U. Holtmann, R. Ernst: Some Experiments with Low-Level Speculative Computation Based on Multiple Branch Predictions, Laguna Miguel, Kalifornien, 4.-6.11.1992.

Some Experiments with Low-Level Speculative Computation based on Multiple Branch Prediction for the Synthesis of High-Performance, Pipelined Coprocessors

Ulrich Holtmann

Dep. of CS
Abteilung Entwurf
integrierter Schaltungen

Rolf Ernst

Dep. of EE
Institut für Daten-
verarbeitungsanlagen

Technische Universität Braunschweig
W-3300 Braunschweig
GERMANY

Fax: (+49)531 391-5840
e-mail: holtmann@eis.cs.tu-bs.de

Abstract

Coprocessor design is one application of high-level synthesis. We want to focus on high-performance coprocessors to speed up time critical parts in hardware-software codesign of embedded controllers [16].

Time critical software parts often contain nested loops, often with data dependent branches and data dependent number of iterations. When (loop) pipelining is employed for high performance, the control dependencies become a dominant limitation to pipeline utilization. Branch prediction is a possible approach, but is usually restricted to few instructions and to one branch because of hardware and control overhead. Multiple branch prediction and speculative computation take a more global view on the program flow. We give practical examples of how speculative computation with multiple branch prediction increases performance far beyond an usual ASAP scheduling based on a CDFG. For scheduling, speculative computation requires a modification of the CDFG and, for the allocation phase, the insertion of register sets to save the processor status. The controller needs slight modification.

We conclude that manual application of our approach will in general be too difficult, such that it can only be used in connection with synthesis.

Introduction and Related Work

Coprocessor design is one application of high-level synthesis. We want to focus on high-performance coprocessors to speed up time critical parts in hardware-software codesign of embedded controllers [16]. Time critical software parts often contain nested loops, often with data dependent branches and a data dependent number of iterations. In a "blue screen" algorithm for the digital HDTV television, e.g., 90% of the run time is spent in

only 10 lines (i.e. 1% of the program), which contain 2 loops, 2 IF statements and 16-bit integer arithmetic [15]. It is only because of these loops that an expensive 32 bit RISC processor must be used to even come close to the time constraints.

The coprocessors we want to synthesize are targeted to speed up such time critical program sections ("point accelerators") and not to speed up a specific class of operations such as a FPU. We primarily aim at maximum performance for the critical parts. The algorithms to partition the software and hardware components are not part of this paper. We have analyzed around 20 programs implementing small embedded control tasks [1] and found that the program segments for a high performance coprocessor are small program sections (10..100 lines of C), which contain 2 to 5 nested loops with several IF statements, integer or floating point arithmetics and data arrays of small to large size (10 ... 100.000 Bytes). So internal registers as well as off-chip memory access are necessary.

The scheduling of pipelined loops is well understood [2,12] and yields the necessary high speed. We found, however, that in many cases the decision to end the loop or to execute another iteration is derived dynamically *inside* the loops and is not known at compile time. Known pipeline scheduling approaches fail to fill the pipeline when such data dependencies are involved.

If we still want to fold the loops with maximum density of up to one iteration per cycle, it is necessary to begin the execution of the next iteration before the evaluation of the loop condition. This is an application of speculative computation. *Speculative computation* means that tasks are executed before this is known to be necessary whenever no other task is ready for execution [17]. *Branch prediction* is one particular case of low-level speculative computation. Branch prediction means that, at a branch instruction, the target is predicted whenever it is yet unknown. A branch prediction must never influence the correct execution of an algorithm. So, when a prediction error occurs, all effects of operations, which were initiated as a result of the prediction, must be removed.

Branch prediction is well known in pipeline processor design. Techniques were developed to predict branches with a hit rate above 80% [3,4,5]. The pipeline control point is that pipeline stage, from which on an instruction can irreversibly change the processor state [6]. A late control point can be used for branch prediction. As soon as a value is written to a register or memory, the control point is reached, except when the previous value had been saved and can be restored in case of a prediction error. Hardware solutions exist, where a special register pool – a "history buffer" – stores the new register values until the branch target is calculated [9,10]. Such solutions have already been used in industrial designs [19].

Multiple branch prediction goes even further. (The "prediction life time" is the time between the beginning of a branch prediction and the final calculation of the real branch target. We speak of *multiple branch prediction*, if the prediction lifetimes of sequent branches are overlapping.) It allows the prediction and execution of additional branches before the first branch prediction has been evaluated. Multiple branch prediction has been introduced in [20] and has shown to be very efficient, in particular for deep pipelining [7].

While for processor design, the circuit and interconnect overhead is considerable [19], already for single branch prediction, the overhead in application specific hardware can be very small, even for multiple branch prediction, as we will show in the sequel. While "history buffers" in processors are only targeted to a single central register file, we can even handle decentralized special purpose registers which are necessary for efficient high level synthesis. This is possible by assigning individual "history" register sets to each special purpose register and adapting their size to the number of predicts for which a variable must be saved.

So far the opportunities of a late control point are not exploited in synthesis. Path based scheduling [13] is a way to consider basic blocks and branches together, but the control dependencies are not changed. Percolation scheduling [14] is able to fold sequential test conditions and to execute them in parallel, similar to multi-way independent branching [18], but this scheduling technique is limited to *test* conditions and has no effect on the control point.

The Examples

We will apply speculative computation to four examples. The first one, exam, has been constructed just for the demonstrating of speculative computation with multiple branch prediction. Program blue1 is that part of an HDTV chromakey algorithm. Applying normal pipelining to it leads to an optimum pipeline latency of 3 clock cycles. Speculative computation then compresses the pipeline latency to only 2 clocks. This is only a small improvement because the algorithm contains only few control dependencies and therefore normal pipelining is already successful. Another part of the same program, blue2, has more control dependencies but is less often executed. Here speculative computation leads to a speed up of 3.

The third example, quick, is the inner part of a quicksort algorithm. This is a hard example, because the probability for a correct branch prediction is only 0.67. But in spite of this effect the overall speed up is nearly 3.

Example 1: exam

Figure 1 shows the given example exam. The main part is a loop with two different termination conditions. Each iteration reads a value from array tab and adds it to sum. The first condition is the limitation of index i to l_i and value tab[i] to l_val. The second condition arises when sum reaches the limit l_sum. After the loop has finished, the values of i, val and sum are used. The number of iterations is clearly data dependent.

In figure 2 the control/data flow graph is shown. It is an HBR graph, the CDFG we use in our high-level synthesis system BSS [11]. The large "break"-box represents the loop. The circles and ovals inside the loop are *actions*, e.g. an increment or the read-access to array tab. The two multiplexers in the above part of the box represent the variables i and sum. They change their outputs only in the beginning of each iteration. The curved lines are control dependencies which have an effect on the order of actions. (Please, note that tab[i] is read only once and stored in register val.)

To simplify the example we assume that every action needs 1 clock cycle to execute (not a limitation to our approach). Because we want to use a clock with high speed, the controller is assumed to be pipelined, too, and needs some time to respond to the signals from the data path. Here, we assume a delay of one clock cycle. This means, e.g., if the condition of a branch is known at the end of clock cycle 1 the first clock cycle processing the branch target is 3. This is realistic in processors with a clock period in excess of 50 MHz.

Figure 3 shows the best possible schedule (ASAP), if no speculative computation is permitted. No hardware constraints are given, the order of execution is only limited by data and control dependencies. Statement "sum += tab[i]" is only reached when the termination condition 1 is fulfilled. Therefore it must be tested (executed) before the addition may start (see the curved lines in figure 2 and 3a). The controller delay occupies one further clock cycle, so the following operation cannot be executed before cycle 3. (the small boxes "1" and "2" in figure 3a indicate the point in time, when the controller has reacted to the condition). The increment of the index must be placed in clock cycle 6, because the statement is not reached, if condition 2 is fulfilled. Although it is already known in clock cycle 6 that the next iteration will be executed this iteration must wait until the end of clock cycle 6 because there is a data dependency between the increment (actual iteration, clock 6) and the read (next iteration, clock 0). So no pipelining is possible (loop unfolding [3] could save at least 1 cycle). Using this schedule a new iteration can be started only every 7 clock cycles.

Now we use speculative computation with multiple branch prediction and ignore the three control dependencies. We get the schedule in figure 3b. Each iteration needs four clock cycles, but a new iteration is started *every* clock cycle. The high speed up is achieved by the dense folding of the pipeline. (See figure 4 how the four pipeline stages are working in parallel.) As long as there is no break, this new schedule is 7 times faster. It is essential to note that this improvement is only possible because the schedule ignores the control dependencies due to the two termination conditions.

The decision to execute the next iteration or to end the loop is a branch. We predict the branch because the following actions are executed before the branch condition is reached. We always predict the jump to the top of the loop, which is the best static prediction for loops [3,4,7] shortening the critical path.

What will happen, if, e.g., condition 2 ends the loop within iteration no. n? Because the next iterations are already initiated, we get the situation shown in figure 4. All shaded actions have been executed "speculatively" and their effects must now be made undone. Because the loop breaks within iteration n, all actions of following iterations are initiated illegally. The increment of iteration n is also illegal because of a control dependency (see CDFG in figure 2). Our primary goal is not to really make the speculative actions undone but to restore the status at the end of iteration n. We do this by simply extending all registers to register sets. From a trace back, as shown in figure 4, the number of illegal actions due to a prediction error is known. For each action the maximum number of illegal actions is easily determined. The result of an action, if any, is stored in a register. If there are at most i illegal actions, it is necessary to store at most i previous values. The register becomes a register set with at most i+1 entries. To "undo" the i actions means to replace the actual value with the one written i times before. In this example the register sets of i, val and sum need 5, 4 and 3 entries, respective.

A register set acts as a ring buffer where the new value is always written to the top. Figure 5a shows a possible implementation with (expensive) dynamic edge-triggered flip-flops. Another implementation (5b) uses (cheaper) transparent latches which are written in a cyclic order controlled by a modulo counter.

After loop termination the correct status of the data path must be found during a "restore phase". In both cases (of fig. 5) the restore phase is a simple change of the counter value (i.e. the buffer pointer). In both cases nothing else needs to be done.

Figure 6 shows the final net list. The shaded parts of the register sets for i, val and sum are those registers which are only necessary for speculative computation.

Example 2: key1

The most frequently executed part of the chromakey algorithm mentioned above is "key1" which is shown in figure 7. Because it depends on two nearly identical loops, only the first of them is explained in the following. The other has always the same schedule and same results.

The loop searches for a minimum value in a key table and reads a value from the array "vtab" in each iteration. Because vtab is a larger table and, therefore cannot be an on-chip register, a read access to the external memory/cache is necessary. Assuming a 20ns clock cycle this read access will need 2 clock cycles. The function "abs", returning the absolute value of an integer, is performed by a conventional ALU with a following multiplexer. The IF and the assignment "iabsv=ihilf" can be collapsed to a comparison and a multiplexer. Their execution, therefore, needs only one clock. In the diagrams only the comparison "<" is shown.

Normal optimum pipelining leads to a pipeline latency of 3 clocks (see figure 8a). Although it is already known in clock cycle 2 that the next iteration will follow, this iteration has to wait until the index is incremented. Application of speculative computation compresses the pipeline from 3 to 2 clocks (fig. 8b).

Figure 9 shows that three actions are illegal when the loop breaks (terminates). But, these values are not used outside the loop, so it is not necessary to add any register. In general, this means, whenever a value is not used outside the loop, no extension to a register set is necessary. This can reduce the overhead.

Example 3: key2

Figure 10 shows the segment key2 which is another part of the chromakey algorithm. In every iteration the loop reads a value from vtab (mapped onto the external RAM) every iteration and tests it. As long as the while-condition is fulfilled, the index is decremented. The "or" of the test is chained with comparison operations and needs no additional clock (figure 11). To save the correct values for the restore phase, register set cr needs 3 entries and tabl 2 (see figure 12).

Example 4: Quicksort

Figure 13 shows a quicksort algorithm. The darker part is the inner loop which is synthesized. Because the THEN- and ELSE-part belonging to "IF (up)" condition are very similar, only the THEN-part is shown in the diagrams to simplify the figures. It is not necessary to *predict* the IF conditions because the criterion is already calculated. The remaining difficult branches are no. 1 which breaks the loop and no. 2, an IF statement. While branch 1 can be predicted with a very high accuracy of 99%, the IF is predicted to "FALSE" with a hit rate of only 67%. This values have been derived by a program profiler. According to the prediction that the THEN-clause of the IF is not executed the scheduler is able to start the next iteration every clock cycle.

Figure 14 shows the schedules for normal pipelining and for pipelining with speculative computation. As long as the branches are correctly predicted the schedule in figure 14b is 6 times faster. The "rst" actions implement the restore phase after a prediction error at the IF statement (see box "2" for the exact point in time). They restore the correct values of val and l (or u respective). After restoring, the THEN-clause of the IF can be executed which depends on the "write" and the two "cp" actions. The "cp" actions perform only a simple copy from one register into another.

Because the overall prediction accuracy is only 67% the typical situation is that after two correct predictions a false one follows. Figure 15 shows this situation. The dashed actions have never been executed (they belong to the THEN-clause of the IF which were not taken). The shaded actions have been executed speculatively because of the prediction in iteration n+2. Now they become illegal and have to be made undone what is done by the restore actions "rst". Then, the THEN-clause of the IF statement of iteration n+2 is executed and the next iteration will follow. The average execution time is approximated as follows:

probability of correct prediction *
no. of clock cycles on predicted path
+ probability of prediction error *
no. of clocks cycles on other path
(including time for correction).

In our case: 0.67*1 + 0.33*5 = 2.32. Compared with the normal ASAP schedule this means a speed up of 6/2.32 = 2.586. The registers for l, u and val have to be extended to register sets with 3 entries each.

Results

We used speculative computation with multiple branch prediction for the three realistic programs key1, key2 and quick described above.

The speed up depends mainly on the number of iterations which we derived from a program profiler. Table 1 shows the speed up and hardware overhead due to the register sets. We calculated hardware overhead for static latches (as in figure 5b) using the cell library of LSI Logic 1.5 μm gate array.

program	key1	key2	quick-sort
speed up	1.5	3.0	2.586
pipeline latency			
without spec. comp.	3	5	6
with spec. comp.	2	2	1
avg. no. iterations	120	8	---
branch prediction accuracy	99%	88%	67%
hardware overhead	0%	13%	19%
no. additional registers	0	3	6

table 1: speed up of realistic algorithms as a result of speculative computation with multiple-branch prediction

It is possible to use another restore scheme with less hardware and more clock cycles by actively "undoing" operations. For the algorithm exam, e.g., the register count may shrink from 13 to 7 while the restore phase growths from 1 to 2 clocks. But these schemes are more sophisticated and not part of this paper.

Summary

We have demonstrated the use of low-level speculative computation combined with multiple branch prediction for application ASICs, in particular coprocessors. We did experiments with several examples and compared the results with normal ASAP scheduling. The speed up is of a factor 1.5...3.0, while the hardware overhead is of only 0...19 %.

The approach can be used for synthesis but automatic overhead minimization is still an open problem. For complex tasks, however, synthesis seems the only way to apply efficient low-level speculative computation in circuit design.

References

[1] J. Steinleitner, "Design and Implementation of a benchmark for small 'embedded systems' with real-time constraints", master thesis, Institut für Datenverarbeitungsanlagen, Technische Universität Braunschweig, Germany, 1991 (in German).

[2] G. Goossens et al., "Loop optimization in register-transfer scheduling for DSP-systems," 26th DAC, pp. 826-831, 1989.

[3] J.E. Smith, "A Study of Branch Prediction Strategies," SIGARCH Newsletter, Vol. 9, No. 3, pp. 135-148, 1981.

[4] J.K.L. Lee, "Branch Prediction Strategies and Branch Target Buffer Design," COMPUTER, pp. 6-22, Jan 1984.

[5] D.R. Ditzel and H.R. McLellan, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero," Proc. 15th Ann. Symp. on Computer Architecture , pp. 2-9, 1987.

[6] R.W. Holgate and R.N. Ibbett, "An Analysis of Instruction-Fetching Strategies in Pipelined Computers," IEEE Trans. on Computers, Vol. C-29, No. 4, pp. 325-329, April 1980.

[7] R. Ernst, "Long Pipelines in Single-Chip Digital Signal Processors - Concepts and Case Study," IEEE Trans. Circuits and Systems, Vol. 38, No. 1, pp. 100-108, Jan 1991.

[8] P.Y.T. Hsu, "Highly Concurrent Scalar Processing," Proc. 14th Ann. Symp. on Computer Architecture, pp. 386-395, 1986.

[9] G.S. Sohi, "Instruction Issue Logic for High-Performance, Interruptable Pipelined Processors," Proc. 15th Ann. Symp. on Computer Architecture, pp. 27-34, 1987.

[10] A.R. Pleszkun et al., "WISQ: A Restartable Architecture Using Queues," Proc. 15th Ann. Symp. on Computer Architecture, pp. 290-299, 1987.

[11] U. Holtmann, "Hierarchical Behaviour Representation in the Braunschweig Synthesis System," IFIP International Workshop on Applications of Synthesis and Simulation, Lenggries, Germany, Aug 25-28, 1991.

[12] M.S. Quale, L.K. Grover, "Software Pipelining for Loop Synthesis," 5th Int. Workshop on High-Level Synthesis, Bühlerhöhe, Germany, March 3-6, 1991.

[13] R. Camposano, "Synthesis using Path-Based Scheduling: Algorithms and Exercises," 4th Int. Workshop on High-Level Synthesis, Shawmut Inn, Kennebunkport, Maine, USA, October 15-18, 1989.

[14] R. Potasman et al., "Percolation Based Synthesis," 27th DAC, pp. 444-449, 1990.

[15] C. Ricken, "Optimierung der automatischen Einpegelung eines HDTV-Chroma-key-Mischers, "master thesis, Institut für Nachrichtentechnik, Technische Universität Braunschweig, Germany, 1992.

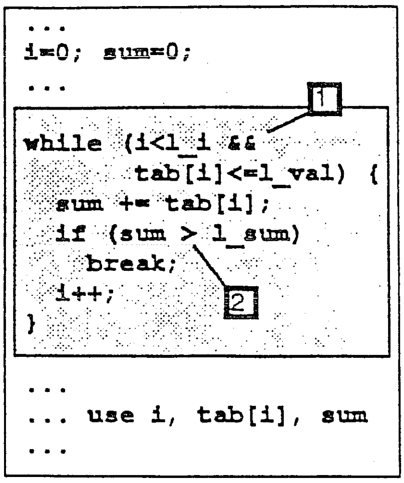
[16] R. Ernst, J. Henkel, "Hardware-Software Codesign of Embedded Controllers Based on Hardware Extraction," Int. Workshop on Hardware-Software Codesign, Estes Park, Colorado, 30.9.-2.10.1992.

[17] F.W. Burton, "Speculative Computation, Parallelism, and Functional Programming," IEEE Trans. on Computers, Vol. C-34, No. 12, pp. 1190-1193, December 1985.

[18] J.A. Fisher, "2^N-Way Jump Microinstruction Hardware and an Effective Instruction Binding Method," Proc. of MICRO 13. In: SIGMICRO Newsletter, pp. 64-75, Sep.-Dec. 1980.

[19] K. Diefendorff, M. Allen, "Organization of the Motorola 88110 Superscalar RISC Microprocessor," IEEE Micro, pp. 40..63, April 1992.

[20] R. Ernst, "Architecture of a Monolithic Digital Signal Processor with a Novel Control Flow Concept," ESSCIRC, Delft, 1986, pp. 110-112.



```
...
i=0; sum=0;
...
while (i<l_i &&
       tab[i]<=l_val) {
  sum += tab[i];
  if (sum > l_sum)
    break;
  i++;
}
...
... use i, tab[i], sum
...
```

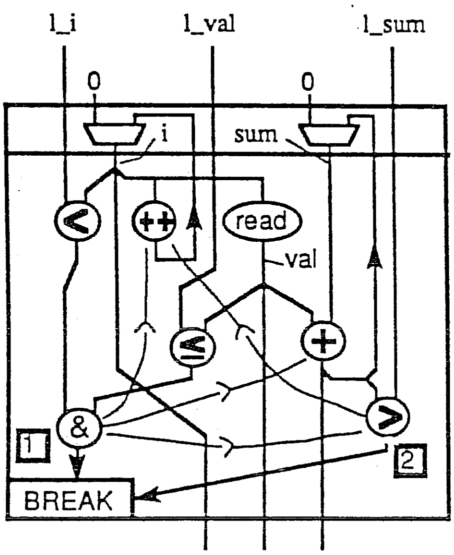
1 break condition (we call every break, condtion or branch which may end the loop a "break condition")

Figure 1:

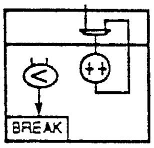
This program, named "exam", is a typical algorithm: the main part of the runtime is spend in loops which mostly perform integer arithmetic, use arrays, have several IF's and data-dependent break conditions.

It is no complete procedure but a part from a larger program which the HW/SW co-design system decided to be executed on a coprocessor.

The loop inside the shaded box is to be synthesized. The small boxes "1" and "2" point to conditions which may end the loop. "use ..." means that a variable calculated inside the loop is read by statements outside the loop (when the loop has ended).



Legend:

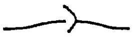


break block: describes all kinds of loops: REPEAT, WHILE, FOR/... . The body contains the actions which are executed every iteration. The block stops when one of the break conditions becomes true.

block multiplexor: used to describe variables within loops. The output changes only at the beginning of each iteration. The left input is only used in the first iteration.

< read (simple) action: performes data calculation

data connection & data dependency



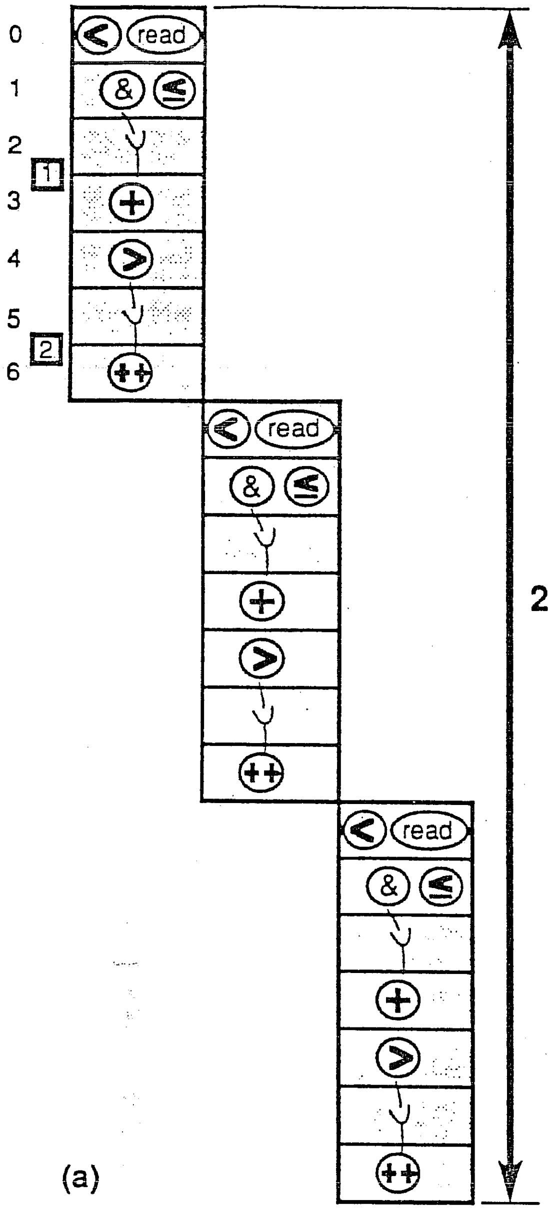
control dependency

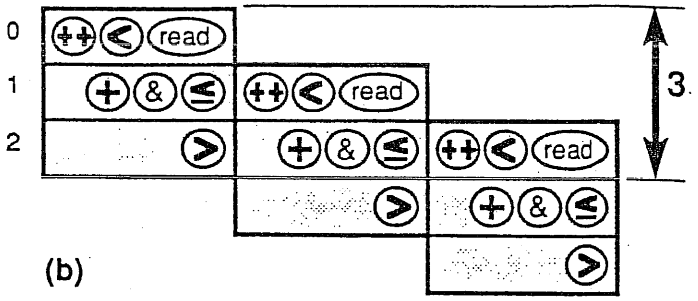
1 break condition which ends the block

Figure 2:

CDFG of "exam". The statements of the program have been transformed into actions with the same behaviour. "tab[i]" has become a read-action what means that tab[] is an array and that the access to an element needs some time.

The value of tab[i] is ridden only one time and internally called "val". The curved lines describe the control dependencies. For example: the addition of the statement "sum += tab[i]" is pointed by the "&" what means that these statement is only executed when the WHILE-condition is true.





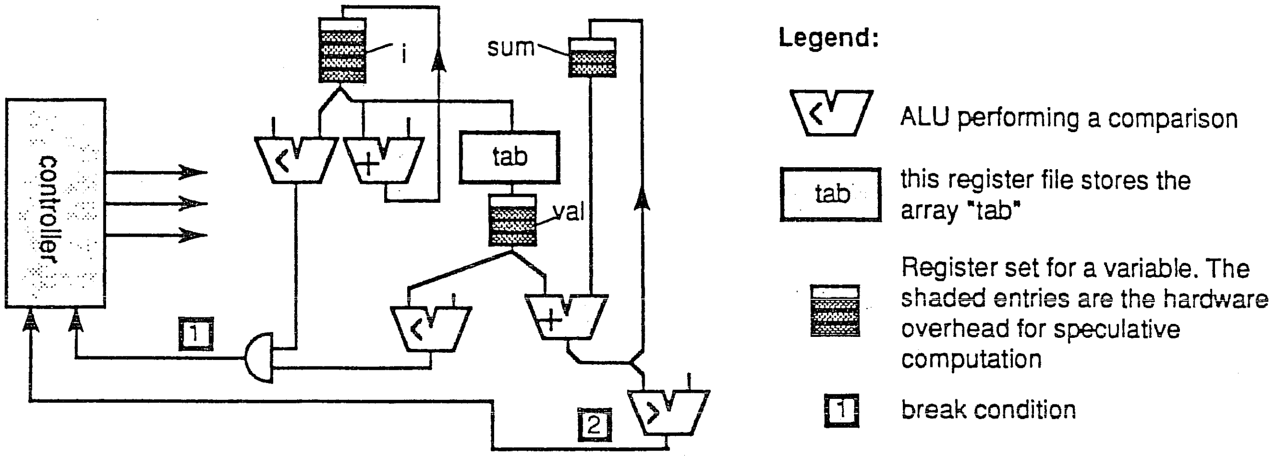
1 The point in time when the controller is able to react to the break condition

Control dependency. The action where the arc is pointing to is only executed if the condition calculated by the action on the other side is true.

Figure 3:

Two possible schedules of "exam". The schedule on the left side (a) uses no speculative computation. The two empty clocks are considered necessary due to the control dependencies (see the curved lines). Because of data pedendencies no pipelining is possible and each iteration takes 7 clock cycles.

The other schedule (b) uses speculative computation and is able to apply a very dense pipelining with one iteration starting every clock.



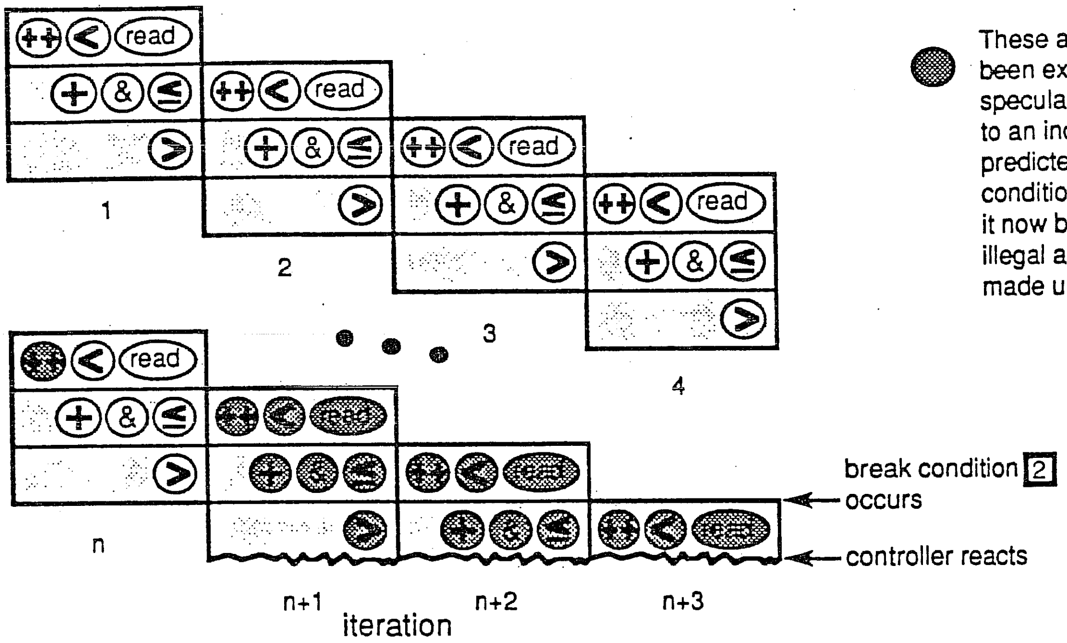
ALU performing a comparison

this register file stores the array "tab"

Register set for a variable. The shaded entries are the hardware overhead for speculative computation

break condition

Figure 6: Controller and datapath for example "exam". The boxes "i", "val" and "sum" are register sets as described in figure 5.



These action has been executed speculatively. Due to an incorrectly predicted break condition (branch) it now becomes illegal and must be made undone.

Figure 4: The situation if condition 2 breaks the loop within iteration n. The shaded actions have been executed speculatively and now they become illegal. Because of the loop break in iteration n, all actions of following iteration have been initiated illegally. The increment of iteration n is also illegal because of a control dependency (see CDFG).

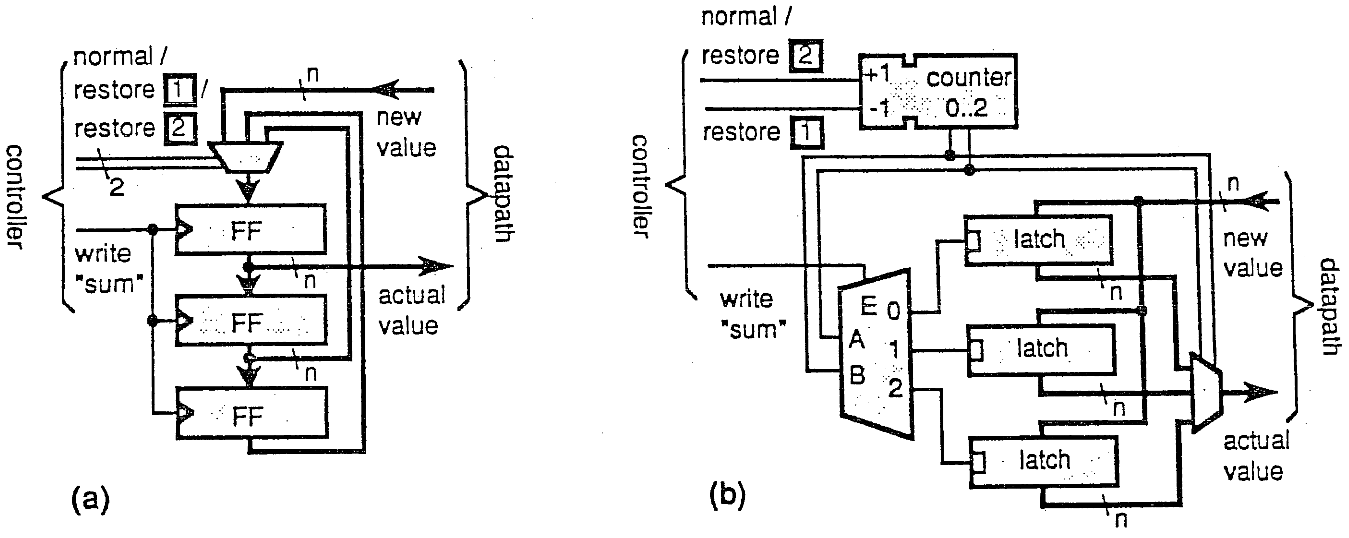
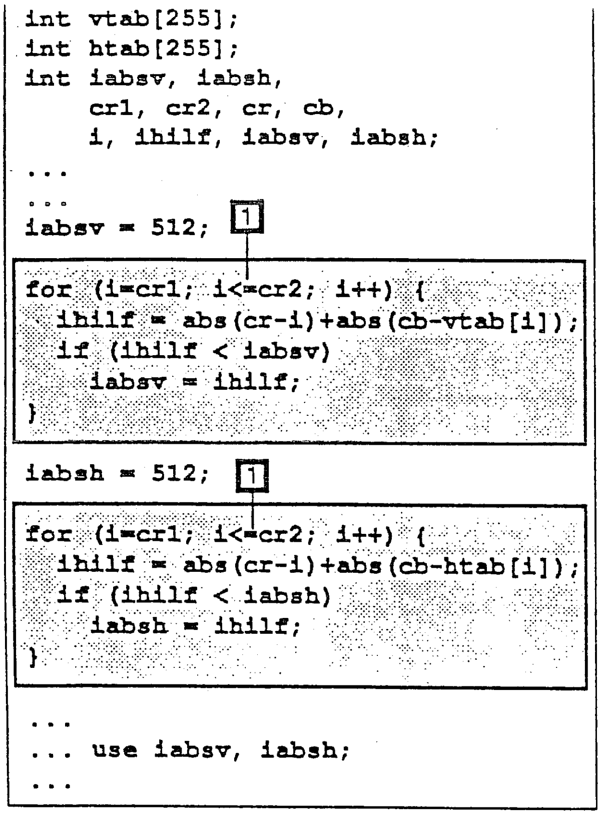


Figure 5:

Two possibe low-cost implementations of the register set storing variable "sum" of program "exam": a shift register with master-slave (dynamic) flipflops (a) or a ring buffer with static latches (b).

1 branch condition



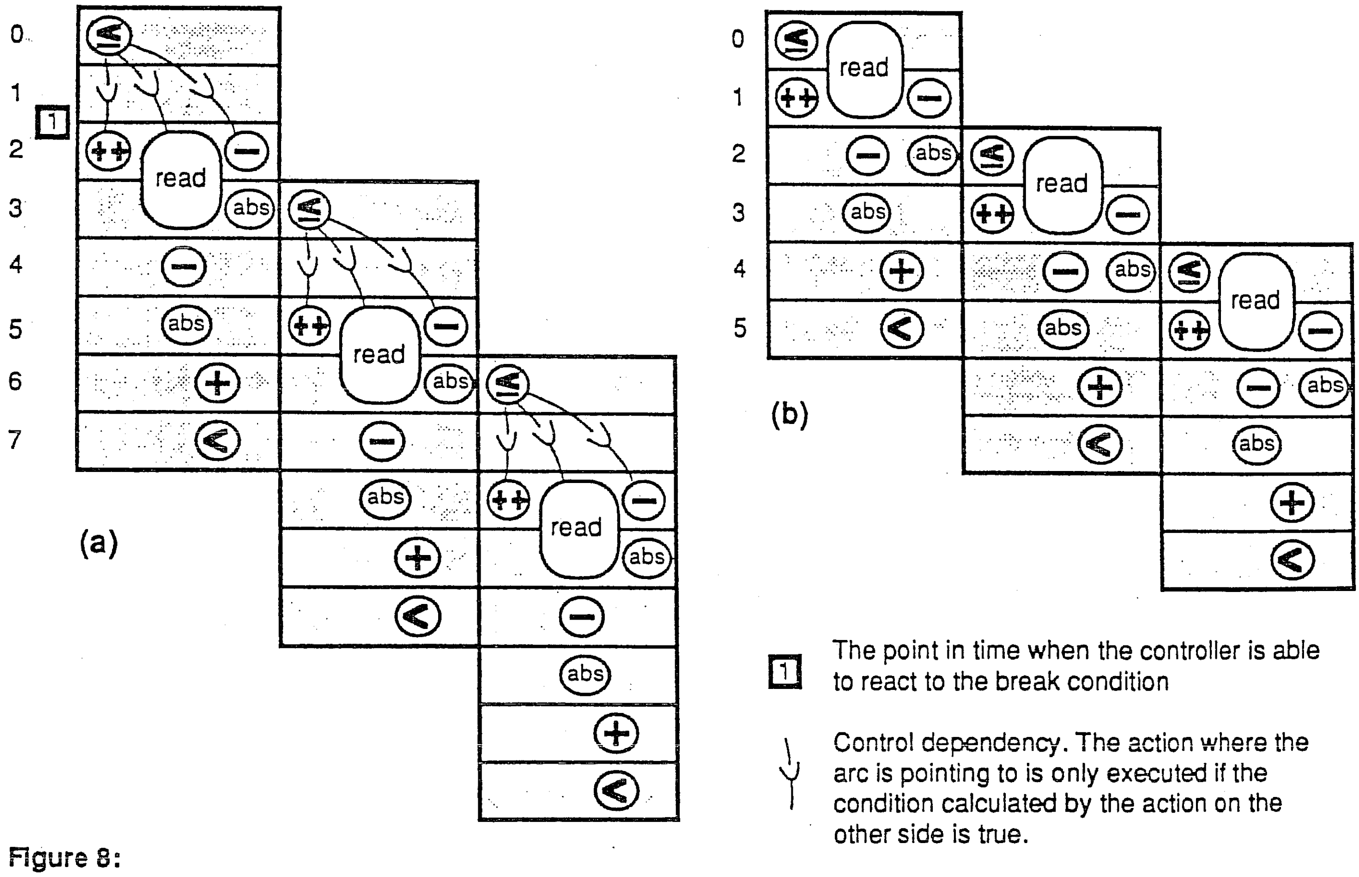
```
int vtab[255];
int htab[255];
int iabsv, iabsh,
    cr1, cr2, cr, cb,
    i, ihilf, iabsv, iabsh;
...
...
iabsv = 512; [1]

for (i=cr1; i<=cr2; i++) {
  ihilf = abs(cr-i)+abs(cb-vtab[i]);
  if (ihilf < iabsv)
    iabsv = ihilf;
}

iabsh = 512; [1]

for (i=cr1; i<=cr2; i++) {
  ihilf = abs(cr-i)+abs(cb-htab[i]);
  if (ihilf < iabsh)
    iabsh = ihilf;
}

...
... use iabsv, iabsh;
...
```

1 break condition (we call every break, condition or branch which may end the loop a "break condition")

Figure 7:

"blue1" is part of the blue screen algorithm. Although "blue1" is only 1% of the complete program it needs 90% of the processor time.

The loops inside the shaded boxes are to be synthesized. The small box "1" points to the ending condition of the loops. "use ..." means that the variables "iabsv" and "iabsh" calculated inside the loops are read after the loop has ended.

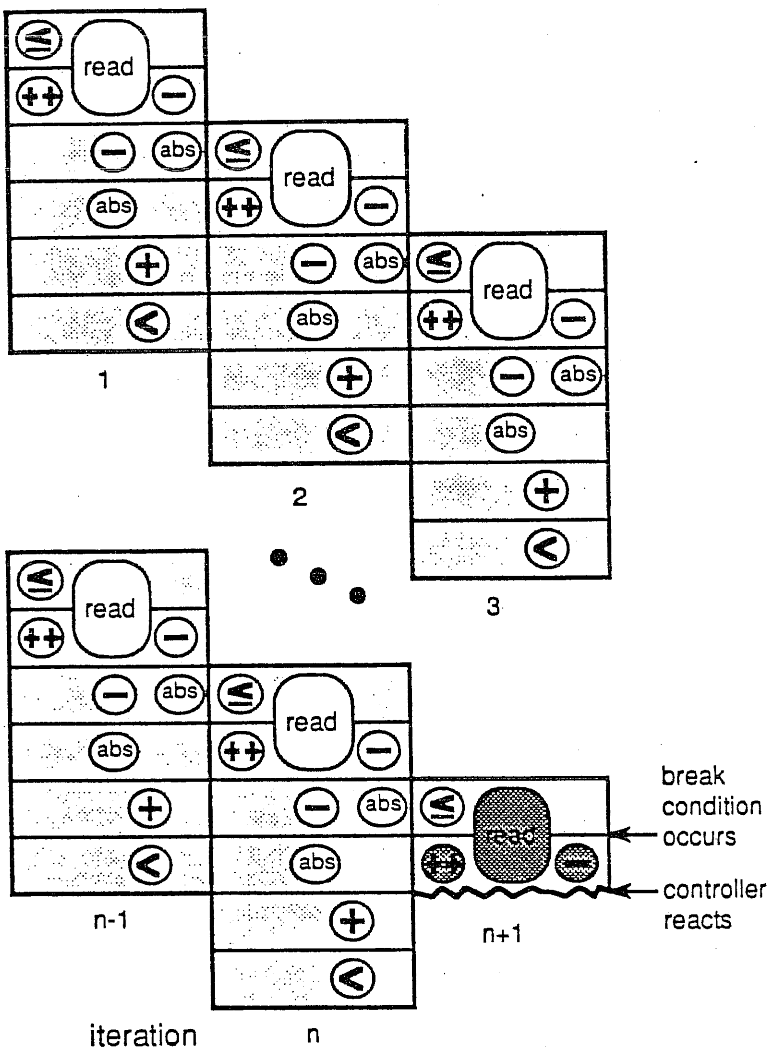


1 The point in time when the controller is able to react to the break condition

Control dependency. The action where the arc is pointing to is only executed if the condition calculated by the action on the other side is true.

Figure 8:

Schedule of "key1" with pipelining (a) and pipelining combined with speculative computation (b). To simplify these figure only the first of the two loops is shown (the second has exactly the same schedule).



● These action has been executed speculatively. Due to an incorrectly predicted break condition (branch) it now becomes illegal and must be made undone.

Figure 9:

The situation when "key1" breaks the loop within iteration n. The shaded actions have been executed speculatively and now they become illegal.

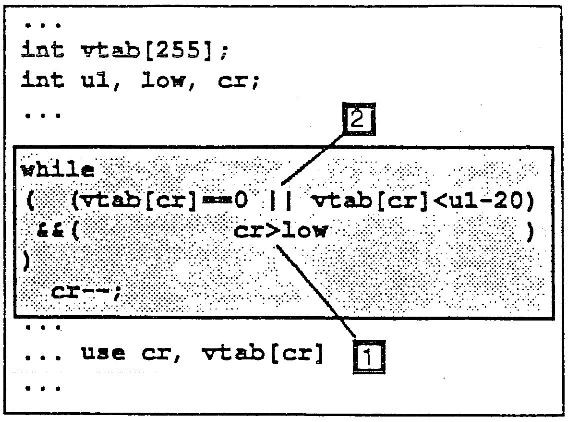
```
void quick (int L, int U,
            int field[]  )
{
  int free, crit, up,
      l, u, val;
  free=u;
  crit=field[free];
  up=TRUE;
  l=L; u=U-1;

  while (l<u) {
    if (up) {
1     val=field[l];
      if (val>crit) {
        field[free]=val;
2       free=l;
        up=FALSE;
      }
      l++;
    } else {
      val=field[u];
      if (val<krit) {
        field[free]=val;
2       free=u;
        up=TRUE;
      }
      u--;
    }
  }

  field[free]=crit;
  if (L<free-1)
    quick(L,free-1);
  if (free+1<U)
    quick(free+1,U);
}
```

1 break condition (may end the loop)

Figure 13:

The inner loop of this Quicksort algorithm is to be synthesized. The break condition no. 1 (the WHILE-condition) is predicted to true with an average accuracy of 99% due a profiler report. The condition of the inner IF is predicted to false with an accuracy of only 67% (profiler). The schedule in figure 14 will show that it is not necessary to predict the condition "up" of the first IF because this value is always known in time.

Because the THEN- and the ELSE-part of the first IF-statement are nearly identical only the THEN-part will be shown in the following figures.



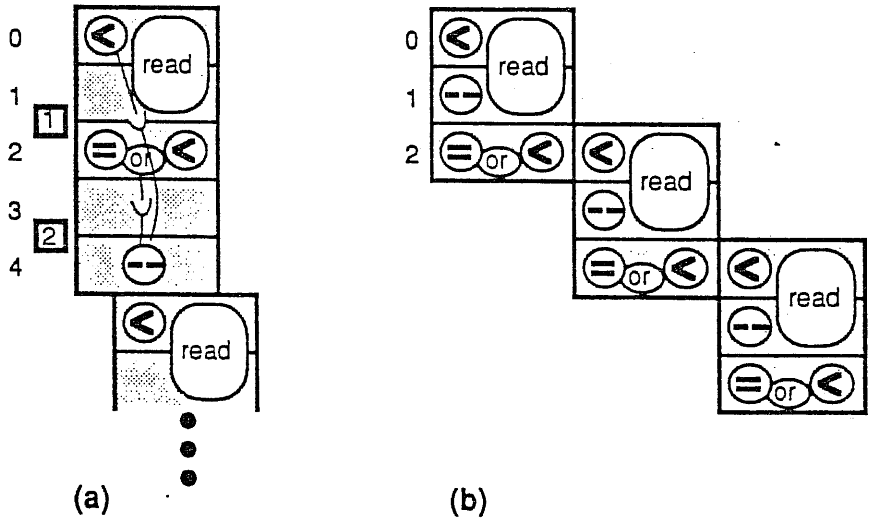
```
...
int vtab[255];
int ul, low, cr;
...
while
(  (vtab[cr]==0 || vtab[cr]<ul-20)
 && (           cr>low             )
)
  cr--;
...
... use cr, vtab[cr]
...
```

[1] break condition may end the loop

Figure 10:

"key2" is part of the blue screen algorithm. It is less critical than "blue1" but has more complex control dependencies.

The shaded loop is to be synthesized. After the loop has ended the values of "cr" and "vtab[cr]" are read.

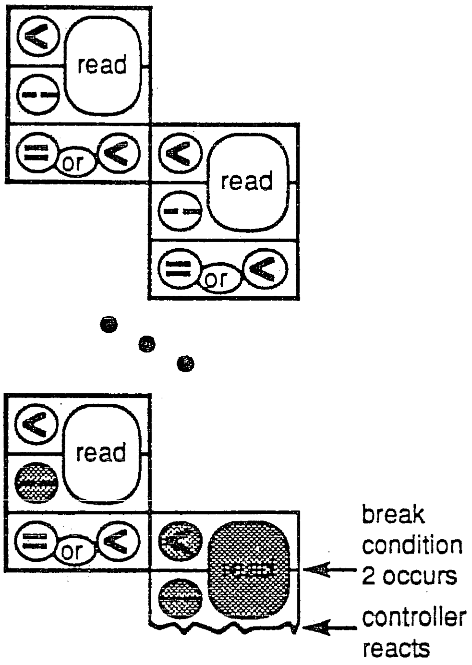


[1] break condition

Figure 11:

a: Schedule for "key2". No pipelining is possible because of a data dependency from the decrement to the read.
b: Schedule possible due to speculative computation.

The "or" is chained together with the two comparisons in the same clock cycle.



● These action has been executed speculatively. Due to an incorrectly predicted break condition (branch) it now becomes illegal and must be made undone.

Figure 12:

The situation when the break condition 2 breaks the loop within iteration n. The shaded actions now have become illegal.

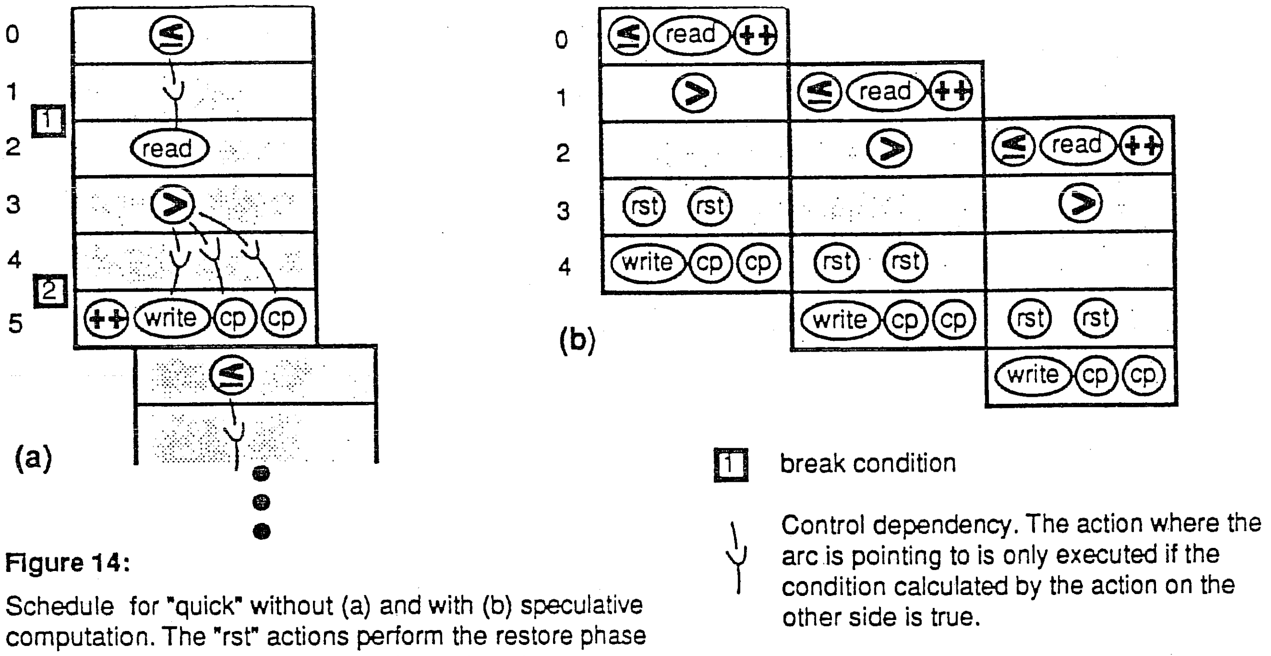

0
1
2
3
4
5
1
2
≤
read
>
++
write
cp
cp
(a)
0
1
2
3
4
read
rst
(b)
1 break condition
Control dependency. The action where the arc is pointing to is only executed if the condition calculated by the action on the other side is true.

Figure 14:

Schedule for "quick" without (a) and with (b) speculative computation. The "rst" actions perform the restore phase after an incorrectly predicted condition of the inner IF-statement. As soon as the restore phase is completed the actions belonging to the body of these IF are executed.

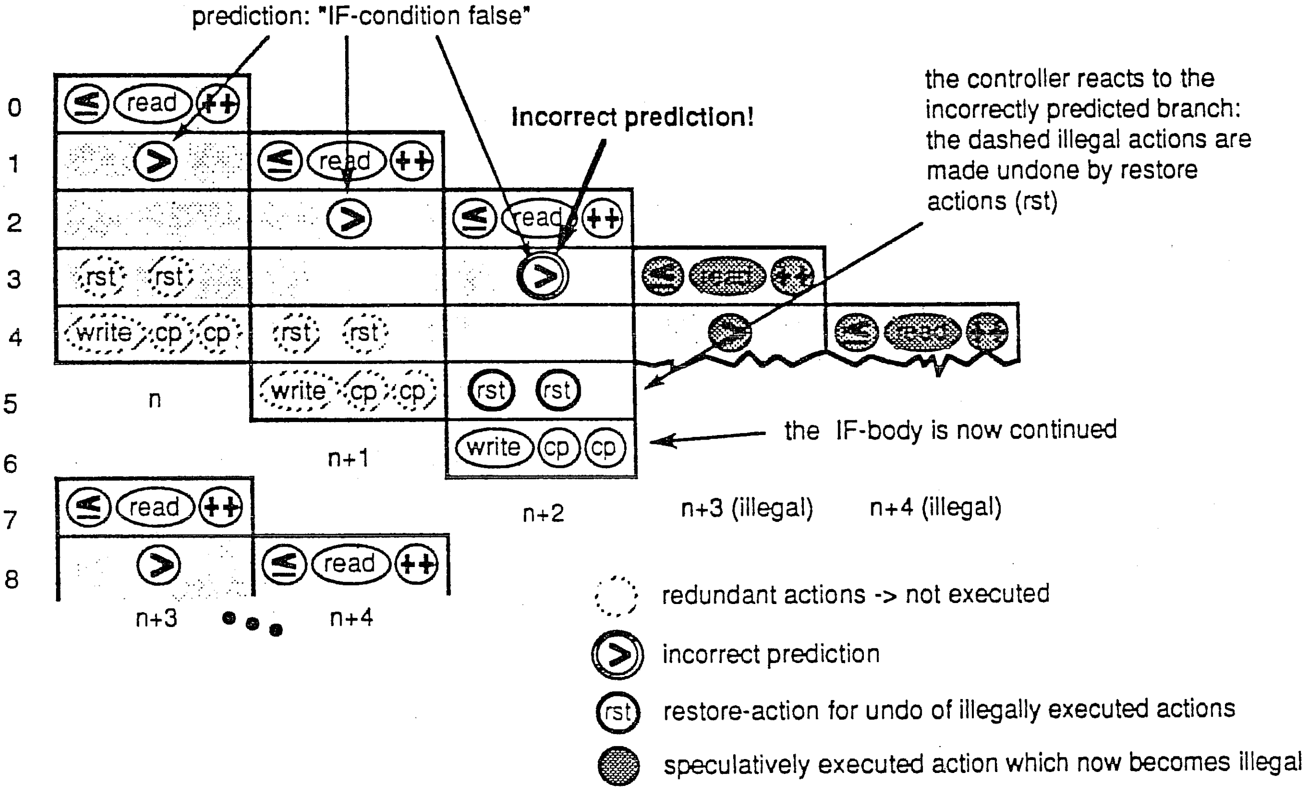

prediction: "IF-condition false"
Incorrect prediction!
the controller reacts to the incorrectly predicted branch: the dashed illegal actions are made undone by restore actions (rst)
the IF-body is now continued
0
1
2
3
4
5
6
7
8
≤
read
++
>
rst
write
cp
n
n+1
n+2
n+3 (illegal)
n+4 (illegal)
n+3
n+4
redundant actions -> not executed
incorrect prediction
restore-action for undo of illegally executed actions
speculatively executed action which now becomes illegal

Figure 15:

The IF-condition is predicted correctly with a probability of only 0.67. The figure shows a typical situation: the IF-condition of two following itterations is predicted correctly but not the third.

The shaded actions have been executed speculatively and now they become illegal and have to be made undone. This is done by the outlined rst-actions. As soon as the restore-phase is completed the IF-body is executed and the next iteration (no. n+3) will follow.