UNIVERSITAT POLITÈCNICA DE CATALUNYA

LOOP PIPELINING WITH RESOURCE AND TIMING CONSTRAINTS

Autor: Fermín Sánchez

October, 1995

9

CONCLUSIONS AND FUTURE WORK

This dissertation deals with loop pipelining, by considering constraints on both the number (and type) of resources and the time to execute each loop iteration. We propose two approaches to solve these problems in high-level synthesis of VLSI circuits, superscalar processors and VLIW machines.

- *UNRET* has been devised for loop pipelining with resource constraints. It has been compared to other approaches. Results have shown that *UNRET* obtains optimal-time schedules in most cases, improving the results obtained by other techniques.
- *TCLP* has been devised for loop pipelining with timing constraints. Since no results are available in the literature (scheduling with timing constraints has been previously studied, but it cannot be compared with loop pipelining with timing constraints), we have compared *TCLP* to theoretical lower bounds. Optimal results have also been obtained in most cases.

Both *UNRET* and *TCLP* use an algorithm called *RESIS* to reduce the number of registers required by the final schedule. It has been demonstrated that *RESIS* is very efficient. The obtained results show that not only is *RESIS* a good algorithm for reducing the number of registers required by a schedule, but also *UNRET* and *TCLP* find quite good schedules from the point of view of register requirements before using *RESIS*.

9.1 CONTRIBUTIONS

9.1.1 Software pipelining: retiming and scheduling are separated into independent tasks

This work presents a new way of performing software pipelining based on separating retiming and scheduling tasks. Retiming is used to successively transform the π-graph representing the loop, obtaining different configurations for the same loop. Each configuration is separately scheduled by using the scheduling algorithm proposed at Chapter 5.

In order to determine when no further retiming can be performed, a heuristic called *quality* has been defined. *Quality* is also used to "guess" when a given configuration of a loop body is easier to schedule than another one. This allows the retiming to be guided independently from the scheduling algorithm.

Other authors simultaneously perform retiming and scheduling [RG81, GVD89, CLS93]. Instructions are successively scheduled, and retiming is performed in the π-graph after scheduling each instruction, according to where the instruction has been placed.

Scheduling an instruction may prevent the appropriate retiming from being performed in the π-graph (in order to find a feasible schedule), Therefore, we believe that separating both tasks is a better approach to the software pipelining problem. With this assumption, no interdependence exists between retiming and scheduling, and retiming can be appropriately guided to find a *good* π-graph to schedule. This is not possible in other methodologies, in which retiming is being fixed while scheduling is performed. Moreover, separating retiming and scheduling allows the use of any scheduling algorithm described in the literature, according to the desired objective.

By using a software pipelining approach based on separating retiming and scheduling, we have proposed two different algorithms: *UNRET* for loop pipelining with resource constraints and *TCLP* for loop pipelining with timing constraints.

9.1.2 Analysis of data dependences and scheduling

Separating retiming and scheduling allows the execution of an exhaustive dependence analysis for each π-graph. We have shown how to analyze data dependences in a π-graph from a new point of view: i.e. how they constrain the scheduling process. A dependence $e = (u, v)$ is classified according to where v may be initially scheduled. If v can be scheduled anywhere into the schedule, e is called a *free scheduling dependence* (FSD). Otherwise, e is a *negative scheduling dependence* (NSD) or a *positive scheduling dependence* (PSD) according to whether v may be scheduled before u or not.

We have shown that PSDs and NSDs initially impose constraints on the scheduling process. However, whilst PSDs always constrain the scheduling process, not all NSDs do so. Moreover, some NSDs which initially constrain the scheduling may cease to be restrictive during the scheduling process.

By using NSDs, we have defined the *negative depth* of a node, which is a concept directly related to the maximum time constraints imposed by the node to the scheduling process, and it may change during the scheduling process. *Negative depth* has been successfully used by a scheduling algorithm, obtaining very promising results.

The scheduling algorithm proposed in Chapter 5 is a *list scheduling* which uses a combined priority function. Besides the *negative depth*, we propose two new priority functions: the *0-mobility* and the resource utilization of an instruction.

The *0-mobility* of a node is a particular case of the mobility. It is only able to decide when the node to be scheduled has no mobility. Otherwise, other priority functions take the decision about which instruction will be next scheduled.

Finally, the use of resources performed for each instruction is taken into account by the scheduling algorithm because the schedule may be overlapped. Instructions using fewer resources are in general easier to overlap than instructions using more resources.

9.1.3 Exploration of the solution space

Current software pipelining approaches explore the solution space in only one dimension, increasing the expected initiation interval when a schedule is not found by using the expected initiation interval.

Unlike the software pipelining approaches proposed by other authors, *UNRET* explores the solution space in two dimensions, the initiation interval and the unrolling degree of the loop. To do so, the throughput is explored in decreasing order, starting at the maximum throughput determined by the recurrences of the loop and the resources of the architecture. For each expected throughput, both the unrolling degree of the loop and the initiation interval of a schedule with such a throughput are calculated. The loop is unrolled and the software pipelining approach described in Chapter 6 is used to find a schedule.

In order to explore the throughput in decreasing order of magnitude, a maximum number of cycles of the schedule must be defined by the designer. We have shown how a high percentage of the throughput can be obtained by significantly limiting the maximum number of cycles previously defined. This limitation produces a reduction in the number of schedules to explore, avoids code explosion and in general allows us to obtain schedules which use a reduced number of registers.

Finally, a figure of merit, ε, is defined to measure how far the found schedule is from the theoretically optimal one. ε compares the throughput of the found schedule with the throughput of a theoretical optimal one.

9.1.4 Register reduction

In Chapter 7 of this work we also present *RESIS*, a new algorithm for register optimization. *RESIS* is divided into two phases: *SPAN reduction* and *incremental scheduling*. *SPAN reduction* is based on reducing the maximum index for any instruction in the schedule. *Incremental scheduling* is a code reordering technique, oriented to reduce the number of variables whose lifetime overlaps at

any cycle. Techniques similar to *incremental scheduling* have previously been proposed by other authors. However, *SPAN reduction* has been proposed here for first time.

We have derived different lower bounds on register requirements: An *absolute lower bound* for the number of registers required by any schedule of the loop and a *relative lower bound* for the number of registers required by any schedule of a given π-graph (without transforming the π-graph). Although similar ideas have previously been proposed by other authors, we propose here a new method for calculating such a lower bounds, more efficient than that used by other authors.

The obtained results show that *RESIS* is a good approach to reduce the number of registers required by a schedule. *RESIS* has also been used with success to optimize the schedules found by a modulo scheduling algorithm.

9.1.5 Time-constrained loop pipelining

TCLP, an algorithm for loop pipelining with timing constraints, is presented in Chapter 8. The algorithm calculates the lower bound on the number of resources of each type for a given timing constraint. Therefore, we can compare the obtained results with the theoretical optimal ones. Results show that the proposed approach works very efficiently, and it finds optimal results in most cases. Moreover, the proposed approach improves the results obtained by other techniques which do not consider loop pipelining.

Once a schedule has been found for the given timing constraint, the execution throughput is increased by using the current set of resources. This is done by exploring the solution space following an approach similar to the one used by *UNRET*. Finally, the number of required registers is reduced by using *RESIS*.

9.2 FUTURE RESEARCH

9.2.1 Decreasing the execution time

Although we have obtained very good results for most benchmarks, *UNRET* consumes a great deal of time pipelining some loops. This is due to the nature of the approach, and it may be a drawback to include *UNRET* in compilers for superscalar or VLIW processors.

In general, only a few configurations are explored before a schedule is found. However, in cases in which no schedule is found in the expected initiation interval for a given unrolling degree, a lot of configurations are explored before deciding that no schedule exists. This may be very time consuming, especially if the unrolling degree is quite high and several unrolling degrees are explored.

A possible solution consists of integrating the ideas behind *UNRET* in a modulo scheduling algorithm, perhaps also including a small amount of backtracking. Other authors have demonstrated that a small amount of backtracking does not consume too much time [Huf93, Rau94].

We believe that a modulo-scheduling mainly guided by positive and negative depth would probably obtain very promising results. After scheduling each instruction, the appropriate retiming transformation would be calculated, and the new depths would be assigned to the nodes. This differs from current modulo scheduling approaches in that nodes are dynamically selected by means of negative depth.

This approach differs from *UNRET*, in which scheduling is done before retiming for each node, and both scheduling and retiming are simultaneously done for the loop. Several configurations of the loop would be explored thanks to the backtracking, correcting some decision errors introduced by the heuristics used to select and schedule a node.

9.2.2 Span reduction and incremental scheduling at a time

Experiments have shown that *SPAN* reduction sometimes transforms the loop in such a way that the results obtained by incremental scheduling after *SPAN* reduction are worse than before. Although this happens in a reduced number of cases, this fact shows that it is not always useful to reduce the *SPAN*. We conclude that *SPAN* reduction must be selectively used, only in those nodes which are required to be reduced.

A way to do this is by giving to the incremental scheduling algorithm the ability of moving nodes across several schedules, instead of only within a schedule. A movement traversing the schedule would produce a reduction (or perhaps an increment) on the index of the node moved, and therefore it would change the *SPAN*. Since only the appropriate nodes would be moved, the *SPAN* would be selectively changed. We believe that this approach would obtain the best results, in addition to reducing the execution time of the algorithm to reduce registers.

9.2.3 Integer Linear Programming

Loop pipelining with resource constraints can be addressed by an integer linear programming (ILP) approach by employing some concepts proposed in this work.

Retiming has been modeled by equations which modify indices of nodes and distances of edges in the π-graph. The maximum value for the distance of any edge has also been calculated. Therefore, an ILP approach to explore all the equivalent configurations of a loop may be designed.

Scheduling can also be modeled by the ILP approach because the number of resources is known in advance.

A reduction in the number of registers required by the schedule can also be achieved by the ILP approach, since formulation is given to calculate the number of registers required by a dependence at any cycle. The cycle that requires more registers is that which determines the register requirements of the schedule.

ILP formulation may be oriented to both maximizing throughput (by exploring Farey's series) and minimizing register pressure. Moreover, the iteration time can also be minimized by reducing the *SPAN*.

9.2.4 Extension towards conditional sentences, while-like loops and multiple-nested loops

This work considers loop bodies composed only of basic blocks. Although most loops are of this type, loop bodies with conditional sentences are also common for some applications (for example, in the Perfect club benchmarks [BCKK88] there are a 6.75 per cent of loops[1] of this type). In the future we are interested in extending our model to permit the existence of conditional sentences, as well as to work with while-like loops.

The main difference between while-like loops and do-like loops is that the number of iterations is unknown at compilation time, and therefore a special code must be generated to detect when the execution of the loop has finished. A special code must also be generated to recover some results that may not be available due to the loop pipelining.

Finally, we are also interested in studying how multiple-nested loops can be pipelined. Other authors have previously studied this topic, but we believe that there is still much work to do. We are of the opinion that transformations such as *unroll and jam* [Car93] or *loop expansion* [Rim93] would be suitable for use with *UNRET* to exploit parallelism in multiple-nested loops.

[1] This number has been calculated after optimizing the loops. Only 85 from the 1285 maintain conditionals.