HLL-to-HDL Generation: Results and Challenges

Yana Yankova, Koen Bertels, Stamatis Vassiliadis, Georgi Kuzmanov, Ricardo Chavez
Computer Engineering Laboratory
Delft University of Technologies
Email: {y.d.yankova, k.l.m.bertels, s.vassiliadis, g.kuzmanov}@ewi.tudelft.nl,
ricardo.chaves@inesc-id.pt

Abstract—**This paper presents preliminary results of automated hardware generation from C code and discusses specific challenges. The research is part of a bigger project that aims to provide a semi-automatic tool platform for hardware-software co-design in the context of the reconfigurable computing systems. We present a case study involving the AES encryption algorithm. The automatically generated VHDL is compared to a manually crafted design. The generated design, as well as the manual one, are synthesized in Xilinx ISE 6.3i and the reported results are obtained from the actual execution on the MOLEN polymorphic processor. Even though the automatically generated VHDL does not contain any optimizations, speedup of 7 is observed. However, compared to manually tuned VHDL, there is still a difference of several orders of magnitude. This paper will explore the differences that explain the performance gap and identify sources of improvement and necessary optimizations that have to be implemented. Some of the considered optimizations regard parallelization of the execution, memory accesses and data location, optimal utilization of the available hardware resources (on-chip memories, multipliers, etc), as well as identification of the most suitable computation model for a given algorithm.**

Keywords— **HW-SW co-design, high-level synthesis, reconfigurable computing, HLL, HDL**

I. Introduction and Background

The main advantage of reconfigurable computing (RC) is that it combines software flexibility with hardware execution speed. However, RC systems have the main disadvantage of a non-trivial application development process, where both software and hardware design skills and knowledge are necessary. Tools and workbenches that bridge the gap between hardware and software design are therefore necessary. Components of such workbenches assist the designer throughout the entire design flow starting with the initial partitioning of the application in software and hardware segments up to the final implementation of the hybrid executable code. Such platform is the **Delft Workbench**. It aims to provide a semi-automatic platform for hardware-software co-design in the context of the reconfigurable computing systems. The targeted over-
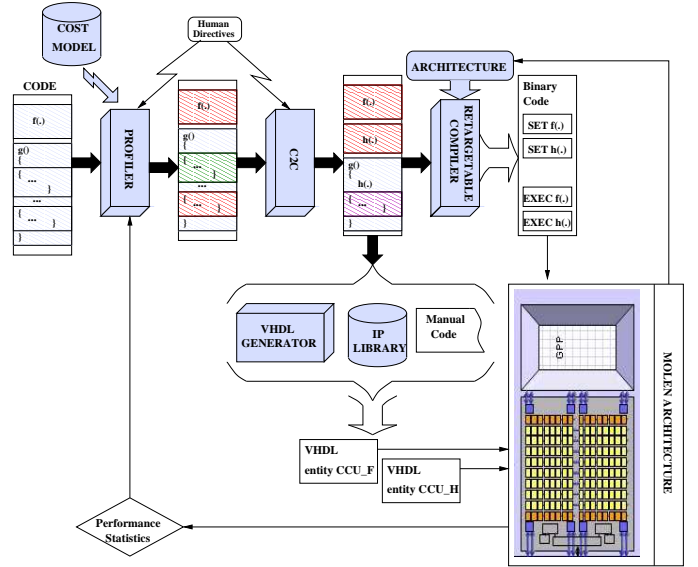

Fig. 1. Delft Workbench

all design flow is presented in Fig 1. A C application is processed by a profiler that is based on a cost model and selects code segments that are promising hardware-implementation candidates. Next, the candidates are futher examined by a C2C compiler and the application is restructured in function of the reconfigurable platform. At this point the design flow forks into the VHDL generation and backend compilation. The backend compiler[1] replaces the annotated code segments with the necessary reconfiguration and hardware execution instructions. In addition, the compiler provides optimal scheduling of hardware reconfiguration in order to hide the reconfiguration latency.

The selected hardware segments are passed through the VHDL generation branch. If a given design is available in a hardware IP library, it is instantiated from there. Otherwise, if the selected code is extremely critical and requires highly optimal hardware implementation, manual HDL code should be written.

---

[1]An online prototype of the MOLEN compiler is available at [1]

For the purposes of fast prototyping and fast performance estimation during the design space exploration, automated VHDL code is considered. This functionality, with the necessary optimizations, would allow non-hardware designers to develop application for an RC system.

The Delft Workbench targets the MOLEN machine organization (Fig. 2) and the MOLEN programming paradigm [2]. The **MOLEN polymorphic processor** [3] couples a general purpose "*core*" processor (GPP) with a "*reconfigurable processor*" (RP). The communication between the GPP and the RP is performed via exchange registers (XREGs). The reconfigurable processor consists of $\rho\mu$-*code unit* and *custom configured unit* (CCU). The $\rho\mu$-code unit reconfigures the hardware and initiates the execution on the CCU. The CCU contains memory and reconfigurable hardware.
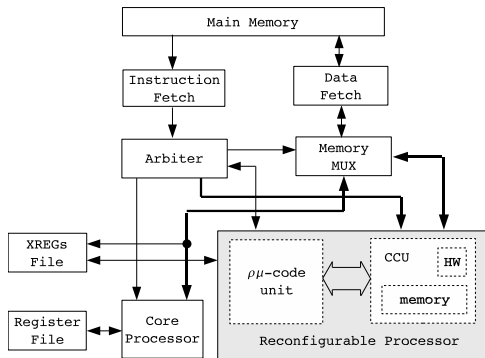

Fig. 2. MOLEN Machine Organization

The automated VHDL generation within Delft Workbench is performed by the tool set presented in this paper. The input of the tool set is a pragma annotated C code and the output is a VHDL design to be mapped and executed on the CCU. The MOLEN backend compiler was used to generate the corresponding executable for the GPP and the execution was performed on the MOLEN prototype [4]. Although, the current state of the VHDL generation does not offer rich set of optimizations, a speedup of 7 times over a software execution is observed. Moreover, it was estimated that the attained speedup is approximately 90% of the theoretically achievable speedup. Comparison with a manually crafted design for the same application is performed and some possible optimizations are discussed.

The rest of the paper is organized as follows. Section II presents the current structure and functionality implemented in the VHDL generation tool-set being developed. Section III describes the case study application and the experimental platform. Section IV presents results from the actual execution of the automatically generated VHDL code as well as comparison with a manually crafted design. Section V discusses some of the existing C-to-VHDL research projects and Section VI outlines further research directions and concludes the paper.

II. THE VHDL GENERATOR

The input of a VHDL generation tool in the general case is C or another high-level language code. The tool processes the input code and generates as output a VHDL model. The processing includes several phases of analysis, transformations, and optimizations of the code in order to derive a suitable hardware design. The performed optimizations and transformations can be roughly separated to high-level and low-level ones. The high-level transformations aim to expose and exploit the available parallelism. Another goal of these transformations is to transform the sequential control-flow oriented algorithm to a concurrent and data-flow oriented form that is suitable for hardware implementation. Derivation of the most suitable computation model could also be added in this group, although usually the algorithm is mapped to a pre-determined model. The low-level optimizations target the optimal utilization of the available resources. This optimal utilization is expressed in reserving the minimal amount of resources that would guarantee a correct execution given a minimal execution time. Additionally, the number of the idle resources and the time in which they are idle also has to be minimized.

The preliminary version of the VHDL generation tool-set, implemented within the Delft Workbench, consists of two parts (see Fig. 3): a data flow graph builder (DFG) and a VHDL generator. The **DFG Builder** is implemented within the SUIF2 compiler framework [5]. It accepts as input pragma annotated C code. The pragma annotation is used to mark the functions that have been selected for hardware implementation. The purpose of this tool is to perform the necessary high level optimizations of the input code and to transform it into a form, suitable for hardware generation. The output of the tool is a function's DFG in a binary format.

Being in its early development stage, the VHDL generation tool set offers limited functionality. The limitations concern the supported C constructs and the implemented optimizations. Currently, only *if*-statements and arithmetic and logic operations over a scalar or one-dimensional arrays of scalar data are
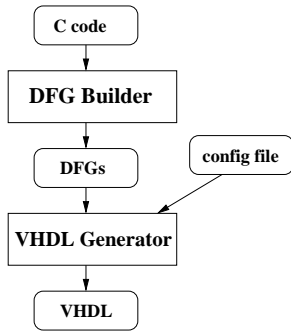
Fig. 3. VHDL Generation Tool Set

supported. The applied optimizations are also limited. Nevertheless, scalar replacement and static single assignment (SSA) transformations are performed. The former aims minimization of the memory accesses, while the latter simplifies the data-dependencies analysis and the DFG build. The SUIF framework and more particularly the Machine SUIF framework, provides SSA analysis pass. Nevertheless, in order for this pass to be triggered, the high-level structures in the input code have to be dismantled to jumps and labels which is not a suitable form for hardware generation. Therefore, in the developed DFGBuilder a customized and simplified SSA pass is implemented. In addition, the tool also performs preliminary code processing to restore some expressions (like the C conditional operation (? :) and the logic operations (&& and ||) that are dismantled by the front-end to branches and labels.

The **VHDL generator** is implemented as a standalone console application. It takes as input a DFG and a configuration file. The configuration file specifies the size of the memory word, the size of the memory address, the starting address of the function parameters (the XREGs starting address), the size of the different data types, the memory access times, and the endianness. The tool performs low-level optimizations and operations scheduling, taking into account the information from the configuration file, and generates a corresponding VHDL model.

The currently selected computation model is a staged execution in which the operations at each stage are data-independent and produce results that are used in the following stages. A finite state machine (FSM) operates as a sequencer of the execution. The VHDL generator divides the input DFG's operations into the necessary number of stages, taking into account the required memory accesses. The currently used scheduling strategy is *As Soon As Possible* (ASAP). Although no resource reuse optimizations are currently implemented, some straightforward optimizations like mem-

ory accesses pipelining are implemented.

An example of the implemented generation process is presented in Fig. 4. Fig. 4a) contains a part of a C code that is used as input of the DFG Builder. This code is processed at three passes. First, scalar replacement is performed (Fig. 4b). Next, the code is transformed into SSA form (Fig. 4c) and finally, the corresponding DFG is generated (Fig. 4d). The generated DFG is used as input for the VHDL generator, that schedules it (Fig. 4e) and generates the corresponding hardware design.

III. Experimental Setup

To evaluate the generated VHDL and to outline directions for optimizations, a comparison between automatically generated and manually crafted VHDL code is performed. This comparison is performed in terms of both resource utilization[2] and execution time.

AES Encryption The application, selected for the case study, is the AES encryption/decryption algorithm. The pseudo code for the AES encryption algorithm is presented in Fig. 5[3]. The structure of the software test application is shown in Fig. 6. The main function performs two tests: in Electronic Code-Book (ECB) and Cipher-Block Chaining (CBC) mode (Fig. 6b). The corresponding tests are run for 10, 12, and 14 rounds for encryption and decryption respectively. The encryption and the decryption are invoked through a dispatcher function (Fig. 6a) that calls the corresponding kernel as many times as the number of the blocks contained in the original data. The kernel functions are loops with iteration count equal to the number of the rounds. The loop bodies (shaded blocks in Fig. 6a) implement the actual encryption/decryption logic over a single block of data[4]. These loop bodies were processed by the described in the previous section tools and the generated VHDL designs are used in the experiments.

The C code that implements the AES algorithm, performs the operations, shown in Fig. 7. The shaded parts correspond to the loop bodies, for which VHDL designs are automatically generated. The loop control remains in the software. The bases addresses of the of the four tables ($T0$ - $T3$) are passed as input parameters. These tables contain precomputed byte values. Additional parameters are the s and the t

---

[2]Resources, area, and slices are used interchangeably in the paper.

[3]Taken from [6]

[4]A detailed description of the AES algorithm can be found in [6]

a)
```
tmp1[j] += block[0]*C11;
tmp2[j] += block[0]*C21;
tmp1[j] += block[1]*C12;
tmp2[j] += block[1]*C22;
tmp1[j] = tmp1[j] > limit_up ?
          limit_up : tmp1[j];
if(tmp2[j] < limit_dn)
          tmp2[j] = limit_dn;
```

b)
```
t1 = tmp1[j];
t2 = tmp2[j];
b0 = block[0];
b1 = block[1];
t1 += b0*C11;
t2 += b0*C21;
t1 += b1*C12;
t2 += b1*C22;
t1 = t1 > limit_up ?
     limit_up : t1;
if(t2 < limit_dn)
     t2 = limit_dn;
tmp1[j] = t1;
tmp2[j] = t2;
```

c)
```
t1 = tmp1[j];
t2 = tmp2[j];
b0 = block[0];
b1 = block[1];
t1_1 = t1 + b0*C11;
t2_1 = t2 + b0*C21;
t1_2 = t1_1 + b1*C12;
t2_2 = t2_1 + b1*C22;
t1_3 = t1_2 > limit_up ?
       limit_up : t1_2;
if(t2_2 < limit_dn)
       t2_3 = limit_dn;
else t2_3 = t2_2;
tmp1[j] = t1_3;
tmp2[j] = t2_3;
```

e)

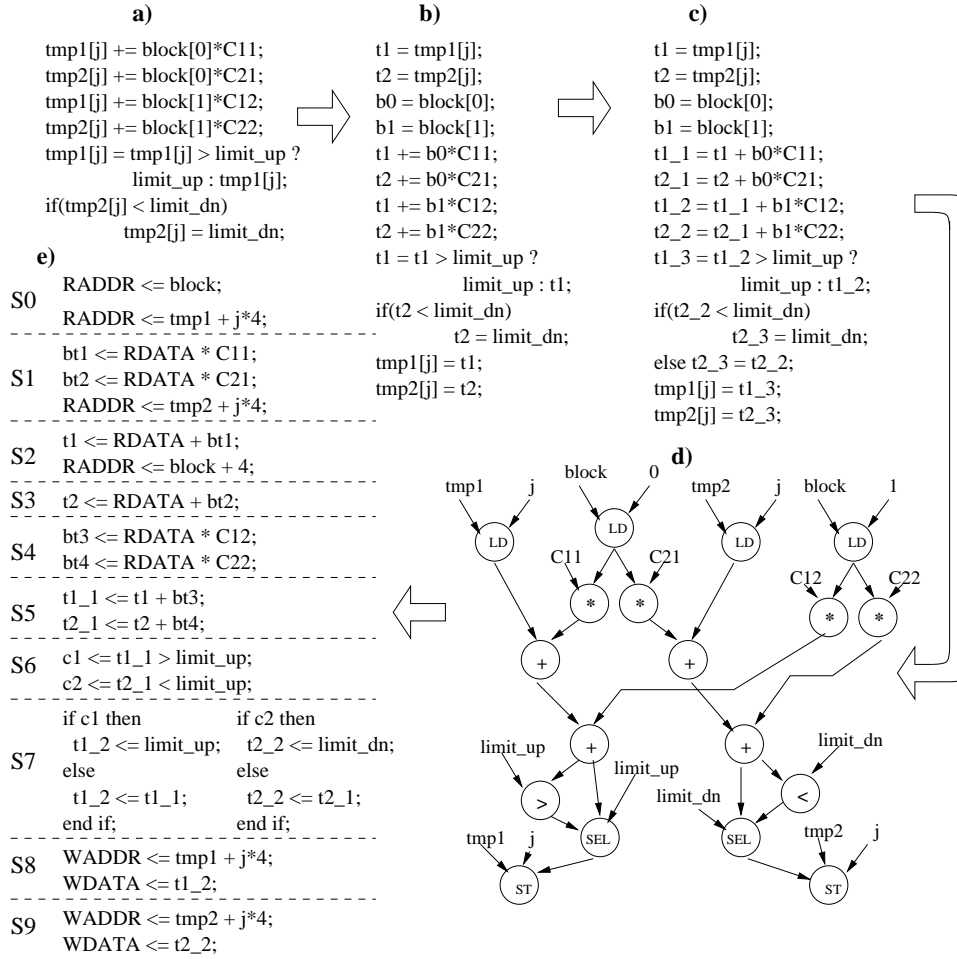|	
S0	RADDR <= block; RADDR <= tmp1 + j*4;

S1	bt1 <= RDATA * C11;
	bt2 <= RDATA * C21;
	RADDR <= tmp2 + j*4;

S2	t1 <= RDATA + bt1; RADDR <= block + 4;
S3	t2 <= RDATA + bt2;
S4	bt3 <= RDATA * C12; bt4 <= RDATA * C22;
S5	t1_1 <= t1 + bt3; t2_1 <= t2 + bt4;
S6	c1 <= t1_1 > limit_up; c2 <= t2_1 < limit_up;

S7	if c1 then if c2 then
	 t1_2 <= limit_up; t2_2 <= limit_dn;
	else else
	 t1_2 <= t1_1; t2_2 <= t2_1;
	end if; end if;

S8	WADDR <= tmp1 + j*4; WDATA <= t1_2;
S9	WADDR <= tmp2 + j*4; WDATA <= t2_2;

d)

Fig. 4. VHDL generation process

```
State = in

AddRoundKey(State, key[0 to Nb−1])
 for round= 1, round<Nr, round=round+1 do
SubBytes(State)
ShiftRows(State)
MixColumns(State)
AddRoundKey(State,key[round×Nb to (round+1)×Nb−1])
 end for

SubBytes(State)
ShiftRows(State)
AddRoundKey(State,key[Nr×Nb to (Nr+1)×Nb−1])

out = State
```

Fig. 5. Pseudo Code for AES Encryption.

blocks, the current round number (r), and the round key (rk). The updated value of the round number (r) is returned as output parameter to the software.

The manual implementation of the AES algorithm, described in [6], is based on the same C source, but it includes the functionality up to the dispatcher function (Fig. 6a).

Synthesis Platform The generated VHDL designs are synthesized in the Xilinx ISE 6.3i [7] design environment. The hybrid software-hardware execution of the application is performed on the MOLEN prototype [4], implemented using the Xilinx VirtexII Pro. The synthesis of the generated VHDL and the execution is performed for/on XC2VP30 chip. The selected clock frequency for the generated CCUs is 100MHz. The tests are run with 128 bits input data.

IV. Results

The reported post-synthesis estimation for the device utilization and the clock frequency are presented in Table I. The quoted numbers show that the speedup of both the encryption and the decryption would require 42% of the available area. Nevertheless, the manual implementation of both kernels, including also control and dispatch functionality occupies only 515 slices([6]) or 2.7% of the available slices.

The recorded execution times are reported in Table II and Table III. Table II contains hardware and software execution cycles for one round of encryption and decryption respectively. Table III shows the soft-

Implementation	Slices	Slice Flip Flops	4 Input LUTs	Frequency (MHz)
Encryption	2969 (21%)	3566 (13%)	4349 (15%)	157.282
Decryption	2964 (21%)	3565 (13%)	4352 (15%)	157.282
Manual	515 (3.7%)	not available	not available	182
Device Capabilities	13696	27392	27392	N/A[a]

[a]Not applicable
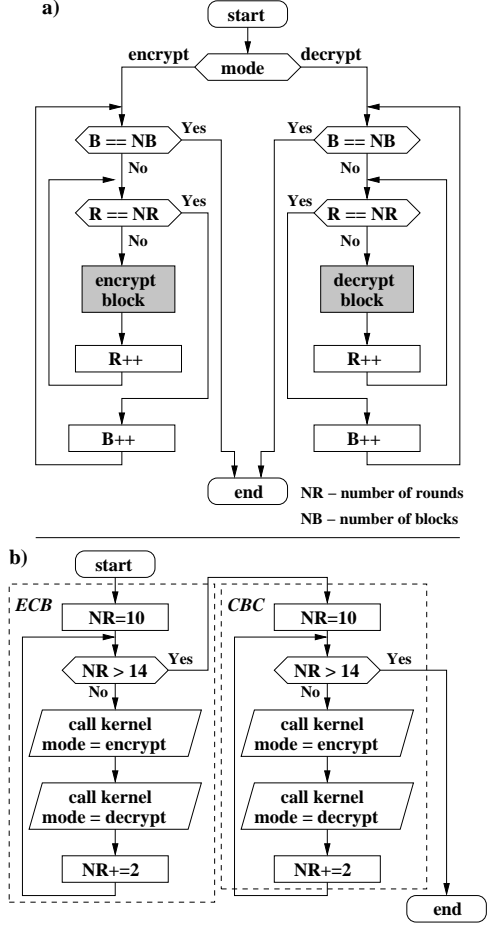
Fig. 6. AES application: a) encryption/decryption dispatcher and kernels; b) main function

Fig. 7. Particular AES Implementation

ware execution cycles of the entire application as well as the application cycles when either the encryption or the decryption is implemented in the hardware. The achieved speedup kernel wise is modest compared to the reported speedup of the manual implementation (kernel speedup of 43 times [6]). Nevertheless, related to the entire application, the results are promising. Although the manual implementation reports approximately 7.5 times higher kernel speedup, the achieved speedup by the automatically generated code, application wise, is close to the theoretically achievable speedup. Considering the part of the code, implemented automatically in the hardware (fifth column of Table III), and using Amdahl's law, the theoretically achievable speedup is computed (sixth column of Table III) and the achieved efficiency is reported in the seventh column of Table III. In other words, the achieved application speedup with the automatically generated code is approximately 90% of the theoretically possible application speedup for the given part of the code.

The obtained results show that even the straightforward HDL generation with almost no optimizations

TABLE III
PERFORMANCE IMPROVEMENT - APPLICATION

Implementation	Execution Cycles	Speedup	Invocation Count	Application Part	Theoretical Speedup	Efficiency
Software only	819372	N/A	N/A	100%	N/A	N/A
Encryption	565356	1.45	108	39%	1.65	88%
Decryption	621228	1.32	72	26%	1.35	98%

TABLE II
PERFORMANCE IMPROVEMENT - SINGLE
ENCRYPTION/DECRYPTION ROUND

Implementation	Software Cycles	Hardware Cycles	Speedup
Encryption	2997	408	7.35
Decryption	2997	408	7.35

can provide performance gains. However, the differences with the manual implementation in terms of both resource utilization and speedup are significant.

Resource Utilization In the selected application there are three major reasons for the increased area usage in the automatically generated code compared to the manually written one. The first reason, is duplication of functional units for the encryption and the decryption phase, even though the computations are the same (see Fig. 7). Whether the data are encrypted or decrypted is determined by the used tables $T0$-$T4$. In the manual code, the encryption and the decryption is performed by the same hardware module and the base addresses of the necessary tables are selected based on the mode. Such an implementation reduces by a factor of two the necessary area for the design. During the automated code generation, however, such design requires non trivial high-level analysis to discover the existing graph isomorphism.

The second reason for the larger area is the number of registers (or flip-flops, see Table I). In the current computation model, each operation is executed in one cycle and the result of the operation is stored in a register to be used in the following stages. In order not to increase the critical path (respectively the clock period), the compound operations are separated into series of simple ones. And since no optimizations are performed in the current generator version, three 32 bit registers (96 flip-flops, see Fig. 8a) and Fig. 8b)) are used in the computation of each table index (see Fig. 7), while only one register (32 flip-flops) is actu-
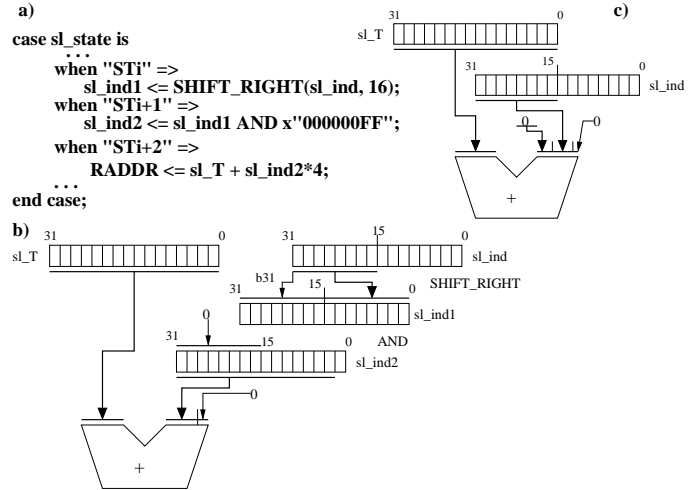

Fig. 8. Index Computation: a) VHDL code b) Inferred Hardware c) Sufficient Hardware

ally necessary (Fig. 8c)).

The third reason for the bigger area consumption is the number of the look-up tables (LUTs) (see Table I). The core computation during the encryption/ decryption process is the XOR sum over five operands. This computation translates to the VHDL code, shown in Fig. 9. The XOR operations usually are implemented as LUTs. As the operands are 32-bit, 32 LUTs are assigned to one XOR operation in the code. During the synthesis process for each operation dedicated resources are allocated. In the implemented code there are 32 32-bit XOR operations. This translates to 1024 LUTs dedicated to perform the XOR operations. However, one can notice that the different XOR operations cannot be executed in parallel due to the memory accesses (see Fig. 9). Moreover, each state produces valid results only once during the execution. Hence, the resources dedicated for the XOR operation in one state, can be reused for an XOR operation performed in another state. Of course, such optimization is not a "sure win" as multiplexers have to be dedicated to select the necessary inputs at each stage.

The above outlined optimizations are a small part of the high-level and low-level optimizations that can

```
case sl_state is
    . . .
    when "STi" =>
        ADDR <= sl_td0_s0a;
    when "STi+1" =>
        ADDR <= sl_td1_s3a;
    when "STi+2" =>
        ADDR <= sl_td2_s2a;
        sl_tmp1 <= RDATA;
    when "STi+3" =>
        ADDR <= sl_td3_s1a;
        sl_tmp2 <= RDATA XOR sl_tmp1;
    when "STi+4" =>
        ADDR <= sl_rk_4a;
        sl_tmp3 <= RDATA XOR sl_tmp3;
    when "STi+5" =>
        sl_tmp4 <= RDATA XOR sl_tmp3;
    when "STi+6" =>
        sl_t0 <= RDATA XOR sl_tmp4;
    . . .
end case;
```

Fig. 9. $t0$ computation

be performed in order for the resource utilization to be minimized. Some of these optimizations are trivial and "sure win". However, other optimizations require more complex analysis (graph isomorphism, eg). Moreover, the implementation of most of the optimizations is at a certain price (eg, the reduction of the XOR LUTs increases the number of the multiplexers, that, at a certain point, could nullify the effect of the decreased LUTs). Therefore, evaluation of the potential gains and penalties is necessary.

Execution Time Considering the execution time, respectively the achieved speedup, two possible optimizations can be noticed at a first glance in the studied application. In the automatically generated code, the loop control and the mode selection are performed by the software. In the manual design, this functionality is implemented in the hardware. This reduces the execution time in several ways. First, the execution flow switches less between hardware and software. Second, less parameters are transferred to the hardware (eg, s, t, r become all local data), which reduces the parameter fetch time. Another reason is that the tables with the pre-computed values are local for the design, while in the automatically generated designs these tables reside in main memory. This fact implies that the start addresses of the tables are not transferred as parameters (reduced times for parameters fetch). Second, the memory access times are shorter in the manual design. Moreover, in the automated design each array element is fetched separately from the memory, while in the manual design the fact that the BRAMs have 64-bit word is exploited. The

outlined two optimizations (implementing the control and transferring the tables in the hardware) are not that difficult to implement for the particular C code. However, the implementation of only these two optimizations within the current computation model, although beneficial for the performance to some degree, would not be enough to result in the speedup, reported for the manual design. In order to close the gap to the manual design, the semantic of the computation should be inferred and the most suitable computation model should be selected.

V. RELATED WORK

The automated HDL generation is in focus of multiple research projects. The ROCCC project [8][9] aims parallelization of the computation. Also off-chip memory accesses optimization is considered. The target application domain of the ROCCC project is image and signal processing. The DEFACTO project [10] exploits the fine-grain parallelism, found within the loop nests of the image processing applications. Considering multiple memory banks, customized data layout is also applied ([11]). Another major focus in the DEFACTO project is the design space exploration, and more particular finding the optimal unroll factors for the loop nests ([12]). The SPARK project [13], [14] targets also the image processing and multimedia domain. The emphasis in the project is speculative code motion that aims to increase instruction level parallelism.

To circumvent problems like data ambiguity and automatic detection of task-level parallelism, various projects develop derivates of the C language. The SA-C language [15], [16] is developed to facilitate the expression and the subsequent hardware implementation of image processing algorithms. The Streams-C language [17] follows the Communication Sequential Processes (CSP) model and implements it in the context of the reconfigurable computing. The application parallelization is performed by the designer. The controllers are instantiated from libraries and the data path generation is performed by the MARGE [18] tool. The optimization emphasis is on loop scheduling and pipelining.

While in ROCCC, SPARK, DEFACTO, SA-C, and Streams-C projects, entire loop nests are mapped to the hardware, in the Garp compiler [19], [20] hyperblocks from the most frequently executed paths in the loop nests are formed and implemented in the hardware. The Garp hardware is not based on off-the-shelf reconfigurable hardware. Hence, there is no standard

synthesis tool to generate the final bitstream for the Garp architecture. Therefore, the Garp compiler is emphasizes on generation and optimization of low-level modules (adders, multipliers, etc).

The projects listed above (as well as many other research efforts) suggest various optimizations and transformations to be applied during the automated HDL generation. Those optimizations vary from high-level parallelizing techniques to low-level scheduling and resource allocation strategies. Different hardware architectures and computational models are also suggested. The low level optimizations would be beneficial applied to any algorithm mapped to the hardware as they address the low-level hardware utilization. However, the high-level optimizations and the computation models are predominantly suggested within the context of the image processing applications and would not bring significant performance gains in other application domains. For example, within the **ROCCC** compiler a "smart buffer" ([21]) is suggested to optimize the memory accesses for "sliding window" operations over an array. Such operations are typical for the image processing applications - an array is traversed with a constant stride and a regular processing is performed. The gains offerred by the "smart buffer" rely mainly on full scalar replacement in the loop body. However, the code studied in the previous section does not allow full scalar replacement. Moreover, not all array data can be prefetched, as the index of the array is computed in the same iteration. Hence, applied on the studied application, the "smart buffer" would not bring a lot of performance gain.

The **DEFACTO** project also addresses the image processing application and optimization of the memory accesses. Their strategy is distribution of the processed data between multiple memory banks according to the access pattern. However, the suggested analysis and algorithm are oriented towards array indexing with affine function, which is not the case in the discussed example in the previous section.

The **SPARK** project emphasizes speculative execution over control structure boundaries in order to increase the fine grain parallelism. This approach is beneficial when the transformed code is control dominated. In the considered AES implementation, the control is minimized and the code is dominated by memory accesses.

The **SA-C** compiler is the predecessor of the ROCCC compiler and as such it targets the same application domain and offers similar optimizations. Again, the regular and sequential memory accesses are optimized,

but the access pattern in the AES application would not benefit from this optimizations.

The **Stream-C** compiler targets applications that are characterized by several distinguished phases of stream data processing. The high-level optimizations and the distribution of the processing are left to be performed by the designer. The compiler optimizations emphasize loop pipelining. However, the loops in the considered application cannot be pipelined due to data-dependencies and memory bandwidth constraints.

The **Garp** compiler searches for "hot" paths within loop nests and forms hyperblocks to be implemented in the hardware. This is beneficial for loop nests that contain multiple paths. However, the considered application has only one path in the kernel. The Garp compiler does not address memory optimizations and relies on the three pre-fetch queues, provided by the Garp architecture. But again, data pre-fetch is beneficial for access patterns, in which the next necessary array element can be predicted. In the current AES implementation, the next necessary array element becomes known just before initiation of the memory read.

VI. Conclusion and Future Work

The naive straightforward translation from C to VHDL is possible and brings performance gains as it was shown by the experimental results. Without any complex optimizations, a speedup of factor 7 was achieved compared with the pure software execution. Moreover, the achieved speedup application wise is approximately 90% of the theoretically achievable speedup considering the percentage of the code implemented in the hardware. However, the application speedup is still modest compared with a manually crafted design, where bigger part of the code is implemented in the hardware and additional optimizations are applied. In order to increase the quality of the automated designs, a C to VHDL compiler should be able to infer the semantic of the high-level code. Based on this semantic analysis, an appropriate computation model and optimization set should be selected.

References

[1] Molen backend compiler. [Online]. Available: http://ce.et.tudelft.nl/ molen/cgi-bin/read.php

[2] S. Vassiliadis, G. Gaydadjiev, K. Bertels, and E. M. Panainte, "The molen programming paradigm," in *Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation*, ser. Lecture Notes in Computer Science (LNCS), vol. 3133. Samos, Greece: Springer-Verlag, July 2003, pp. 1–19.

[3] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The molen polymorphic processor," *IEEE Transactions on Computers*, pp. 1363–1375, November 2004.

[4] Molen prototype. [Online]. Available: http://ce.et.tudelft.nl/MOLEN/Prototype/

[5] The suif 2 compiler system. [Online]. Available: http://suif.stanford.edu/suif/suif2/

[6] 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.

[7] Xilinx inc. [Online]. Available: http://www.xilinx.com

[8] Z. Guo, B. Buyukkurt, W. Najjar, and K. Vissers, "Optimized generation of data-path from c codes for fpgas," in *Proceedings of Design, Automation and Test in Europe (DATE'05)*, vol. 1, 2005, pp. 112–117.

[9] Riversite optimizing complier for configurable computing. [Online]. Available: http://www.cs.ucr.edu/ roccc/

[10] P. C. Diniz, M. W. Hall, J. Park, B. So, and H. E. Ziegler, "Bridging the gap between compilation and synthesis in the defacto system." in *Proceedings of 14th International Workshop on Languages and Compilers for Parallel Computing (LCPC'01)*, Cumberland Falls, KY, USA, Aug. 2001, pp. 52–70.

[11] B. So, M. W. Hall, and H. E. Ziegler, "Custom data layout for memory parallelism," in *IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004)*, San Jose, CA, USA, Mar. 2004, pp. 291–302.

[12] H. E. Ziegler and M. W. Hall, "Evaluating heuristics in automatically mapping multi-loop applications to fpgas," in *Proceedings of the ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays, (FPGA 2005)*, Monterey, CA, USA, Feb. 2005, pp. 184–195.

[13] S. Gupta, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau, "Using global code motions to improve the quality of results for high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 2, pp. 302–312, Feb. 2004.

[14] Spark: A parallelizing approach to the high-level synthesis of digital circuits. [Online]. Available: http://mesl.ucsd.edu/spark/

[15] W. Najjar, W. Bohm, B. A. Draper, J. Hammes, R. Rinker, J. R. Beveridge, M. Chawathe, and C. Ross, "High-level language abstraction for reconfigurable computing," *IEEE Transactions on Computers*, vol. 8, pp. 63–69, Aug. 2003.

[16] Compiling high-level programs to fpga configurations! [Online]. Available: http://www.cs.colostate.edu/cameron/

[17] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented fpga computing in the streams-c high level language," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM 2000)*, Apr. 2000, pp. 49–56.

[18] M. Gokhale, J. Kaba, A. Marks, and J. Kim, "Malleable architecture generator for fpga computing," in *Proceedings of SPIE, Design Methods And Tools For Reconfigurable Computing*, vol. 2914, 1996, pp. 208–217.

[19] T. Callahan, J. Hauser, and J. Wawrzynek, "The garp architecture and c compiler," *Computer*, vol. 33, no. 4, pp. 62–69, Apr. 2000.

[20] Automatic c compilation to sw + reconfigurable hw. [Online]. Available: http://brass.cs.berkeley.edu/compiler.html

[21] Z. Guo, B. Buyukkurt, and W. A. Najjar, "Input data reuse in compiling window operations onto reconfigurable hardware." in *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*, Washington, DC, USA, June 2004, pp. 249–256.