Register Synthesis for Speculative Computation*

Dirk Herrmann Rolf Ernst

Institute of Computer Engineering
Technical University of Braunschweig, Germany

Abstract

Speculative computation and branch prediction have been used in high-performance processor design for many years. Recently, it has also been applied to high-level synthesis where a priori knowledge of possible control paths provides an even higher performance potential. One problem of speculative techniques is the circuit overhead necessary for correctness preservation. While in processors, overhead is high due to the required generality, high-level synthesis can, again, employ a priori knowledge. The paper presents a register synthesis and allocation technique for speculative computation with branch prediction which is based on life time trees. It creates shift register structures with little register and control overhead.

1 Introduction

Modern RISC processors use branch prediction and speculative computation to increase the potential parallelism in the presence of control dependencies [1]. The major challenge of these techniques is the preservation of a correct processor state without compromising cost efficiency and performance. A common solution for processors is to increase the number of registers to hold results of speculative operations and to use associative memory and tagging techniques, often combined with mechanisms for dynamic scheduling. This solution is costly in terms of interconnect and register file complexity and it is mainly suited for architectures with a centralized register file. However, it provides the flexibility needed for general purpose processors.

For high-level-synthesis, the potential performance increase through speculative computation is even larger: A priori knowledge of algorithm specific control path segments allows controlled prediction [3] or parallel execution of branches [5]. On the other hand, while the overhead for speculation is only a small part of current RISC processor cost, high-level synthesis is used to generate smaller circuits where the overhead would be much less acceptable. Most importantly, however, high-level synthesis exploits specialized registers and interconnect which does not match current techniques used in processors.

In this paper, we will present a register synthesis and allocation technique to work with speculative computation in high-level synthesis. It generates shift registers and uses a generalized variable life time analysis for register allocation which is based on life time trees. The technique has been developed for MBP-SC [2] but is not limited to this specific speculative computation approach.

The technique presented here distinguishes three phases:

First, by *value life time analysis* the relationship between the production of a value and its last consumption is determined. Conventionally, intervals are used to represent value life times [4]. This shows to be inappropriate if different paths through the control flow of an algorithm are scheduled separately: Then each operation can be assigned as many execution cycles as there are paths it belongs to, and, consequently, the life time of values is path dependent, too. Life times in our approach therefore are represented by trees.

In the second phase, *logic register file synthesis*, a set of logic shift registers (instead of register files as in processors) is generated to store each value over its life time. The algorithm presented in this paper determines the necessary shift register length and provides specialized connectivity to realize value restoration in case of prediction errors. Additional constraints on the register structure due to loop pipelining or the presence of multi-cycle operations are respected.

In the final *register allocation* phase, logical registers are mapped to physical ones. The aim of this phase is to minimize the number of physical registers needed. This corresponds to conventional register synthesis. For this paper, we will regard the first two

[1]Supported by the German Science Foundation (DFG)

phases only.

The rest of this paper is organized as follows: First we describe how state graphs are used in our system as a generalized way to represent scheduling information. The following section illustrates how life time analysis is performed for these state graphs. Then, after a short discussion about different forms of register file organization, a rule based algorithm for logical shift register synthesis is presented. By applying the algorithm to some examples we show the register overhead caused by the MBP-SC scheduling technique. Finally, we give a conclusion.

2 Schedule representation

```
do {
    temp = a;
    a = temp + j;
    b = 2 + a;
    x = b + temp;
    c = a * a;
    d = a * b;

    if (5 < a) j = a - 2;
    else       j = j * a;
} while (j <= 100);
```

Figure 1: Example C Code

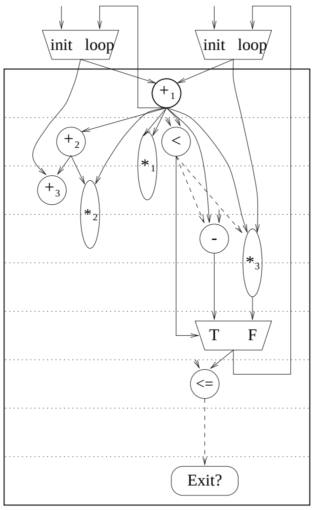

Figure 2: ASAP-Schedule

To illustrate the algorithms, we will refer to the C-code example in fig. 1 throughout this paper. In control and data flow graphs (*CDFGs*) like fig. 2, besides circles for operations and arrows for data dependencies, dashed arrows are used to denote control dependencies (a controller delay of two cycles is assumed). Rectangles with rounded corners represent the evaluation of conditions within the controller (e.g., *Exit?* for a possible termination of a loop). Usual multiplexors with inputs for the `true` and `false` case are labelled T and F, while *loop multiplexors* labelled *init* and *loop* are used to describe the initialization of values for the first iteration of the loop and, for further iterations, the passing of values from iteration n to iteration $n+1$. Note that in our example the multiplication is not pipelined.

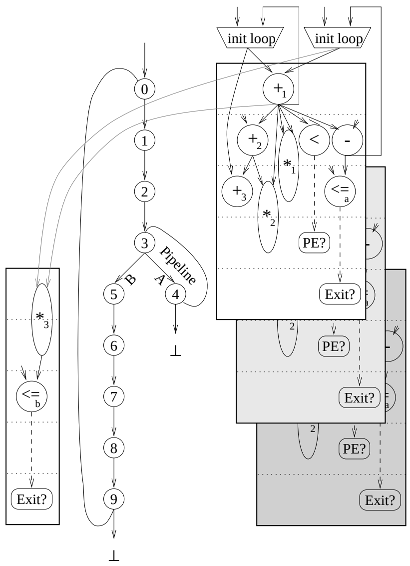

Figure 3: MBP-SC Schedule

The MBP-SC algorithm schedules different paths or sets of paths of the control flow independently and generates a combined schedule in form of a state graph. To keep matters simple, in this paper a state graph will be regarded as a representation of a Moore automaton. We will refer to nodes of a CDFG as *operations* and to nodes of a state graph as *steps*.

Consider the example code in fig. 1 and the corresponding ASAP-Schedule in fig. 2. The critical path from the $+_1$-operation to the same operation in the next iteration limits the minimum pipeline latency to six cycles. Within the loop, two paths can be distinguished: Path A leads through the `then` part, and B through the `else` part of the `if` block. Assume that code profiling determined that most of the time path A was taken. Fig. 3 shows the two schedules and the state graph generated by MBP-SC with A being predicted (each step of the state graph corresponds to the cycle of the schedule on the same horizontal level):

- The pipelined part of the loop consists of the schedule steps 0 to 4, where steps 0, 1 and 2 form the initialization phase. In this part only operations of path A are executed, while everything else (operation $*_3$, multiplexors and control dependencies between operations) has been removed. As a result, a pipeline latency of two cycles could be achieved. Nevertheless, the `if` condition still has to be evaluated in the pipelined part: In step 3 of fig. 3 the result is needed to determine if a prediction error occured (*PE?*: controller can react on a prediction error) and, in this case, to change to the second schedule part. Otherwise the execution of the pipeline is continued.

- The part of the schedule consisting of steps 5 to 9 in fig. 3 executes the operations of path B. Operations common to A and B which have already been executed in the first part of the schedule don't have to be invalidated or repeated, except for those, which are invalid due to the prediction error. In our example this is only true for the $\leq$-operation, which computes the loop exit condition. On path B, the operation $*_3$ delivers input data for the condition, thus in step 2 it has been executed with wrong input values. Therefore in step 7, the condition is evaluated again with the correct inputs. After all necessary operations of the `else` path are executed, a transition to step 0 is taken, thus for the following loop iterations the pipelined part of the schedule with the prediction of path A is executed again.

By using MBP-SC the pipeline latency in this example could be reduced, but a penalty must be payed in case of prediction errors: Extra cycles are needed to execute the remaining operations of the unpredicted paths and the pipeline has to be reinitialized. Nevertheless, given that the prediction was correct for 90% of the iterations, the average iteration length in our example would be 2.61 cycles, compared to 6 cycles without the use of MBP-SC.

As can be seen from the $\leq$-operation, a schedule generated with MBP-SC can not be described as a mapping *operation* $\rightarrow$ *cycle*. Instead we use the state graph itself to represent the relationship between operations and their execution times. For the above example, table 1 shows the operations which are executed in each step. In addition, for each operation op in step s its iteration number $i(op, s)$ is given and, in case of multi-cycle operations, which part c of the full execution time is executed in that step.

Step	$+_1$	$+_2$	$+_3$	$-$	$<$	$\leq_a$	$\leq_b$	$*_1$		$*_2$		$*_3$	
	i	i	i	i	i	i	i	i	c	i	c	i	c
0	0												
1		0		0	0			0	0				
2	1		0			0		0	1	0	0		
3		1		1	1			1	0	0	1		
4	2		1			1		1	1	1	0		
5												0	0
6												0	1
7							0						

Table 1: Executed Operations

Further, for each edge e_{s_1,s_2} of the state graph, $o(e_{s_1,s_2})$ defines the *iteration offset* between the steps s_1 and s_2. For the edges $e_{4,3}$ and $e_{9,0}$ in fig. 3 the offset is 1, for all other edges the offset is 0. This way dependencies across pipeline iterations can be modeled: Operation $+_1$ passes its result to operation $-$ in the same iteration. Step 4 executes $+_1$ for the third iteration of the pipeline (see table 1). The corresponding $-$ is, if the loop doesn't exit, executed in the second traversal of step 3, but (according to table 1) as part of the second loop iteration. The iteration offset of 1 for $e_{4,3}$ provides the necessary alignment.

3 Life time analysis

The life time of a value is the interval between the moment of the production of the value and its last use. Applied to state graphs, the step s_r where the result r of an operation is computed determines the begin of the life time of r. All sequences of steps starting from s_r and ending at a step where r is consumed belong to the life time of r. The union of these sequences can be represented by a *life time tree*. We will use the term *node* only for nodes of a life time tree. The symbol op_i will denote operation op executed in iteration i.

Consider operation $(+_1)_0$ in step 0 of fig. 3: In step 1 to 3 the result is consumed by the data dependent operations of iteration 0, and, in step 2, by $(+_1)_1$. If path A was taken, then in step 4 operation $(+_3)_1$ would be reached. Otherwise the life time would lead along path B with steps 5-9,0-2 to operation $(*_3)_0$ in step 5 and 6, $(+_1)_1$ in step 0, and $(+_3)_1$ in step 2. The left part of fig. 4 shows the corresponding life time tree T_0 for this example. Together with each node, the data dependent operations are displayed (operations of the

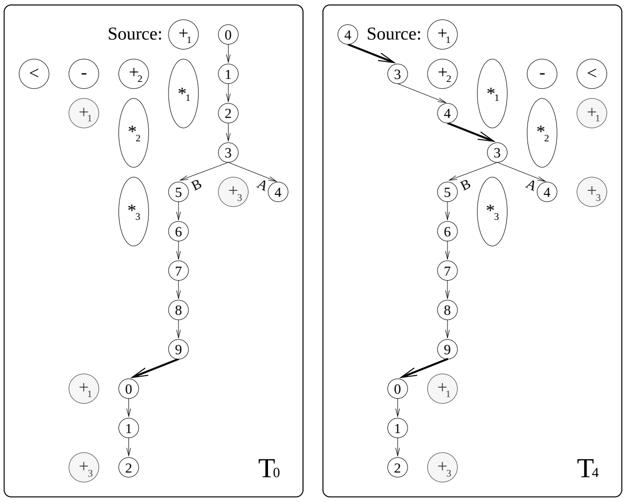

Figure 4: Two (of Three) Life Time Trees for $+_1$

following iteration are colored grey). Edges of the life time tree which are drawn with bold lines correspond to edges in the state graph with an iteration offset of 1.

Please note that there are three steps in the state graph where operation $+_1$ is executed: 0, 2 and 4. As a consequence, there are three different life time trees. The right part of fig. 4 shows, as a second example, the tree for $+_1$ in step 4. Here the pipeline is filled and executes three loop iterations in different phases (compare fig. 3).

Life time trees allow to treat data dependencies within the same iteration (*intra iteration dependencies*) equally to those, which cross iteration boundaries (*inter iteration dependencies*): If a value lives for multiple iterations, the corresponding life time tree is obtained by 'unrolling' the state graph. Therefore each step of the state graph may appear several times in a life time tree, as can be seen from T_4 in fig. 4.

3.1 Generating life time trees

The generation of the life time tree for a value v computed in step s by operation op is done recursively, starting with s as current step. When traversing through the state graph, operations op' in the current step s' are data dependent of v if two conditions hold:

- In the CDFG, op' shows a data dependent of op, possibly across one or more loop multiplexors. Then, let Δi denote the difference in the iteration index of both operations, which is equivalent to the number of loop multiplexors between them.
- The iteration indices of both operations (aligned with regard to step s') must match. Let o describe the summarized iteration offsets of all edges that have been traversed between s and s'. Then the following equation must hold:

$$i(op, s) + \Delta i = i(op', s') + o \qquad (1)$$

  To see why this is true, consider operation $+_1$ in fig. 3. It is connected with the left operand of operation $+_3$ via one loop multiplexer ($\Delta i = 1$). For operation $+_1$ executed in step 4, Table 1 shows an iteration index of $i(+_1, 4) = 2$. Thus, the consuming operation $+_3$ must belong to iteration index 3. In step 2, which can only be reached from step 4 in case of a prediction error (path B), the iteration index of $+_3$ is $i(+_3, 2) = 0$. To align these indices in life time tree T_4 in fig. 4, the sequence of steps from the source operation $+_1$ in step 4 to operation $+_3$ in step 2 must lead across three edges with iteration offsets of 1 (marked with bold arrows), therefore giving a value of $o = 3$.

The life time tree is extended until it is certain that there won't be any further data dependent operations found. This can be determined using the summarized iteration offset o, which is the only value in equation 1 depending on the sequence of steps between s and s'.

4 Logical shift register synthesis

4.1 Alternatives to the shift register solution

For each value v computed during the execution of the algorithm, an arbitrary register structure has to be provided, which fulfills the requirements, that v is stored over its full life time and that in no moment two different instances of v (e.g., of different iterations) are assigned to the same register. Although in this paper we focus on the generation of specialized shift registers, alternative realizations are possible:

- Each active (i.e. currently executed) iteration holds a full set of registers of its own, including additional registers for predicted values. A disadvantage of this solution is the bad register utilization: The number n of register sets is determined by the number of iterations executed in parallel in the pipeline. Let m be the maximum number of values that are alive in parallel during the execution of an iteration. Then the register file must hold $n \times m$ registers, even if there is only a single active iteration at a time that needs the full set of its m registers.

- For each operation op which is executed dependent on a prediction a register set is provided to hold the outputs. This is the solution used in processors. Each value computed by op remains in the same register for its whole life time. This increases control and interconnect complexity as discussed in the introduction.

The first solution doesn't make use of the fact that in every step of the state graph the set of existing values is known. Both alternatives use a dynamic relationship between the operation that uses a value and the logical register where that value is stored. In contrast, by using a shift register for each value, the size of the register sets can be minimized. Further, if a static relationship between each operation and the logical register holding the corresponding input values can be defined, associativity or multiplexor structures for the register sets are unnecessary.

4.2 The algorithm

Related to state graphs, a valid register structure requires that there is no step in which two different values are assigned to the same logical register. Due to the nature of state graphs generated by MBP-SC, the shift registers have the property, that every operation in the CDFG reads its input data always from the same logical registers. Accordingly, results of an operation are always written to the same registers as well.

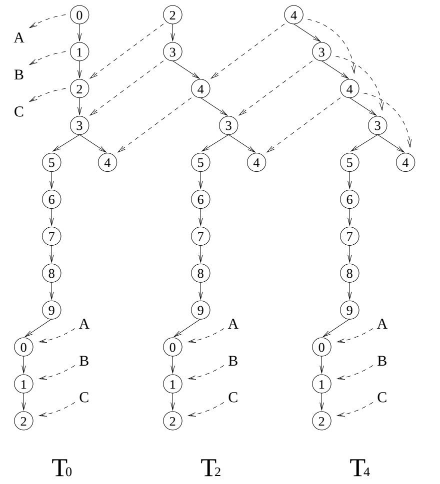

Figure 5: Life Time Interdependencies

This property of the generated shift register structure defines interdependencies between different life time trees of the same operation. In fig. 5 all three life time trees of +1 and their relationships are shown. Dashed arrows between two nodes indicate that the values belonging to their trees will live in parallel at these nodes, i.e. their life times *overlap*. The arrows always point to that node, which represents the value that has been alive for a longer time.

Overlaps of two life times start at the first node where the two values live together, and they end where their courses of life split: In none of the graphs there is a dashed arrow pointing to node 5, because if node 5 is reached, a prediction error has occured and all values of later iterations are invalidated.

To generate a shift register $R_0, R_1, \ldots$ that holds values produced by an operation op, three integer functions are determined: For each data dependent operation op', the value of $r($op'$)$ defines the register stage within the shift register that holds the input data for op'. Further, for any node n of a life time tree T belonging to op, the functions $\underline{r}(n)$ and $\overline{r}(n)$ define an interval of shift register stages. At node n, the value related to T then may be stored in any of these registers.

To compute these functions, first $r($op'$)$ is initialized with -1 for all op'. For all nodes of all life time trees (except for the root nodes) $\underline{r}(n)$ and $\overline{r}(n)$ are initialized with 0, for the root nodes they are initialized with -1. Without further mentioning, $\overline{r}(n) \geq \underline{r}(n)$ has to be provided after any change to $\underline{r}(n)$. Then, iteratively, three rules are applied until no more changes occur:

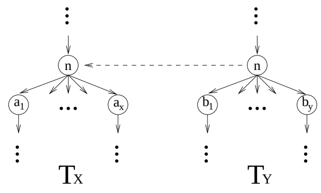

Figure 6: Rule 1

Rule 1: To understand the consequences of a life time overlap, consider fig. 6. If in any node of $b_1, \ldots, b_y$ the value related to T_Y is expected in register R_r, then the corresponding write operation has to be performed in node n of T_Y, regardless of which node of $b_1, \ldots, b_y$ will be the next (Moore automaton). Since a register write changes the register outputs in the following cycle, in node n of T_X the old register content is still available. However, to preserve it, in node n of T_X it must be moved at least to R_{r+1}, again regardless of the path taken. Given that for a life time overlap n_s is the source node of the dashed arrow and n_d is the destination node, this situation can be expressed by the following equation:

$$\bigwedge_{n'_d \in \mathrm{succ}(n_d)} \left(\underline{r}(n'_d) \geq \max_{n'_s \in \mathrm{succ}(n_s)} \{\overline{r}(n'_s) + 1\} \right) \quad (2)$$

Rule 2: To provide stable input signals for a multicycle operation m, this operation must receive its inputs from the same register for all cycles of its execution. Let N denote the set of all nodes of all life time trees where any cycle of m is executed. Then we have to make sure, that in all these nodes a common register stage for the input value of m exists. Since copying of values may only be done towards a higher register stage (in order not to overwrite values of later iterations) for the input register stage of m the following equation must hold:

$$r(m) \geq \max\{\underline{r}(n) \,|\, n \in N\} \quad (3)$$

Rule 3: For a node n let $\mathrm{op}(n)$ denote the set of operations excuted in n. Then that operation in n which expects its input from the deepest stage of the shift register defines the maximum register occupied in n:

$$\overline{r}(n) \geq \max\{r(op) \,|\, op \in \mathrm{op}(n)\} \quad (4)$$

As will be seen from the example, the result of this algorithm is a shift register of minimum length. The connections between registers and operations as well as the internal connections of the shift register are delivered, but not minimized.

4.3 Example

Fig.7 shows the interval $[\underline{r}(n), \overline{r}(n)]$ for each node of T_0 after the algorithm was applied (T_2 and T_4 have been omitted for simplicity). To the right of fig.7 the corresponding shift register structure is displayed. The way the operations are connected to the registers corresponds directly to the resulting function $r(op)$. A dashed arrow from a node to a connection denotes, that in the step of the state graph which corresponds to that node a data transfer within the shift register has to be performed.

Some characteristics of the generated shift registers can be seen from this example:

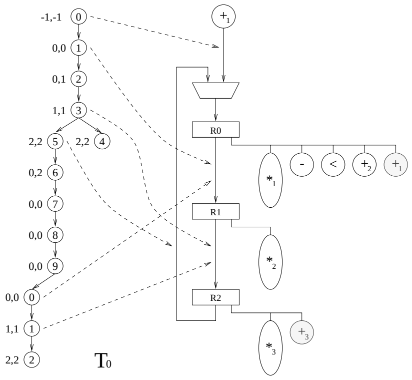

Figure 7: Example Result for T_0

- In contrast to common shift registers, a connection from R_2 to R_0 is generated, which requires an additional multiplexor. The corresponding transfer is performed in step 5 to correct the prediction error.

- The overlapping multi-cycle operations $*_1$ and $*_2$ are handled correctly: Although the pipeline latency is 2 cycles, the value is copied from R_0 to R_1 after only one cycle. This way in steps 1 and 2 the value is provided for $*_1$ in R_0, and in steps 2 and 3 it is provided for $*_2$ in R_1. As a consequence, each stage of the shift register can require to be controlled separately.

5 Results

By applying the life time analysis and register synthesis algorithms to a set of examples (taken from [2] and [3]), we measured the difference in register usage between a list scheduling technique (LS), a pipelined list scheduling technique (PLS) and the MBP- SC scheduling technique, all with the same register allocation algorithm. To reduce the effect of register sharing, only the innermost loop of each algorithm was considered. Two different blue screen algorithms from professional video applications were used (blue2, blue4), two parts of a pattern recognition system (etik3 and 4) and a non-recursive version of quicksort (quick) .

Table 2 shows the register count for each benchmark with each of the scheduling algorithms. All synthesis runs were performed with unlimited hardware resources, to allow deep pipelining. Except for the quicksort, the benchmarks had only one condition in the innermost loop that had to be predicted. Nevertheless, due to pipelining there were between three

	blue2	blue4	etik3	etik4	quick
LS	7	7	33	10	20
PLS	7	9	35	12	23
MBP-SC	8	7	33	9	30

Table 2: Register Usage Comparison

and ten open conditions of different iterations in the pipeline. It shows, that, at least for the examples, there is little register overhead when using MBP-SC scheduling. In half of the cases, we even find a lower register count compared to pipeline scheduling (in case of etik4 even lower than list scheduling), since due to speculation some operations can be executed earlier which leads to shorter life times of intermediate values with the consequence of lower register requirements.

6 Conclusions

A generalized approach to variable life time analysis and register allocation suitable for speculative computation in high-level synthesis has been presented. Using a priori knowledge of execution pathes, it departs from the expensive centralized buffering techniques known from processor design and supports specialized register sets. Even though originally developed for the MBP-SC scheduling approach, it can be adapted to any speculation technique with fixed paths and centralized control.

References

[1] J. Hennessy, D. Patterson *Computer architecture – A quantitative approach. 2nd Ed.,* Morgan-Kaufmann, 1996

[2] U. Holtmann, R. Ernst *Combining MBP-Speculative Computation and Loop Pipelining in High-Level Synthesis,* EDAC'95, pp550-555

[3] U. Holtmann, R. Ernst *Speculative Computation for Coprocessor Synthesis,* Proc. ICCD '93; pp. 126-131, 1993

[4] A. Sharma, R. Jain *Register Estimation from Behavioral Specifications,* Proc. ICCD '94; S. 576-580, 1995

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