Constraint Satisfaction for Relative Location Assignment and Scheduling

Carlos Alba-Pinto [†], Bart Mesman [‡], and Jochen Jess [†]

† Eindhoven University of Technology, Design Automation Section
P.O.Box 513 - Room EH 9.23, 5600MB Eindhoven, The Netherlands
‡ Philips Research Laboratories and Eindhoven Embedded Systems Institute
E-mail: alba@ics.ele.tue.nl

Abstract

Tight data- and timing constraints are imposed by communication and multimedia applications. The architecture for the embedded processor imply resource constraints. Instead of random-access registers, relative location storages or rotating register files are used to exploit the available parallelism of resources by means of reducing the initiation interval in pipelined schedules. Therefore, the compiler or synthesis tool must deal with the difficult tasks of scheduling of operations and location assignment of values while respecting all the constraints including the storage file capacity. This paper presents a method that handles constraints of relative location storages during scheduling together with timing and resource constraints. The characteristics of the coloring of conflict graphs, representing the relative overlap of value instances, are analyzed in order to identify the bottlenecks for location assignment with the aim of serializing their lifetimes. This is done with pairs of loop instances of values until it can be guaranteed that all constraints will be satisfied. Experiments show that high quality schedules for kernels and inner loops can be efficiently obtained.

1. Introduction

The increment on the complexity of communication and multimedia applications, and their requirements of better performance, smaller area or lower power consumption, have motivated embedded system designers to search for architectural alternatives or optimizations in their synthesized architectures or embedded processors.

For architectures with high levels of parallelism, *pipelined schedules* are intended to achieve performance benefits. Unlike schedules in which one iteration of a loop is executed strictly after the execution of the previous one, pipelined schedules consist of multiple overlapping loop iterations with the aim of obtaining potentially much more efficient schedules [7], [5]. Iterations of a loop body are periodically initiated, in a period called the *initiation interval* (*II*), without having to wait for preceding iterations to complete. In pipelined schedules, consecutive instances of each value are generated every *II* clock cycles. If values would be assigned to random-access registers, it has to be ensured that any value instance has to be consumed before another instance of the same value is produced (in the next iteration). This means that a value (instance) cannot be alive longer than *II* clock cycles. It is therefore understood that if values would be assigned to conventional random-access registers the initiation interval is lower bounded by the longest value lifetime. On the other hand, with a defined *II*, values assigned to random-access registers have lifetimes upper bounded by the initiation interval.

In order to exploit better the available parallelism of resources by means of reducing the initiation interval, *relative location storages* for architectural synthesis [11], or *rotating register files* in embedded VLIW processors [9] were introduced. In such storage files, locations (registers) are addressed using a base plus offset model. The location address, specified in an instruction, is added to a *base* to derive the physical location address in the file, and the base is decremented each time a new iteration starts, therefore giving each loop iteration a distinct physical location.

Values can be mapped to distributed storage files each with limited *capacity* (number of locations). Therefore scheduling has to be performed respecting also the storage constraints. As a consequence, the following problem arises: perform scheduling and location assignment for relative location storages or distributed rotating register files with a limited capacity such that *timing* and *resource* constraints are also satisfied.

Traditional approaches deal with scheduling and location assignment in separate independent stages to reduce the complexity of these tasks [11], [9]. This approach introduces the problem of *phase coupling* between scheduling and location assignment: a decision made in one stage may lead to an infeasible constraint set for the other. Relative location assignment can not be performed before scheduling, since the value lifetimes depend on the time step (cycle) in which operations are scheduled, and also because the base is altered every period. On the other hand, if location assignment is performed after scheduling the number of required locations may exceed the capacity in a certain storage file. One solution could be to spill values to background memory like in [1]. When the location assignment violates the file's capacity, some values can be selected to be stored into a background memory. Load and store operations are inserted and all operations are rescheduled. However, the additional memory accesses easily cause timing constraint violations. Furthermore, although the separation of scheduling and location assignment may result in a run-time efficient method, this makes much more difficult to cope with the interaction of timing, resources, and capacity constraints all together.

This paper presents a method to combine relative location assignment and scheduling that overcomes the problem of phase coupling and avoids the difficulties of inserting spill code during architectural synthesis or code generation of algorithms. The method 'alternates' between scheduling and location assignment tasks by making a decision for location assignment and subsequently analyze how that prunes the search space for scheduling.

Potential conflicts between pairs of values before and during scheduling are analyzed. Using constraint analysis techniques essential information is used to identify the values that are bottlenecks for satisfying the storage file capacity. To reduce the identified bottlenecks, this method performs partial scheduling by serializing their lifetimes. Without enforcing any specific location assignment, the method continues until it can guarantee that any completion of the partial schedule will also result in a feasible location assignment. Therefore, it does not unnecessarily reduce the freedom to also satisfy resource and timing constraints.

This paper is organized as follows. Section 2 presents some basic definitions and assumptions. In Section 3 the problem statement is given, and the global solution strategy is proposed. The way conflict graphs are constructed and used in this approach is described in Sections 4 and 5 respectively. Lifetime serialization of values is described in Section 6. Finally, Section 7 presents some experimental results and Section 8 the conclusions.

2. Definitions and assumptions

An algorithmic description can be partitioned into basic blocks, each block is represented by a data flow graph [6], which describes the primitive operations performed and the dependencies between them.

Definition 1 *A data flow graph* $DFG = (\mathrm{V}(DFG), \mathrm{E}(DFG), \mathrm{W})$ *is a directed, edge-weighted and acyclic graph, where:*

- $\mathrm{V}(DFG)$ *is the set of vertices (operations),*
- $\mathrm{E}(DFG) = \mathrm{E}_d \cup \mathrm{E}_s$ *is the set of precedence edges,*
- $\mathrm{E}_d \subseteq \mathrm{V} \times \mathrm{V}$ *is the set of data edges (values),*
- $\mathrm{E}_s \subseteq \mathrm{V} \times \mathrm{V}$ *is the set of sequence edges,*
- $\mathrm{W} : \mathrm{E} \to \mathbb{Z}$ *and* $\mathrm{W} = \{w(O_1, O_2) | (O_1, O_2) \in \mathrm{E}(DFG)\}$ *is a function describing the timing delay (in clock cycles) associated with each precedence edge.*

For reasons of simplicity, it is assumed that all operations have an execution delay of 1 clock cycle. In [8] it is shown how pipelined and multi-cycle operations can be modeled using precedence constraints.

The task of scheduling is to assign each operation $O \in \mathrm{V}(DFG)$ a start time $s(O)$. Start times are constrained by the precedences. A precedence edge $(O_1, O_2) \in \mathrm{E}(DFG)$ states that $s(O_2) \geq s(O_1) + w(O_1, O_2)$. The interaction between several precedence constraints becomes clear by combining these precedence edges into a path.

A *path* of length d_{path} from operation O_1 to operation O_2 is a chain of precedences $O_1 \to ... \to O_2$ that imply $s(O_2) \geq s(O_1) + d_{path}$. The *distance* $\mathrm{d}(O_1, O_2)$ is the length of the longest path from operation O_1 to O_2. A path in the graph thus represents a minimum timing delay. The usual way of administrating these delays is by introducing a *distance matrix* that stores the length of the longest path between every pair of operations. The distance matrix is calculated using an all-pairs longest-path algorithm, an adaptation of the all-pairs shortest-path algorithm from [2].

A schedule also has to satisfy the resource constraints. In this approach, these constraints are modeled by introducing functional resources and associating a certain resource usage with each operation. Timing constraints are also considered, the *initiation interval II* for loops, and the *latency L*, i.e. the number of available clock cycles in which DFG has to be schedule.

By applying rules based on the information in the distance matrix, value lifetime conflict graphs are generated, colored and evaluated to find bottlenecks for storage satisfaction.

Definition 2 *A conflict graph* $CG(RRF) = (\mathrm{V}(CG), \mathrm{E}(CG))$, *for values bound to storage file RRF, is an undirected graph, where* $\mathrm{V}(CG)$ *is its set of vertices and* $\mathrm{E}(CG)$ *is its set of edges.*

Vertices in $\mathrm{V}(CG)$ represent the values bound to file RRF. There is an edge $(u^c, v^c) \in \mathrm{E}(CG)$, e.g. if lifetimes of values u and v overlap.

The *degree* of a vertex is the number of edges incident to it. A *clique* is a subset of vertices that induces a subgraph of CG in which those vertices are completely connected to

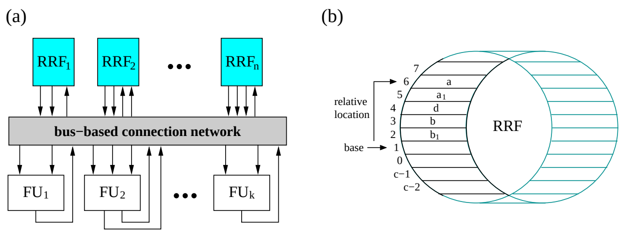

Fig. 1. (a) Architecture template, (b) storage file

each other by edges. The *clique number* $\gamma(CG)$ is the number of vertices of the maximum clique of CG.

Vertex coloring of a graph consists of assigning a color to every vertex so that no two vertices connected by an edge have the same color. *Exact coloring* consists of coloring using the minimum number of colors. The *chromatic number* $\chi(CG)$ is the smallest possible number of colors for coloring CG.

The *saturation* number of a vertex $v^c \in V(CG)$, in a colored graph CG, is the number of colors used by its neighbors. This number is an indication of the number of colors, and hence storages, used by a value v^c and its neighbors in CG. With a larger saturation number, more colors (storages) are used.

2.1. Storage model

The architecture shown in Figure 1a is assumed in this work. This consists on storage files, a bus-based connection network and functional units. No assumptions are made regarding the number of read and write ports of the storage files or the connection network. Nevertheless, port and connection constraints can be modeled as resource constraints and described in the input DFG. It is assumed that the delay for each data transfer is fixed.

In Figure 1b the assumed storage file is shown. The total number of locations of this file is c. Conceptually, the location address specified in an instruction, $I(u)$, (source or target) is added to the *base* to derive the physical location address in the file. The physical location address, $P(u)$, is the modulus of the previous addition and the storage capacity $c(RRF)$, i.e. $P(u) = (base + I(u)) \bmod c(RRF)$. Once a new iteration of the loop starts, the *base* is decremented.

3. Global approach

Given a binding of values to storage files, often implied by the assignment of operations to functional units, the problem statement is as follows:

Problem Definition 1 *Constrained Location Assignment and Operation Scheduling Problem. Given a data flow graph DFG, resource constraints, a binding of values to relative location storages or rotating register files, for each storage file RRF a capacity $c(RRF)$, an initiation interval II, and a latency L. Find an assignment of values to locations and a schedule that satisfy the precedence constraints; the resource constraints; the storage capacity; and the timing constraints II and L.*

Because decisions have to be made that affect the search space in both the domain of location assignment and the domain of scheduling, the problem is decomposed into steps as depicted in Figure 2. The main part, the *constraint analysis* [8], generates additional precedence constraints that are implied by the combination of all constraints. These additional precedences refine the distance matrix thus providing a more accurate estimate of the set of feasible start times. It consists of different analysis methods like the *execution interval analysis* [10].

After constraint analysis, for distributed storage files, a storage file is selected and the values bound to it are used in the following analysis. For the selected file, the worst case or upper bound of required locations is computed. This, by means of using a worst-case conflict graph and its exact coloring. Also, a lower bound of the number of locations is determined through the coloring of a best-case conflict graph, and compared with the respective file's capacity. A lower bound violation indicates an infeasible case.

When the worst case storage requirements for each storage file already respects their capacity, the DFG with the additional precedences is transferred to a conventional scheduler that completes the schedule and then the final location assignment. However, in most cases and especially at the beginning of the process, the mobility of the operations can be relatively large resulting in many potential *conflicts*, hence inevitably violating some storage file capacity constraint.

The approach in Figure 2 has to reduce the maximum number of conflicts by identifying one or more pairs of values that can potentially share an address (*bottleneck identifi-*

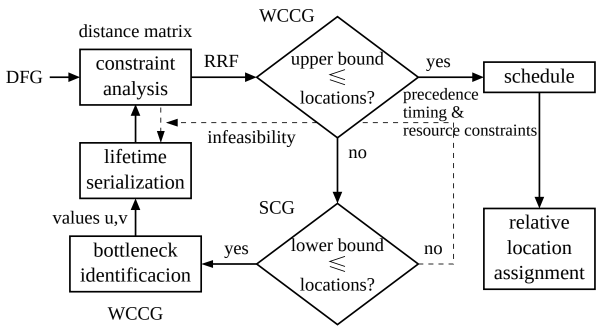

Fig. 2. Global approach

cation) and then proceed to serialize their lifetimes (*lifetime serialization*). The constraint analysis calculates the effect of serialization on the schedule freedom of all operations. This is necessary to guide the process to avoid making decisions that are not possible.

The upper and lower bounds checking, the bottleneck identification, the lifetime serialization, and the constraint analysis keep on alternating until the capacity constraint of each storage file matches the worst case requirements. An advantage of this approach is that in practice any conventional scheduler can be used to complete the schedule.

4. Constructing conflict graphs

This section shows how potential storage conflicts between pairs of values can be analyzed before a complete schedule is known. This analysis uses the distance matrix to determine the worst- and the best-case conflicting situations between value lifetimes. Potential conflicts are used to identify and solve bottlenecks when some storage file is in danger of being overloaded. Unlike traditional methods, this method does not require that all operations are scheduled when constructing conflict graphs.

4.1. Multiple value instances

In pipelined schedules, value lifetimes can not only overlap with others from the same loop iteration, but depending on the initiation interval, lifetimes of value instances from different loop iterations can overlap. Therefore, several instances of a value mapped to relative locations or a rotating register file can co-exist in some clock cycle (see Figure 4a).

Since the location requirement is modeled with conflict graphs for this approach, value instances from successive loop iterations require a corresponding vertex representation in the conflict graphs. The number of value instances represented in the graph is related to the latency and the initiation interval constraints:

$$number\_instances = \min\left(c(RRF), \left\lceil \frac{L-1}{II} \right\rceil + 1\right) \quad (1)$$

A proof for this expression is presented in Appendix A.

4.2. Conflict rules

Because lifetimes are not fixed yet, three different situations may exist between two value instances: a *no conflict*, a conflict for sure or *strong conflict*, or a *weak conflict*.

The essential difference between a strong and a weak conflict is that instances with a strong conflict can never reside in the same location, but the ones with a weak conflict

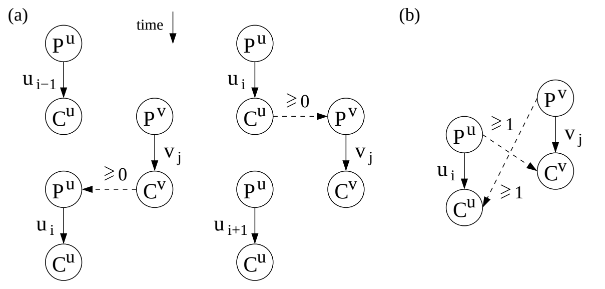

Fig. 3. u_i and v_j have (a) no conflict, (b) a strong conflict. Dashed edges represent the distances derived from Propositions 1 and 2 respectively

have lifetimes that can still be serialized. Lifetime serialization does reduce schedule freedom, since the mobility of individual operations is affected. Therefore, instances to serialize must be selected carefully, such that the number of weak conflicts in a potentially overloaded storage file is reduced, and not too much schedule freedom is sacrificed to obtain that goal. For that purpose, it is convenient to have a clear criterion for each of the three possible situations between value instances.

Consider u_i and v_j instances of values u and v in iterations i and j respectively. u_i and v_j are produced by operations P^u_i and P^v_j and consumed by C^u_i and C^v_j respectively. For non-folded cases $i = j = 0$.

No conflict. Instances u_i and v_j have no conflict if their lifetimes do not overlap, i.e. $d(C^u_i, P^v_j) \geq 0$ or $d(C^v_j, P^u_i) \geq 0$. The no conflict situation is depicted in Figure 3a and, because the information from the distance matrix is used (containing distances between operations from the same iteration 0), this is formalized as follows:

Proposition 1 *Instances u_i and v_j have no conflict if* $d(C^u, P^v) - (i-j) * II \geq 0$ *or* $d(C^v, P^u) - (j-i) * II \geq 0$.

Strong conflict. Instances u_i and v_j have a strong conflict if their lifetimes overlap for sure, i.e. $d(P^u_i, C^v_j) \geq 1$ and $d(P^v_j, C^u_i) \geq 1$. This is depicted graphically in Figure 3b and, because the use of the distance matrix, this is formalized as follows:

Proposition 2 *Instances u_i and v_j have a strong conflict if* $d(P^u, C^v) - (i-j) * II \geq 1$ *and* $d(P^v, C^u) - (j-i) * II \geq 1$.

Additionally, two instances from the same value require different locations, because the accessing mechanism of a relative location file assigns different locations for each value instance.

Proposition 3 *Instances u_i and u_j always have a strong conflict.*

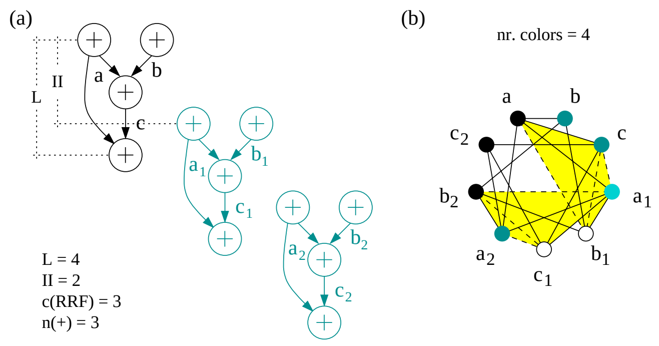

Fig. 4. (a) *DFG*. **(b)** *WCCG*. **Dashed edges represent weak conflicts while solid ones strong conflicts**

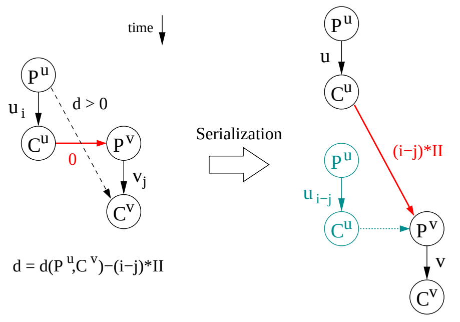

Fig. 5. Lifetime serialization of u_i and v_j. Distance $d > 0$ **determines serialization** $u_i \rightarrow v_j$

Weak conflict. Two value instances u_i and v_j have a weak conflict if Propositions 1 for no conflict, and Propositions 2 and 3 for strong conflicts, are invalid.

Consider the precedence graph of Figure 4a with a latency of 4 and an initiation interval of 2, e.g. instances b_1 and c_1, from the same iteration, have no conflict; instances a and b have a strong conflict in the clock cycle that the operation that consumes both executes; and instances c and a_1 have a weak conflict because it is not yet determined whether their lifetimes overlap or not.

5. Use of conflict graphs

Two different conflict graphs are used in this approach. The first is the *worst-case conflict graph* $WCCG(RRF)$ that represents the case when value lifetimes have the worst-case conflicting situation. This graph groups weak and strong conflicts together. In Figure 4b, a worst-case conflict graph corresponding to the data flow in Figure 4a is shown. The number of instances considered per value is 3 according to Expression 1. Coloring this graph results in a chromatic number of 4, while the storage file capacity is 3, therefore some lifetime serialization has to be performed. The second graph is the *best-case* or *strong conflict graph* $SCG(RRF)$ and represents the case when values have strong conflicts. In Figure 4b the strong conflict graph is represented only by solid edges and its chromatic number is 3.

Both graphs are colored exactly using the simplified coloring algorithm presented by Coudert in [3]. After coloring, the chromatic number of *WCCG* gives an *upper bound* which is checked with the actual capacity of the storage file, while the chromatic number of *SCG* gives a *lower bound* of the amount of locations required. Since value lifetimes are altered with serialization, new strong conflicts can arise, and new lower bounds are obtained. If the updated lower bound is larger than the storage file capacity, an infeasibility is returned as result of the last decision.

The choice of the bottlenecks is based on the largest *saturation* number (see Section 2) of a vertex (value instance) in the colored *WCCG*. Since coloring *WCCG* gives a worst-case situation, vertices with maximum saturation represent the most-location demanding group of value instances. Lifetime serialization of two value instances takes away one or more weak conflicts. When they have the maximum saturation their serialization can potentially reduce the chromatic number of *WCCG* (and hence the number of locations required). Because there might be a lot of value instances with the same maximum saturation, the largest *degree* number in *WCCG* is used as second criterion.

In the example of Figure 4b, instances a, c, a_1, b_1, c_1, a_2 and b_2 are candidates for lifetime serialization and were detected by their saturation number of 3 which is the largest in the graph. Then, instance a_1 is chosen first because its degree number of 6, and instance c because it has a weak conflict relation with a_1.

6. Lifetime serialization

Suppose that the decision is made that lifetimes of u_i and v_j are to be serialized. Serialization consists of the insertion of sequence edges (with weight 0) between consumers of one value and the producer of the other, taking into consideration the actual distances between producers and consumers of values u and v and the iteration numbers i and j.

There are two possibilities that lifetimes can be serialized: with sequence edge $C_i^u \rightarrow P_j^v$ or $C_j^v \rightarrow P_i^u$. Often, the distance relations between producers and consumers exclude one possibility. In Figure 5 a distance from producer to consumer $d(P_i^u, C_j^v) > 0$ determines the only possible solution $C_i^u \rightarrow P_j^v$, a sequence edge with weight $w(C^u, P^v) = (i - j) * II$ is added in the data flow.

After serialization, if the constraint analysis or the lower

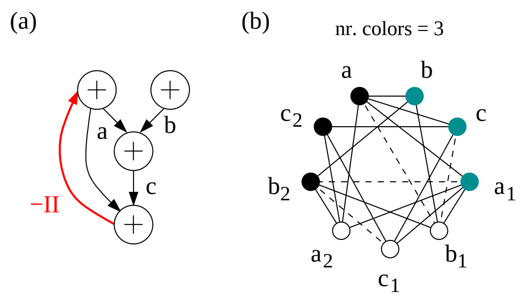

Fig. 6. Resulting (a) *DFG* and (b) *WCCG* graphs after serialization $c \to a_1$

bound checking detects infeasibility as a result of this choice, the other serialization is tried if possible. If both alternatives lead to infeasibility, serialization of u_i and v_j is discarded and another pair of values is chosen to repeat the process. After that, the effect on the storage file pressure is checked using an updated conflict graph with a new coloring.

Figure 6 shows the result from lifetime serialization of c and a_1. The updated worse-case conflict graph in Figure 6b requires only 3 colors, which matches the capacity of 3.

7. Experimental Results

All experiments were run on a Pentium II processor machine running at 350 MHz. Instances of the following examples were used: a fast Fourier transform (fft), an inverse fast Fourier transform example generated by Mistral2 at Philips (ifft), a finite impulse response filter (fir), an infinite impulse response filter (iir), and a loop body of an industrial example (loop). Instances of these examples differ in the number of resources and timing constraints. Only one storage file is considered for each example.

The main characteristics of the examples are shown in Table 1 respectively: the example name, the number of vertices and edges, $|FU|$ which is a reference about the number of functional units (resources) available of each modulo type, the minimum obtainable latency and initiation intervals with that number of resources available, the capacity required c_{req} as result from a branch-and-bound scheduler [10] and a relative location assignment [11], and the CPU time in seconds.

To evaluate the proposed method, it was applied to the instances of Table 1. The branch-and-bound scheduler was used to complete the partial schedule resulting from the satisfaction processes. The mobility is defined as the average difference between the ALAP and ASAP start times of operations: $\frac{1}{|V|}\sum_{u\in V} \mathrm{ALAP}(u) - \mathrm{ASAP}(u)$ and is a good indication of the schedule freedom. The number before respectively after the arrow denote the mobility before and after the satisfaction process.

DFG	$\lvert V\rvert, \lvert E_d\rvert$	$\lvert FU\rvert$	L,II	c_{req}	time(s)
fft	30,43	1	13,4	11	0.09
		2	11,2	18	0.09
fir	16,11	1	6,3	6	0.01
ifft	73,82	2	36,26	13	1.73
iir	27,21	1	9,4	9	0.05
loop	30,34	1	11,4	17	0.07
		2	7,2	24	0.07

Table 1. Examples and reference results

$DFG_{L,II}$	c_{req}	$c(RRF)$	time(s)	mobility
fft$_{13,4}$	11	9	0.40	$3.10 \to 0.00$
fft$_{11,2}$	18	17	1.06	$2.17 \to 0.00$
fir$_{6,3}$	6	6	0.05	$1.44 \to 0.48$
ifft$_{36,26}$	13	12	2.46	$13.9 \to 4.16$
iir$_{9,4}$	9	9	0.18	$1.59 \to 0.33$
loop$_{11,4}$	17	16	263.	$4.40 \to 1.00$
loop$_{7,2}$	24	15	0.54	$2.00 \to 0.60$

Table 2. Results for relative storage assignment

The results from Table 2 show the advantages of this approach dealing with relative location storages or rotating register files: By taking the storage file capacity into account, this method is able to reduce the storage pressure compared to an approach that performs location assignment a posteriori. For instance loop$_{7,2}$, this results in a reduction from 24 to 15 (columns c_{req} and $c(RRF)$ of Table 2) in the total number of locations.

8. Conclusions

This paper presents an approach for relative location assignment and scheduling in the context of using relative location storages or rotating register files in a synthesized architecture or in an embedded processor. Storage file constraints are taken into account from the first phase of scheduling, while there is enough freedom to reduce storage pressure.

Constraint analysis techniques are used to capture the interaction between precedence, timing and resource constraints. By constructing a conflict graph that models the strong and weak conflicts between value accesses, the bottlenecks for location assignment are identified. These bottlenecks are subsequently reduced by serializing their lifetimes. This results in a partial schedule that can be completed by a conventional scheduler without violating the storage file constraints.

The results in Section 7 show that this method is able to satisfy storage file requirements under tight timing and resource constraints. The method provides a good balance between solution quality and run time. Although the problem of spilling values to background memory has not been addressed, the proposed method can help to avoid unnecessary spill code.

References

[1] G. Chaitin. Register allocation & spilling via graph coloring. In *Proc. of the ACM SIGPLAN'82 Symposium on Compiler Construction*, pp. 98–105, Boston, Jun 1982.

[2] T. Cormen, et al. *Introduction to Algorithms*. MIT Press, Cambridge, 1994.

[3] O. Coudert. Exact coloring for real-life graphs is easy. In *Proc. of the 34th ACM/IEEE Design Automation Conference*, pp. 121–126, Anaheim, Jun 1997.

[4] M. Garey and D. Johnson. *Computers and Intractability. A guide to the Theory of NP-Completeness*. W. Freeman, San Francisco, 1979.

[5] G. Goossens, et al. Loop optimization in register-transfer scheduling for dsp-systems. In *Proc. of the 26th ACM/IEEE Design Automation Conference*, pp. 826–831, Las Vegas, Jun 1989.

[6] D. Ku and G. D. Micheli, editors. *High-level Synthesis of ASICs under Timing and Synchronization Constraints*. Kluwer Academic Publishers, Dordrecht, 1992.

[7] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, pp. 318–328, Atlanta, Jun 1988.

[8] B. Mesman, et al. Constraint analysis for dsp code generation. *IEEE Transactions on Computer-Aided Design*, 18(1):44–57, Jan 1999.

[9] M. Schlansker, et al. Achieving high levels of instruction-level parallelism with reduced hardware complexity. Technical Report HPL-96-120, Hewlett Packard, Nov 1994.

[10] A. Timmer, et al. Conflict modeling and instruction scheduling in code generation for in-house dsp cores. In *Proc. of the 32nd ACM/IEEE Design Automation Conference*, pp. 593–598, San Francisco, Jun 1995.

[11] J. van Meerbergen, et al. Relative location assignment for repetitive schedules. In *Proc. of The European Conference on Design Automation with The European Event in ASIC Design*, pp. 403–407, Paris, Feb 1993.

A. Number of value instances in conflict graphs

Proof. Consider a data flow graph DFG with a latency L, an initiation interval II, and a storage file RRF with capacity $c = c(RRF)$. The number of value instances is always upper bounded by c since no more than c instances can be stored in the file. When c is larger enough, a tighter upper bound is obtained: $\lceil (L-1)/II \rceil + 1$, which is related to the maximum number of instances of one value that can be produced during latency L. This is proven by the following steps:

According to van Meerbergen in [11], given L_{low}, the lower bound of locations considering lifetime overlaps, the required number of relative locations L_{rel} satisfies: $L_{rel} \leq L_{low} + 1$ (a)

When using a graph CG to calculate the number of locations, its coloring results in: $\chi(CG) = L_{rel}$ (b)

For any graph, the clique number satisfies ([4]): $\gamma(CG) \leq \chi(CG)$ (c)

The number of instances of any value u, in a maximum clique of CG, also satisfies: $number\_instances_u \leq \gamma(CG)$ (d)

From (a), (b), (c), and (d): $number\_instances_u \leq L_{low} + 1$ (e)

Consider now, value u being produced by operation P^u, consumed by operation C^u, and has a maximum possible lifetime of $lifetime_u = \mathrm{d}(\mathrm{P}^u, \mathrm{C}^u) = L - 1$. The lower bound of the number of locations required by u is equal to the number of overlaps among its instance lifetimes, i.e. $L_{low}(u) = overlaps_u = \lceil lifetime_u / II \rceil = \lceil (L-1)/II \rceil$, and satisfies: $L_{low}(u) \leq L_{low}$. In a particular case, overlaps of u lifetimes can determine the lower bound of required locations, i.e. $L_{low}(u) = L_{low}$. Therefore: $\lceil (L-1)/II \rceil = L_{low}$ (f)

From (e) and (f): $number\_instances_u \leq \lceil (L-1)/II \rceil + 1$. Finally, being conservative, the number of u instances required in CG is equal to $\lceil (L-1)/II \rceil + 1$ ■