A LIGHTWEIGHT SPECULATIVE AND PREDICATIVE SCHEME FOR HARDWARE EXECUTION

Razvan Nane, Vlad-Mihai Sima and Koen Bertels

Computer Engineering Lab
Delft University of Technology
email: $\langle r.nane, v.m.sima, k.l.m.bertels \rangle$ @tudelft.nl

ABSTRACT

If-conversion is a known software technique to speedup applications containing conditional expressions and targeting processors with predication support. However, the success of this scheme is highly dependent on the structure of the if-statements, i.e., if they are balanced or unbalanced, as well as on the path taken. Therefore, the predication scheme does not always provide a better execution time than the conventional jump scheme. In this paper, we present an algorithm that leverages the benefits of both jump and predication schemes adapted for hardware execution. The results show that performance degradation is not possible anymore for the unbalanced if-statements as well as a speedup for all test cases between 4% and 21%.

I. INTRODUCTION

As the increase in frequency of the general purpose processors is becoming smaller and harder to obtain, new ways of providing performance are investigated. One of the promising possibilities to improve the system performance is to generate dedicated hardware for the computation intensive parts of the applications. As writing hardware involves a huge effort and needs special expertise, compilers that translate directly from high level languages to hardware languages have to be available, before this method is widely adopted. As C and VHDL are the most popular used languages in their fields, of embedded and hardware system development respectively, we will focus on compilers for C-to-VHDL. The algorithm presented here can be applied in theory to any such compiler. A C-to-VHDL compiler can share a significant part with a compiler targeting a general purpose architecture, still, there are areas for which the techniques must be adapted to take advantage of all the possibilities offered.

In this context, this paper presents an improved predication algorithm, which takes into account the characteristics of a C-to-VHDL compiler and the features available on the target platform. Instruction predication is an already known compiler optimization technique, however, current C-to-VHDL compilers do not take fully advantage of the possibilities offered by this optimisation. More specifically, we propose a method to increase the performance in the case of unbalanced if-then-else branches. These types of branches are problematic because, when the jump instructions are removed for the predicated execution, if the shorter branch is taken, slowdowns occur because (useless) instructions from the longer branch still need to be executed. Based on both synthetic and real world applications we show that our algorithm does not substantially increase the resource usage while the execution time is reduced in all the cases for which it is applied.

The paper is organized as follows. We begin by presenting a description of the predication technique and previous research, emphasizing on the missed optimization possibilities. In Section III we present our algorithm and describe its implementation. The algorithm is based on a lightweight form of speculation because it does not generate logic to roll back speculated values. It employs a lightweight form of predication because only some branch instructions are predicated, as well as keeping jump instructions. Section IV discusses the results and Section V concludes the paper.

II. RELATED WORK AND BACKGROUND

Given the code in Fig. 1 (a), the straightforward way of generating assembly (or low level code) is presented in Fig. 1 (b). We note that for any of the two branches there is at least one jump that needs to be taken. If the block execution frequency is known, an alternative approach exists in which the two jumps are executed only on the least taken branch.

Branches are a major source of slowdowns when used in pipelined processors as the pipeline needs to be flushed before continuing. Furthermore, branches are also scheduling barriers, create I-cache refills and limit compiler scalar optimizations. In order to avoid this negative effect, the concept of predication was introduced, which does not alter the flow but executes (or not) an instruction based on the value of a predicate. An example is given in Fig. 1 (c). In this scheme no branches are introduced, but, for a single issue processor more (useless) instructions are executed. In case of a multiple issue processor such instructions can be ”hidden” because the two code paths can be executed in

978-1-4673-2921-7/12/$31.00

parallel. We emphasize that the advantage of the predication comes from the fact that there are no branches in the code.

```
                    cond = cmp x,0
                    branchf cond, else
                    add r,a,b
    if (x)          branch end
       r = a + b;   else:              cond = cmp x,0
    else            sub r,c,d           [cond] add r,a,b
       r = c - d;   end:               [!cond] sub r,c,d

         (a)             (b)                (c)
```

Fig. 1. (a) C-Code; (b) Jump- ; (c) Predicated-Scheme.

The predication schemes assumes that the penalty of the jump is huge and thus branching has to be avoided. This is no longer true in the case of VHDL code. For the VHDL code there are no "instructions" but states in a datapath, controlled by a Finite State Machine (FSM). A straightforward implementation in VHDL of the jump scheme is presented in Fig. 2. We will present in the later sections the implications of the fact that the jumps are not introducing a huge delay. For this case, applying predication decreases the number of states from 4 to 2. We will show in the later sections how our algorithm can reduce the number of states even for unbalanced branches, a case not treated in the previous work.

A seminal paper on predication is [3], where a generic algorithm is presented that works on *hyperblocks* which extends the concept of basic blocks to a set of basic blocks that execute or not based on a set of conditions. It proposes several heuristics to select the sets of the basic blocks as well as several optimizations on the resulted hyperblocks and discusses if generic optimizations can be adapted to the *hyperblock* concept. Compared to our work, their heuristic

```
datapath
    state_1:
        cond = cmp x,0
    state_2:
    state_3:
        r=a+b;
    state_4:
        r=a-b;
    state_5:
        .... -- code after if-statement
FSM
    state_1:
        next_state = state_2
    state_2:
        if(cond)
          next_state = state_3
        else
          next_state = state_4
    state_3:
        next_state = state_5
    state_4:
        next_state = state_5
    state_5:
        ....
```

Fig. 2. Jump Scheme

```
void balanced_case(int *a, int *b, int *c, int *d,
                   int *e, int *f, int *result) {
    if (*a > *b)
            *result = *c + *d;
    else
            *result = *e - *f;
}
```

Fig. 3. Balanced if branches.

does not consider the possibility of splitting a basic block and does not analyse the implications for a reconfigurable architecture, e.g. branching in hardware has no incurred penalty.

The work in [4] proposes a dynamic programming technique to select the fastest implementation for if-then-else statements. As with the previous approach, any change in the control flow is considered to add a significant performance penalty. In [5], the authors extend the predication work in a generic way to support different processor architectures. In this work, some instructions are moved from the predicated basic blocks to the delay slots, but as delay slots are very limited in nature there is no extensive analysis performed about this decision.

Regarding the C-to-VHDL compilers, we mention Altium's C to Hardware (CHC) [6] and LegUp [7]. They translate functions that belong to the application's computational intensive parts in a hardware/software co-design environment. Neither of these compilers considers specifically predication coupled with speculation during the generation of VHDL code.

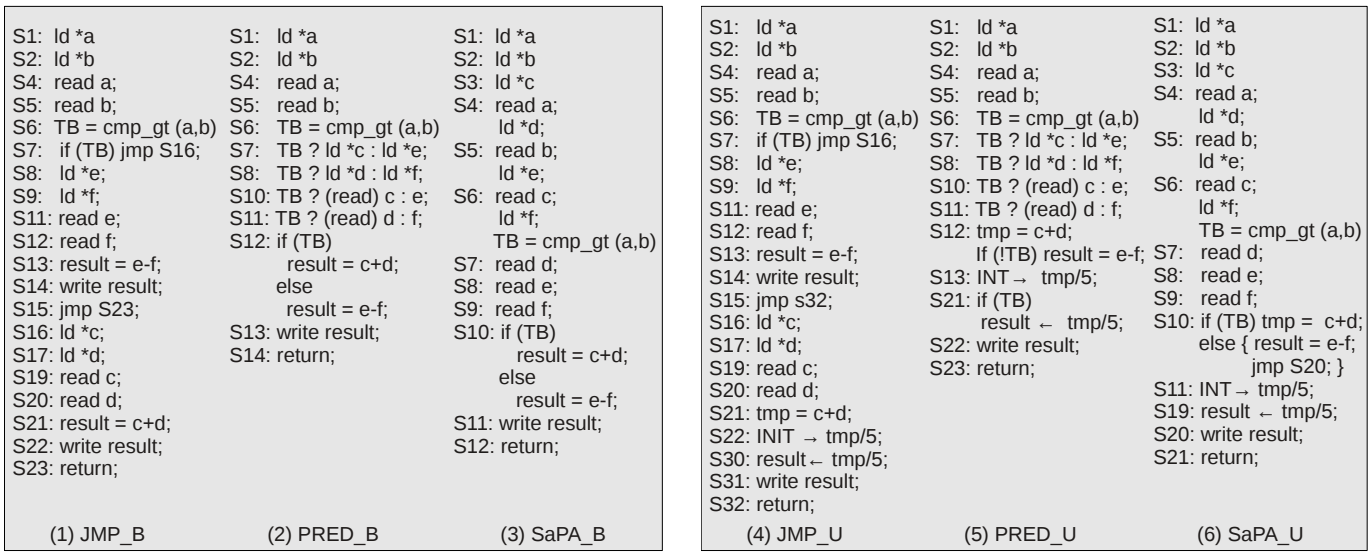
III. SPECULATIVE AND PREDICATIVE ALGORITHM

In this section, we describe the optimization algorithm based on two simple but representative examples which illustrate the benefit of including Speculative and Predicative Algorithm (SaPA) as a default transformation in High Level Synthesis (HLS) tools.

III-A. Motivational Examples

To understand the problems with the predication scheme (PRED) compared to the jump scheme (JMP), we use two functions that contain each one if-statement. The first, shown in Fig. 3, considers the case when the then-else branches are balanced, i.e. they finish executing the instructions on their path in the same amount of cycles, whereas the second case deals with the unbalanced scenario (Fig. 4). In these examples, we assume the target platform is the Molen machine organisation [2] implemented on a Xilinx Virtex-5 board. This setup assumes that three cycles are used to access memory operands, simple arithmetic (e.g. addition) and memory write operations take one cycle, whereas the division operation accounts for eight cycles.

The FSM states corresponding to the two examples are listed in Fig. 5(a) and 5(b). For each example, the



(a) Balanced Example

(b) Unbalanced Example

Fig. 5. Synthetic Case Studies.

```
void unbalanced_case(int *a, int *b, int *c, int *d,
                     int *e, int *f, int *result) {
    int tmp;
    if (*a > *b) {
            tmp = *c + *d;
            *result = tmp /5; }
    else
            *result = *e - *f;
}
```

Fig. 4. Unbalanced if branches

first column represents the traditional jump scheme ((1) and (4)), the middle columns ((2) and (5)) represent the predicated one and columns (3) and (6) shows the SaPA version. This column will be explained in more detail in the next section, as it is presenting the solution to the problem described here. Because each state executes in one cycle, the first five states are needed to load the *a* and *b* parameters from memory. In the first two states, the address of the parameters is written on the memory address bus. State three is an empty state and therefore is not shown in the figures. Finally, in states four and five the values of the parameters are read from the data bus. These operations are common for all possible implementations (i.e. for all combinations of balanced/unbalanced case study and JMP/PRED/SaPA schemes) shown by the (1) to (6) numbered columns in the two figures. Subsequently, the then-branch (TB) predicate is evaluated for the JMP cases (column (1) and (4)). Based on this value, a jump can be made to the then-branch states (states 16 to 22), or, in case the condition is false, execution falls through to the else-path (states 8 to 15). The number of states required for the unbalanced case, i.e. (4) JMP_U, is larger due to the additional division operation present in the then-branch. That is, in state 22 we initialize the division core with the required computation, whereas in state 30 we read the output.

Applying the predication scheme to the balanced example results in a reduction in the number of states. This is achieved by merging both then- and else-branches and by selecting the result of the good computation based on the predicate value. This optimization is ideal for HLS tools because decreasing the number of states reduces the total area required to implement the function. For the examples used in this section, a reduction of nine states was possible, i.e. when comparing (1) and (4) with (2) and (5) respectively. However, because branches can be unbalanced, merging them can have a negative impact on performance when the shorter one is taken. For example in column (5) PRED_U, when the else-path is taken, states 13 to 21 are superfluous and introduce a slowdown for the overall function execution.

Fig. 6 shows all possible paths for both examples as well as their execution times in number of cycles, e.g. from state 1 to state 23. TE represents the execution **T**ime for the **E**lse path while TT is the **T**ime when the **T**hen path is taken. The upper part of the figure corresponds to the balanced if-function and the lower for the unbalanced case. Furthermore, there is a one-to-one correspondence between the columns in Fig. 5 and the scenarios in Fig. 6. The numbers on the edges represent the number of cycles

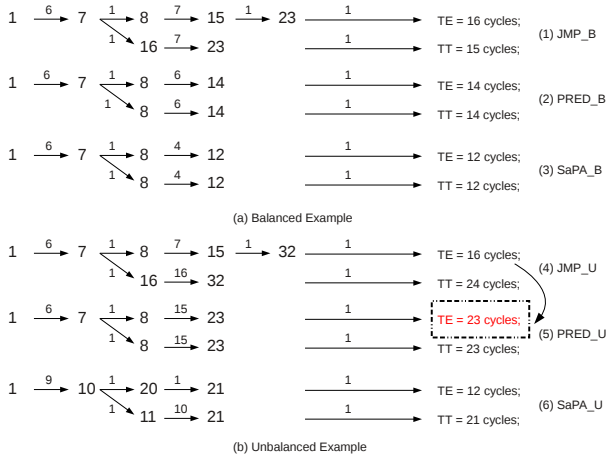


Fig. 6. Execution Sequence of FSM States.

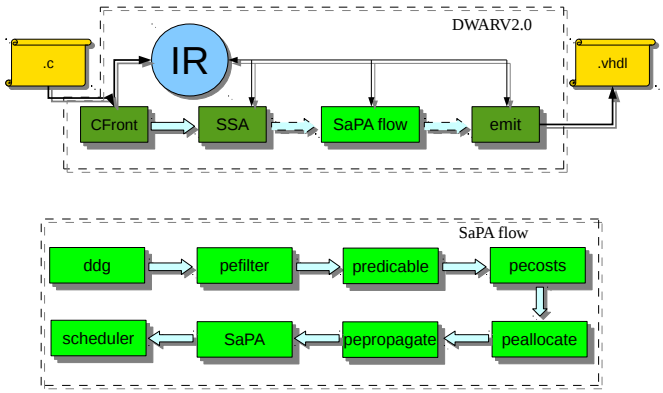


Fig. 7. Engine Flow to Implement SaPA.

needed to reach the next state shown. The last arrow in the paths represents the cycle required to execute the return statement in the last state of the FSM. Considering first the balanced flows, we observe that the predication scheme improves performance compared to the jump scheme (i.e. (2) is better than (1)). This is because jump instructions are removed.

However, care has to be taken to avoid performance degradation when shorter paths are taken. This is shown in Fig. 6 (5) compared to (4), where the execution time increased from 16 to 23 cycles. Therefore, the predication scheme has to be adjusted for hardware execution to cope with unbalanced branches. This is described next.

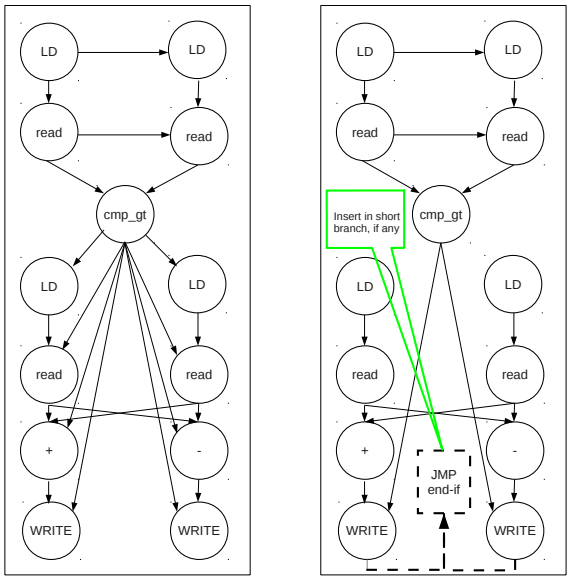
III-B. Algorithm Description and Implementation

To alleviate the short branch problem from the PRED scheme we need to introduce a jump statement when the shorter branch is finished. Fortunately, for hardware execution this is possible without any penalty in cycles as opposed to conventional processors. This extension to the predicated scheme is shown in state S10 of Fig. 5 (6). Including jump instructions in the FSM whenever a shorter path has finished guarantees that no extra cycles are wasted on instructions that are superfluous for the path taken. This is possible because in hardware execution there is no penalty when performing jumps. This motivates that this transformation can always be applied for hardware generation because a hardware kernel is always seen as running on an n-issue slot processor with a jump penalty equal to 0. Furthermore, the flows in (3) and (6) of Fig. 6 show that speculation improves performance even more by starting the branch operations before the predicate is evaluated. It is important to note that speculation in the case of hardware execution comes without any penalty as we do not have to roll back if the predicate value did not select the proper branch for execution. In hardware we have enough resources to accommodate speculative operations, i.e., sacrifice area, in favour of improving performance.

The compiler modifications required to implement the algorithm are shown in Fig. 7 in the lower dashed rectangle. In the upper part of the figure the global flow of the DWARV 2.0 compiler [1] is shown, where standard and custom engines performing various transformations and optimizations are called sequentially to perform the code translation from C to VHDL. In this existing flow, the *SaPA flow* wrapper engine was added. This wrapper engines is composed of seven standard CoSy engines and one custom engine that implements the SaPA algorithm.

The first engine required is the data dependency graph (*ddg*) engine, which places dependencies between the predicate evaluation node and all subsequent nodes found in both branches of the if-statement. Next, the *pefilter* engine is called to construct if-then-else tree structures. That is, basic blocks containing *goto* information coming from an if-statement are included in the structure, however, basic blocks with goto information coming from a loop are not. The *predicable* engine annotates the Intermediate Representation (IR) with information about which basic blocks can be predicated. The compiler writer can also express in this engine if he does not want the if-construct to be predicated. *pecosts* computes the cost for each branch of the if-statement based on the number and type of statements found in these and decides what scheme should be used to implement this if-statement. For hardware generation, this engine was reduced to simply returning SaPA. *peallocate* allocates registers in which if conditions are stored, whereas *pepropagate* propagates those registers to instructions found in both if-branches.

The SaPA engine implements the lightweight predication by introducing a jump instruction in the case of unbalanced branches. That is, when one of the branches reached the end. Whenever this point is reached, a jump to the end of the other branch is inserted. The *SaPA* engine performs this step. Furthermore, the control flow edges from the predicate register to the expressions found in both branches are also



(a) Predicated Graph (b) SaPA Graph

Fig. 8. Data Dependency Graphs.

removed here. That is, for simple expressions with local scope, dependency edges coming from the if-predicate are not needed. These expressions can be evaluated as soon as their input data is available. We name this lightweight speculation because by removing the control flow dependencies from the if-predicate we enable speculation, however, we do not introduce any code to perform the roll back in case the wrong branch was taken as this is not necessary in our hardware design. The dependencies to the memory writes however remain untouched to ensure correct execution. Fig. 8(b) exemplifies the removal of unnecessary dependency edges for the balanced case study. It shows as well in the dashed box the (optional) insertion of the jump instruction, that in the case illustrated, was not necessary.

Finally, the scheduler is executed which can schedule the local operations before the if-predicate is evaluated, i.e. speculating. Furthermore, when the FSM is constructed, whenever the branches become unbalanced, a conditional jump instruction is scheduled to enforce the SaPA behaviour. If the predicate of the jump instruction is true, the FSM will jump to the end of the if-block, therefore avoiding extra cycles to be wasted if the shorter branch was taken. This ensures no performance degradation is possible with this scheme. If the predicate is false, the default execution to move to the next state is followed.

IV. EXPERIMENTAL RESULTS

The environment used for the experiments is composed of three main parts: i) The C-to-VHDL DWARV 2.0 compiler [1], extended with the flow presented in Section III, ii) the Xilinx ISE 12.2 synthesis tools, and iii) the Xilinx Virtex5 ML510 development board. This board contains a Virtex 5 xc5vfx130t FPGA consisting of 20,480 slices and 2 PowerPC processors. To test the performance of the algorithm presented, we used seven functions. Two are the simple synthetic cases introduced in the previous section, while the other five were extracted from a library of real world applications. These applications contain both balanced and unbalanced if-branches.

Fig. 9 show the speedups of the PRED and SaPA schemes compared to the JMP scheme. The *balanced* function shows how much speedup is gained by combining the predication scheme with speculation. Similar to this, the speedup of the *unbalanced* function, tested with inputs that selects the longer branch (LBT), shows a performance improvement compared to the JMP scheme due to speculation. However, when the shorter branch is taken (SBT), the PRED scheme suffers from performance degradation. Applying the SaPA scheme in this case allows the FSM to jump when the shorter branch finishes and therefore obtaining the 1.14x speedup.

The execution times for both schemes for the *gcd* function are the same because both paths found in this arithmetic function have length one, i.e. they perform only one subtraction each. Therefore, the only benefit is derived from removing the jump-instruction following the operands comparison. It is important to note that applying speculation is useful in all cases where both branches and the if-predicate computation take more than one cycle. Otherwise, the PRED scheme is enough to obtain the maximum speedup. Nevertheless, the speedup that can obtained by simply predicating the if-statement and thus saving one jump instruction per iteration can be considerable, e.g. 20% when the gcd input numbers are 12365400 and 906. *mergesort* provided a test case with a balanced if-structure where each of the paths contains more than one instruction. Therefore, the benefit of applying SaPA was greater than using the PRED scheme. This example confirms that whenever the paths are balanced, the application can not be slowed-down.

Finally, the last three cases show results obtained for unbalanced cases with inputs that trigger the shorter branches in these examples. As a result, for all these functions the PRED scheme generates hardware that performs worse than the simple JMP strategy. Applying the SaPA algorithm and introducing a jump instruction after the short branch will allow the FSM to break from the ”predication mode” execution of the if-statement and continue further with executing useful instructions.

To verify that the presented algorithm does not introduce a degradation of other design parameters, i.e. area and frequency, we synthesized all test cases using the environment described in the beginning of this section. Table I summa-

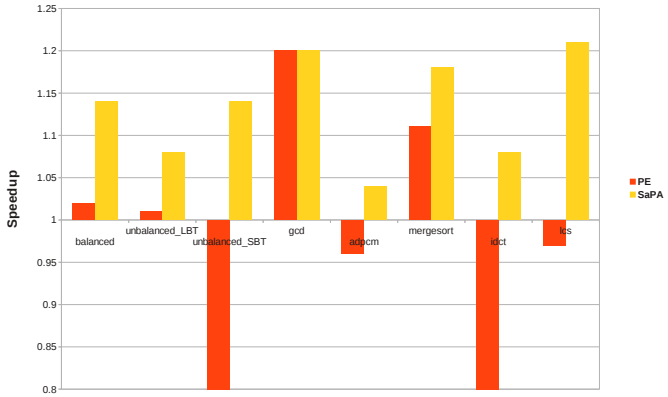


Fig. 9. Predication and SaPA speedups vs. JMP Scheme.

Table I. Implementation metrics for the different schemes.

Function	Scheme	FSM states	Area (slices)	Freq. (MHz)
balanced	JMP	31	258	644
	PRED	20	461	644
	SaPA	16	411	644
unbalanced	JMP	61	780	644
	PRED	50	910	629
	SaPA	46	855	335
gcd	JMP	16	161	350
	PRED	14	153	357
	SaPA	14	153	357
adpcm	JMP	113	1409	328
	PRED	100	1470	328
	SaPA	95	1424	352
mergesort	JMP	102	726	324
	PRED	73	666	304
	SaPA	69	740	360
idct	JMP	240	2361	211
	PRED	162	2048	211
	SaPA	151	2409	211
lcs	JMP	76	748	329
	PRED	52	740	300
	SaPA	49	786	300

rizes the outcomes for the SaPA, PRED as well as the base JMP scheme. Column three list the number of FSM states needed to implement the corresponding scheme from column two. Column four shows how much the complete generated design, i.e. FSM with data-path, took in actual FPGA slices for the target device. Finally, the estimated frequency is reported in column five. Studying the numbers, we can observe that the area does not increase nor does the frequency decrease substantially when we combine both paths of if-statements. Our experimental results therefore support the claim that SaPA brings additional performance improvement while not substantially increasing the area nor negatively affecting the execution frequency. Nevertheless, future research is necessary to investigate the impact in terms of frequency and area for a large number of test cases.

V. CONCLUSION

In this paper, we argued that the typical JMP and PRED schemes found in conventional processors are not performing ideally when we consider hardware execution. The problem with the first is that we loose important cycles to jump from the state where the if-condition is evaluated to the correct branch state corresponding to the path chosen for execution. The PRED scheme solves this issue, however, it suffers from performance degradation when the if-branches are unbalanced.

To combine the advantages of both schemes, we presented a lightweight speculative and predicative algorithm which does predication in the normal way; however, it introduces a jump instruction at the end of the shorter if-branch to cope with the unbalanced situation. Furthermore, to leverage the hardware parallelism and the abundant resources, SaPA also performs partial speculation, i.e., no roll back necessary. This gains extra cycles in performance when the if-predicates take more than one cycle to evaluate.

We demonstrated based on two synthetic examples the benefit of this optimization and we validated it using five real world functions. The results show that performance degradation will not occur anymore for unbalanced if-statements and that the observed speedups range from 4% and 21%. SaPA is therefore a transformation that should be part of any HLS tool. Future research will analyse the impact of SaPA on a large number of kernels and investigate if any systematic relation can be observed between the input parameters (e.g. number of instructions per path or nested ifs) and the measured hardware metrics (i.e. area, frequency).

ACKNOWLEDGEMENT

This research is partially supported by the Artemisia iFEST project (grant 100203), the Artemisia SMECY project (grant 100230), and the FP7 Reflect project (grant 248976).

VI. REFERENCES

[1] R. Nane, V.M. Sima, B. Olivier, R. Meeuws, Y. Yankova and K. Bertels. *DWARV 2.0: A CoSy-based C-to-VHDL Hardware Compiler*. To Appear in Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL '12).

[2] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov and E. M. Panainte. *The molen polymorphic processor*. In IEEE Transactions on Computers(November 2004). pages: 1363-1375.

[3] S. A. Mahlke and D. C. Lin and W. Y. Chen and R. E. Hank and R. A. Bringmann, *Effective compiler support for predicated execution using the hyperblock* , 25th Annual International Symposium on Microarchitecture,1992

[4] R. Leupers, *Exploiting conditional instructions in code generation for embedded VLIW processors*, Proceedings of the conference on Design, automation and test in Europe (DATE), 1999

[5] M. Hohenauer, F. Engel, R. Leupers, G. Ascheid, H. Meyr and G. Bette, *Retargetable Code Optimization for Predicated Execution*, in Proceedings of the conference on Design, automation and test in Europe 2008

[6] Altium Designer 10. [Online]. Available: http://www.altium.com/

[7] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown and T. Czajkowski. *LegUp: high-level synthesis for FPGA-based processor/accelerator systems*. (FPGA'11). pages: 33–36.