARV designs were generated in a couple of seconds, while several men months would be necessary for the development of a highly optimized manual design.

V. ACKNOWLEDGEMENTS

This work was partially sponsored by hArtes, a project (IST-035143) of the Sixth Framework Programme of the European Community under the thematic area Embedded Systems; the Dutch Technology Foundation STW, applied science division of NWO and the Technology Program of the Dutch Ministry of Economic Affairs (project DCS.7533); and the RCOSY project (DES-6392) financed by the Dutch Science Foundation (STW) and the Associated Compiler Experts (ACE).

REFERENCES

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of AFIPS 1967 Spring Joint Computer Conference*, pages 483–485, 1967.

[2] Delft Workbench, Faculty of Computer Engineering, TU Delft. http://ce.et.tudelft.nl/DWB/.

[3] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[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*, pages 330–335, 1997.

[5] E. M. Panainte, K. Bertels, and S. Vassiliadis. The Molen Compiler for Reconfigurable Processors. *ACM Transactions in Embedded Computing Systems (TECS)*, 6, Feb. 2007.

[6] S. Vasiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte. The Molen Polymorphic Processor. *IEEE Transactions on Computers*, 53(11):1363 – 1375, Nov. 2004.

[7] S. Vassiliadis, G. N. Gaydadjiev, K. Bertels, and E. M. Panainte. The Molen programming paradigm. *Systems, Architectures, Modeling, and Simulation*, 3133:1 – 10, July 2003.

[8] S. Vassiliadis, S. Wong, and S. D. Cotofana. The Molen $\rho\mu$-coded Processor. *Field-Programmable Logic and Applications*, 2147:275 – 285, Aug. 2001.

[9] Y. D. Yankova, G. Kuzmanov, K. Bertels, G. N. Gaydadjiev, J. Lu, and S. Vassiliadis. DWARV: DelftWorkbench Automated Reconfigurable VHDL Generator. In *Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL07)*, Aug. 2007.