Combining MBP-Speculative Computation and Loop Pipelining in High-Level Synthesis

U. Holtmann, R. Ernst
Technical University of Braunschweig
Braunschweig, Germany

Abstract

Frequent control dependencies caused by IF- and loop-statements limit the parallelism usable in High-Level Synthesis. Loop pipelining is a powerful way to increase parallelism, but is often limited by these control dependencies. Multiple branch prediction (MBP-SC) applies loop pipelining and speculative computation to the most probable path and serves other paths during the restore phase (prediction error correction).

In this paper we combine MBP-SC and loop pipelining and give a scheduling algorithm. Further MBP-SC improvement comes from parallel branch execution.

The results show a considerable speedup compared to previous approaches.

1 Introduction

The hardware/software cosynthesis system COSYMA [1] takes C programs from the embedded control domain as input to speed them up on a combination of a programmable processor (e.g. SPARC) and an application specific hard-wired coprocessor. Analysis of input programs, partitioning into SW and HW, as well as generation of the coprocessor with high-level synthesis are done automatically. In this paper we focus on the high-level synthesis part in COSYMA. Inputs are small parts of C programs, typically loops [2]. In the examples, which we took from different areas, "IF"-statements and loops with data dependent number of iterations occur frequently. These control dependencies seriously limit the potential parallelism.

While early scheduling algorithms focused on the problem to distribute operations within a basic block emphasis has, meanwhile, shifted to scheduling *across* basic-block boundaries. Path-based scheduling ([3], improved in [4, 5]) schedules paths of an execution rather than basic blocks, but it does not solve the problem of control dependencies. Speculative computation (SC) potentially reduces the control dependencies by allowing to execute an operation before it is known to be necessary. It is used in percolation scheduling [6], as part of a global list scheduling [7, 8], or in other scheduling approaches [9, 10].

None of the known approaches uses the full potential of SC in the context of loops, which are typically most critical to circuit speedups. In [11] multiple branch prediction and error correction (MBP-SC) is employed as SC-technique with low circuit overhead. Given a loop with profiling information, MBP-SC predicts the path through the loop body with the highest probability to be taken. This path is predicted and scheduled using loop pipelining (LoopP) [12], whereby operations belonging to other paths are deferred. If this path is predicted incorrectly, execution switches to a *restore phase* (prediction error correction). The body itself is always predicted to continue rather than being terminated. HW overhead due to prediction error correction remains low [11].

In this paper we also combine MBP-SC with earlier techniques of parallel path execution [9] and apply it to LoopP. Compared to earlier SC-techniques, it can also reduce memory access. Furthermore, we present the scheduling algorithm MBP-SC which was previously done manually.

The rest of the paper is structured as follows. Chapter 2 explains MBP-SC using an example and chapter 3 gives the exact definition and scheduling approach. Results for practical examples are given in chapter 4.

2 Approach & Example

MBP-SC scheduling is explained using the printer context example in figure 1, given in a C-like description. $x[\,]$ contains a sparse data structure with 16-bit records describing the code and position of characters on a single printer row. There are two fields with variable length, the first given in a Huffman code with

```
  1  a=0; i=0;
100  while (a < limit_a) {
 99    switch (x[i] & 0xc000) {
 50      case (0x0000):  /* 00 */
           a += x[i] & 0x3fff;
           break;
  9      case (0x4000):  /* 01 */
           a += (x[i] & 0x3fff)
               +(x[++i] << 14);
           break;
 40      default:        /* 1- */
           /* skip */;       }
 99    i++;
     }
```

Figure 1: a typical input program in C

values “1”, “00” and “01”. If the first bit (bit 15) is “1”, the following lower 15 bit form a character to be printed at the actual position. With “00”, the position is moved to the right by a distance given by a 14-bit value. Larger moves are indicated with he code “01”. Then, all bits of the following word are concatenated with the 14 bit data field forming a 30-bit value. The example is used in a clipping algorithm to find the position of the last character before a given limit. Inputs are $x[]$ and $limit\_a$, return values are i and a. For a given input pattern, profiling provides the data in the first column. The loop is started only once, performs 99 iterations (50 times “0x0000”, 9 times “0x4000”, 40 times “default”) and then ends.

> **Definition:** A *conditional precedence* (CP) with boolean expression v, pointing from operation A to B allows the execution of B only if the outcome of A matches v.

The CDFG is shown in figure 2. The while-condition is described by the two small multiplexers on bottom and all CP's pointing from the comparison “<”. If it is false (value “0”), no operations are executed and the loop terminates. The 3-input multiplexers and all remaining CP's represent the “switch” statement. The value of statement “case (0x4000)” is here written as “01”. Value “1-” represents the “default” statement and is the complement of “01” and “00”. Node links are used to describe a constant shift and the extension of bit-widths from 16 to 32 bit. They only restructure wires and need no execution time. Access to array $x[i]$ is done by a RAM access with operation

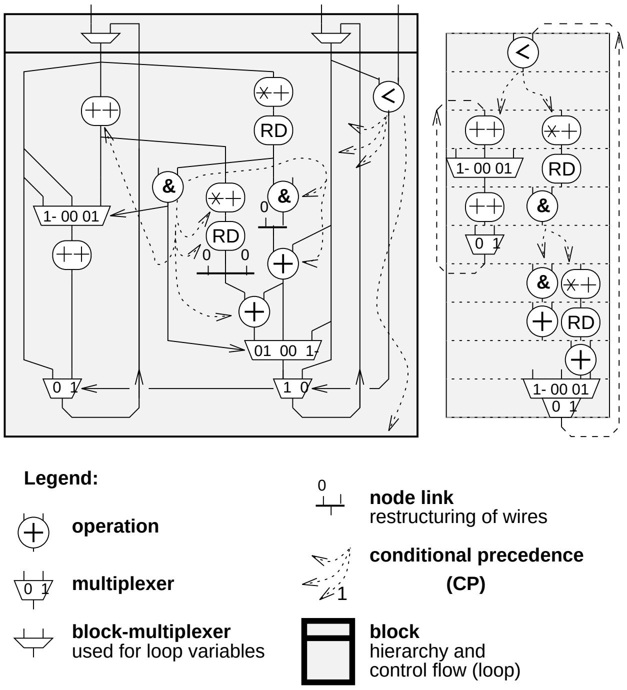

Figure 2: CDFG and schedule of the example

“RD” and preceding address calculation with multiply/accumulate operation “*+”. If scheduled without any SC, the schedule given on the right hand of figure 2 may result. All CP's are regarded. We use a pipelined (and therefore fast) controller with a delay of one clock cycle. The schedule needs a long time (10 clocks) for a complete iteration and, what is more disturbing, LoopP is not possible due to a control dependency.

Now, we make use of MBP-SC to speed-up the loop. As in [11] we may predict the most probable path (that one with the highest probability to be taken). It goes through the statement “case (0x0000)” and is taken in 50 iterations (prediction accuracy $p = 0.5$). As long as this prediction is correct, a new iteration may be started every clock cycle and yields a speed-up of 10 compared to the schedule without SC. From [11] we know that the necessary *restore phase* decreases the speed-up. Here, the probability for a prediction error correction is very high ($p = 0.5$) and significantly limits the potential speed-up from 10 ($p = 1$) to 2.778.

That was the result of our previous work [11]. In the following we give an improvement and for the first time present a suitable scheduling algorithm.

The idea of multiple branch prediction, as used in [11], was to predict several branches in sequence in order to exploit maximum concurrency along a *single* path. In other words, we formed chains of predictions c_i

$$c_i = \bigwedge_{j=1}^{n_i} a_j, \qquad a_j \in A,$$

where A is the set of possible predictions on conditional control operations (branches, loops, ...). Any c_i defines a path through the program. In this paper, we introduce sets of predictions CS_i corresponding to sets of paths, which can be predicted together

$$CS_i = \bigvee_{j=1}^{m_i} c_j.$$

This way, the prediction accuracy increases to

$$p(CS_i) = \sum_{j=1}^{m_i} p(c_j).$$

In our example, the accuracy of the prediction set CS_1\{"00","1-"\} is $(50 + 40)/99 = 0.91$. CS_i may, now, contain contradictory predictions.

Some of the operations may have to be executed on all predicted paths or, if not, they might at least not lead to incorrect results on other paths. Only operations where this does not hold must not be executed or must be corrected. Any set of predictions CS_i defines a 3-way-partition on the CP's into those CP's where the boolean expression is true, $CS_{i,T}$, those where it is false, $CS_{i,F}$ and all other CP's, where the prediction is not sufficient to decide on the value, $CS_{i,U}$.

Now we will compute the corresponding schedule. In the beginning all conditions are predicted according to CS_1 =\{"00","1-"\}. Due to the predictions, the CDFG may be simplified, or *"restricted"* as we will say in the sequel. All operations with a $cp \in CS_{1,F}$ pointing to them are removed from the CDFG. The result is the CDFG **B** in figure 3. The upper "++" operation is removed together with the CP pointing towards it, because the condition "01" is not valid with respect to CS_1 as well as the operations "*+", "RD" and "+" on the left. The multiplexer inputs "01" are removed. If a prediction error occurs and MUX input "01" turns out to be correct, the multiplexer outcome will no longer be valid. Therefore, its *"scope"* is restricted to the predicted alternative CS_1. The scope of any operation which reads the multiplexer output becomes also restricted. *The scope is forwarded through data dependencies as far as possible.* As a result, the scope of the remaining increment is also restricted. The two inputs of the upper multiplexer read the same operation and are reduced to a single one. Because only one input remains, the complete multiplexer is removed from the CDFG and its (data) input and output are connected directly. This also happens to the 2-input multiplexers after removal of input "0". All remaining CP's are $cp \in CS_{1,T}$, i.e. always true with respect to CS_1 and are *removed.* $cp \in CS_{1,U}$ stay in the CDFG but with modified boolean expression $f|_{CS_1}$. The remaining CDFG is scheduled using LoopP and yields a latency of two clock cycles.

As long as no prediction error occurs, the next iteration is started every two clock cycles. This happens with the probability of $99/100 * 90/99 = 0.9$. That is the second main advantage of MBP-SC: *LoopP can be applied to the predicted sets of program paths.* The first prediction error may be detected after the fifth clock cycle (operation "&" is executed during the third clock cycle, but the controller delay adds two additional cycles). If "01" is true, a prediction error occurs. Then, two things happen. First, the knowledge about conditions increases: condition "&" is now *known* to be "01". Second, the executed CDFG/schedule is no longer sufficient because it was restricted to CS_1. A new CDFG must be determined according to known conditions and previously executed schedule(s). It is given with CDFG **C** in figure 3. There, the "<" and other operations are removed because they were already executed in schedule **B**. The lower increment is not removed because its scope in schedule **B** is not sufficient. The remaining CDFG elements are scheduled without LoopP, because we assume CS_1 to be true again in the next iteration. The small "W" operations write the correct values of variables i and a into that registers where schedule **B** assumes them to be.

Only the first clock cycle of schedule **C** is executed. Then, the controller responds to the next condition ("<" from schedule **B**). It is a concept of MBP-SC to *fork within prediction error correction every time when a new condition is computed* and a prediction error may arise. An particular CDFG and schedule is computed for every branch, and will, itself, fork until all conditions are known. Here, schedule **C** forks into schedules **D** (with condition "<" known to be "1") und **E** ("<" known "0"). Schedule **D** needs three clock cycles and then jumps back to the beginning schedule **B**. The loop is restarted.

The other branch, **E**, is taken when the loop terminates ("<"="0"). Although its CDFG contains no elements, the schedule needs one clock cycle to write the result values a and i to registers. This is necessary, because the loop may terminate within schedules **E** or **F** (see below), and as a consequence result values may be found at different locations. With such operations *MBP-SC can guaranty unique register locations for result values after the loop has finished.* So, conditions from inside the loop are not necessary to be known outside and controller overhead remains low.

The last CDFG, **F**, is used when the loop terminates after a correctly predicted "&".

The average number of cycles per iteration is 2.68. If only the most probable path is predicted with CS_1, it increases to 3.78. Please note, that a large CS does not always yields the best results. If we use $CS = \bigvee_{all} c_i$, we need 3.06 cycles per iteration! The reason are "disturbing" alternatives which have a low probability to be taken and increase the latency because they require HW or enlarge the critical path.

Access to RAM is typically a bottleneck which can't be removed by more HW. *MBP-SC is able to focus LoopP on those operations which do not disturb the most probable path* and can reduce memory access.

3 Scheduling Algorithm

One key idea of MBP-SC is to restrict the original CDFG to predicted and known conditions and then apply a conventional scheduling algorithm to it, for example FDLS. Restriction of CDFG is done according to rules 0...8 (see below).

CS_i, c_i and "scope" are boolean expressions with respect to output signals of conditions. For clarity, we use terms used in logic optimization. A condition cd with n bits has 2^n possible values, given in the set V_{cd}. $V = V_1 \times ... \times V_n$ is an n-dimensional cube, formed by all conditions $cd_1, ..., cd_n$ of the CDFG. Any interdependence of predictions, known and unknown conditions and prediction error correction can be derived with simple logic operations.

A condition may have the attribute "unchanged", "predicted" or "known". With "unchanged" no MBP-SC is applied. Such conditions are not considered in V and always remain "unchanged". All other conditions will be predicted to some alternatives. Before the condition operation is computed, the condition is "predicted", after it is "known". Controller-delay is considered.

Let cp be a CP with expression v, pointing from operation a to b. m denotes a multiplexer controlled by a. The *scope* of an operation op, $scope(op) \subseteq \bigcup CS$, describes when the outcome of op is valid.

Start: take the original CDFG (where no elements have been removed). Add operations from previous iterations if they are not yet finished. Set the scope of any operation to V. Now restrict the CDFG to $CS = CS_1$, the initial prediction.

Rule 0: If two inputs of m read the same result of an operation then merge them into one input.

Rule 1: If input i of m is never selected for any value in CS then remove i from m and restrict the scope of m to CS: $scope(m) := scope(m) \cap CS$. The scope is not restricted if CS has a known result (all conditions evaluated).

Rule 2: If there is a data dependency between operations a and b and $scope(a)$ changes, then restrict the scope of b: $scope(b) := scope(b) \cap scope(a)$. The scope of block-multiplexers is never restricted.

Rule 3: If m has only one data-input, then directly connect it with the output and remove m from the CDFG.

Rule 4: If m has no data-inputs, then remove m.

Rule 5: The following table decides what to do with cp and b:

	cd is		
	predicted	known	unchanged
v not true	*1	*1	*3
v true	*4	*2	*3
v unknown	*4	*3	*3

*1 remove cp and b
*2 remove cp
*3 remove nothing
*4 remove cp, if b has no side effect

Term "*v true*" is fulfilled, iff $v \subseteq CS$. Term "*v not true*" is fulfilled, iff $v \cap CS = \emptyset$. In all other cases term "*v unknown*" is fulfilled. For example, suppose CS ={00,01}. Then follows "*v*=1- *is not true*", "00 *is true*" and "-1 *unknown*".

Rule 6: If operation a has no side effect and its outcome is not used by the controller or any other operation, then remove a.

Rule 7: If operation a has already been executed previous to operation a', remove a if $scope(a) \subseteq scope(a')$.

Rule 8: If operation a has a side effect and its scope is restricted then insert CP's into the CDFG according to the restriction. (The cutset of the conditional expressions of these CP's must be identical to $scope(a)$). This rule assures that a is executed only with valid input data.

Rules may be applied several times. Rule 2 has priority to rules 3...8. Rule 8 is applied when no other rule matches. The restriction is complete when no more rule matches.

The overall schedule is constructed beginning with the *periodic schedule* (example: **B**). All conditions have either status "predicted" or "unchanged". Only this schedule may use LoopP and is executed until the first prediction error is detected. If this happens in iteration n, then (1) abort already running iterations $n+1, n+2, ...$, (2) apply prediction error correction (restore phase) to iteration n and then (3) either restart the periodic schedule with iteration $n+1$ or abort the whole loop. For each possible prediction error within the periodic schedule, a separate CDFG and schedule is computed. These schedules are aborted as soon as the next condition changes its status from "predicted" to "known". The control flow forks, regardless whether a prediction error occurs or not (example: this happens after the first clock cycle of schedule **C**).

Average run time, HW amount and controller complexity are effected by the CP-partition. Determination of the optimal 3-way-partition is a combinatorial problem. Due to limited space we will deal with its computation in a later paper. Allocation is performed as in [11] .

4 Results

We applied MBP-SC to several inputs which are the most run time intensive fragments from real-world programs of embedded control tasks, mostly video applications. "quick" is the inner loop of a quicksort algorithm. Lengths vary between 3 and 29 lines of C code (without comments), containing 1 up to 4 loops and 3 IF-statements (average number). HW is limited to 5 ALUs, 1 multiplier, 1 shifter and 1 RAM. Access to RAM and multiplication need two clock cycles each, all other operations need one. Up to five 2-input multiplexers may be chained with another operation within one clock cycle. A pipelined controller is used with the same delay as discussed above in the example.

Program	BBS	BBS +LoopP	**MBP-SC** all	**set**
etik1	16	12.00	2.00	2.00
etik3	12	10.00	6.00	**3.75**
etik4	13	7.00	7.00	**5.02**
blue1	10	3.06	2.03	2.03
blue2	6	6.00	3.00	3.00
blue3	11	11.00	2.80	2.80
blue4	8	3.70	2.20	2.20
smooth	15	8.25	7.75	7.75
quick	14	12.08	5.92	**4.11**

Table 1: Average number of clock cycles per iteration

Program	ALUs	latency	clock cycles total	avg.
gcd	4	1	$n+3$	
counter	4	1		1.00
display	2	1		1.50
display	4	1		1.03
display	8	1		1.00
fancy	4	2	$2n+1$	

Table 2: Some results from the HLS workshop benchmark

Table 1 shows the results. If LoopP is used within BBS (basic block scheduling), the schedules become faster. With use of MBP-SC LoopP is more effective and yields faster results. In column "MBP-SC/all" always all paths are predicted. This corresponds to SC with LoopP without branch prediction. In three cases better result are obtained, if prediction is limited to a subset of paths (column "MBP-SC/set").

Table 2 shows results for some of the programs from the HLS workshop benchmark 91/92. The last two columns give the number of clock cycles necessary for execution of the loops (either for a fixed number of iterations denoted by n or, if the loop never stops, as the average number per iteration). "gcd" and "counter" are simple examples, where fastest results are obtained when all paths are predicted and the correct result is selected by a multiplexer. If HW is restricted to two ALUs for "display", the digit "secs" is predicted to "increment" (and not to "overflow"). With four ALUs, "tsecs" is predicted to "increment" and both paths of "secs" are predicted. With eight ALUs all paths are predicted. Simple optimizations (use common expressions, determine constant expressions outside the loop, simplification of expression "temp6a" and "(a>counter) XOR (a$\leq$counter)") are applied for "fancy" before scheduling. All pathes are predicted.

5 Summary and Conclusions

We have presented the scheduling algorithm of MBP-SC which is used within a HW/SW cosynthesis system to speed-up small fragments of C programs. It combines multiple branch prediction and parallel path execution with loop pipelining. As a main advantage, MBP-SC allows to predict any *set* of possible pathes and thereby enables a tradeoff between branch prediction accuracy and number of speculatively executed operations.

The schedule is computed by restricting the CDFG to predicted and already executed conditions. Thus, management of prediction and error correction is separated from the underlying basic scheduling algorithm. Exact rules are given for this management.

Real-world inputs show that MBP-SC significantly improves loop pipelining and gives better results than simple SC techniques which always predict all paths.

References

[1] R. Ernst, J. Henkel, Th. Benner, "Hardware-Software Cosynthesis for Microcontrollers", *IEEE Design & Test*, Vol. 10, No. 4, pp. 64-75, 1993.

[2] J. Henkel, R.Ernst, U. Holtmann, T. Benner, "Adaption of Partitioning and High-Level Synthesis in Hardware/Software Co-Synthesis", *ICCAD*, pp. 96-100, 1994.

[3] R.A. Bergamaschi, R. Camposano, M. Payer, "Area and Performance Optimization in Path-based Scheduling", *EDAC*, pp. 304-310, 1991.

[4] K. O'Brien, M. Rahmouni, A. Jerraya, "DLS: A Scheduling Algorithm For High-Level Synthesis in VHDL", *EDAC*, pp. 393-397, 1993.

[5] S. Huang, Y. Jeang, et. al., "A Tree-Based Scheduling Algorithm For Control-Dominated Circuits", *DAC*, pp. 578-582, 1993.

[6] R. Potasman, J. Lis, A. Nicolau, D. Gajski, "Percolation Based Synthesis", *DAC*, pp. 444-449, 1990.

[7] P. Gutberlet, W. Rosenstiel, "Scheduling Between Basic Blocks in the CADDY Synthesis System", *EDAC*, pp. 496-500, 1992.

[8] P. Yeung, D. Rees, "Resource Restricted Aggressive Scheduling", *EDAC*, pp. 501-506, 1992.

[9] K. Wakabayashi, H. Tanaka, "Global Scheduling Independent of Control Dependencies Based on Condition Vectors", *DAC*, pp. 112-115, 1992.

[10] U. Prabhu, B. Pangrle, "Global Mobility based Scheduling", *ICCD*, pp. 370-373, 1993.

[11] U. Holtmann, R. Ernst, "Experiments with Low-Level Speculative Computation Based on Multiple Branch Prediction", *IEEE Transactions VLSI*, Vol. 1, No. 3, pp. 262-267, 1993.

[12] G. Goossens, J. Vandewalle, H. DeMan, "Loop optimization in register-transfer scheduling for DSP-systems", *DAC*, pp. 826-831, 1989.

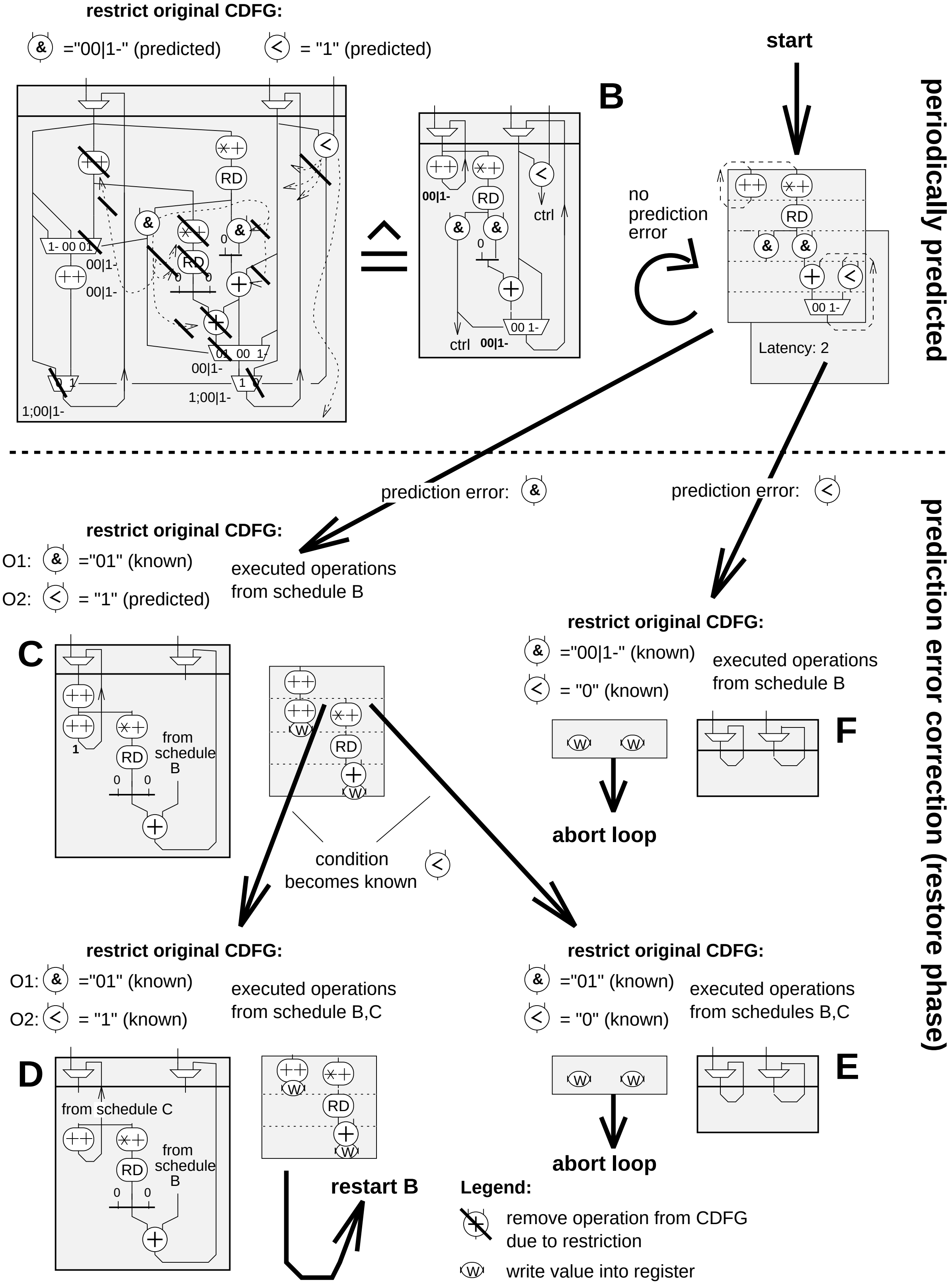

Figure 3: Schedule of the example obtained by use of MBP-SC