A Reordering Technique for Efficient Code Motion

Luiz C. V. dos Santos and Jochen A. G. Jess
Design Automation Section, Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{luiz, jess}@ics.ele.tue.nl

Abstract

Emerging design problems are prompting the use of code motion and speculative execution in high-level synthesis to shorten schedules and meet tight time-constraints. However, some code motions are not worth doing from a worst-case execution perspective. We propose a technique that selects the most promising code motions, thereby increasing the density of optimal solutions in the search space.

1 Introduction

The combination of intensive data-flow, complex control-flow and tight time-constraints creates design problems requiring multiple functional units. The usual scope for exploitation of parallelism is the *basic block* (BB) [2]. Since the parallelism in a BB is limited, the multiple functional units are poorly utilized. This prompts the use of instruction-level parallelism (ILP) techniques [2] in high-level synthesis (HLS), for instance by moving operations across BB boundaries, which is called *code motion.* Code motions ahead of branches may lead to *speculative execution.* Early ILP compiler techniques [2] are oriented to architectures with abundant resources. In HLS, however, ASICs and ASIPs are designed with as few resources as possible. Consequently, the direct application of those techniques may expose too much parallelism. Relaxing control dependences for the sake of global scheduling, leads to a trade-off: on the one hand, it improves exploitation of parallelism; on the other hand, it increases the size of the solution space. Therefore, we propose a technique that prevents the generation of inferior solutions (Section 3). We show experimental evidence that our technique increases the density of optimal solutions in the search space, paving the way to a faster exploration of alternative solutions (Section 4).

2 Our modeling

To represent behavior, we use a *data flow graph* (DFG). An example of DFG is shown in Figure 1b for the description in Figure 1a. Circles represent operations. Pentagons denote either *branch* (B) or *merge* (M) nodes controlled by a *conditional* (c_k). See [4] for an explanation on DFG semantics.

To keep track of code motion, we use a condensation of the DFG, the so-called *basic-block control flow graph* (BBCG). In the BBCG, the nodes represent BBs or junctions (either a *branch* (B) or a *merge* (M)) and the edges represent the flow of control. All operations in the DFG enclosed by a pair of branch, merge, input or output nodes are condensed into a BB in the BBCG. All branch (merge) nodes in the DFG controlled by the same conditional become a single branch (merge) node. All inputs are contracted to a single *source* node; all outputs, to a *sink* node. Given the DFG in Figure 1b, its BBCG appears in Figure 1c.

The relation between a DFG and its BBCG is kept by means of *links.* A link λ connects an operation o_n in the DFG with a basic block BB_i in the BBCG, written $o_n \xrightarrow{\lambda} BB_i$ (e.g. each arrow represents a link in Figure 1c). When a link λ points to the BB where the operation was initially described, we say that λ is an *initial link.* We write $o_n \not\rightarrow BB_i$ when o_n is *not* linked to BB_i and we write $BB_i \xrightarrow{*} BB_j$ when there is a path from BB_i to BB_j.

Given a DFG and a set of resource constraints, our goal is to derive a *state machine graph* SMG $= (S, T)$, where S is set of states, and T is the set of transitions. Figures 1d and 1e show alternative SMGs for the DFG in Figure 1b, assuming 1 adder, 1 subtracter and 1 comparator. For Figure 1d, exploitation of ILP is limited to BBs, whereas code motion is used for Figure 1e. We assume that every operation o_n is mapped to a single module type $\tau(o_n)$ (e.g. $\tau(m) = adder$ in Figure 1). When two operations available for scheduling at a given state map to the same module type, a priority encoding is used to break ties. A priority encoding Π is a permutation of the operations in the DFG. When operation o_n precedes operation o_m in a permutation Π, written $o_n \prec_\Pi o_m$, then o_n has the priority over o_m.

In our approach, solutions are encoded by priority encodings. An *explorer* creates priority encodings Π and a *constructor* builds a solution for each Π and evaluates its cost, i.e. the schedule length of the longest path in the SMG. The explorer searches for the solution with lowest cost and checks if it satisfies a time constraint T_c. The constructor consists of a *scheduler* and a *parallelizer.* The parallelizer manages code motion and speculation and assigns operations to states while the SMG is generated on the fly. Given a *current state* s_k, the parallelizer keeps a set A_k of *available* (or ready) operations [1] for scheduling in state s_k. From A_k, the scheduler selects an operation o_n for executing in state s_k. After scheduling s_k, *next states* are scheduled and so on. In a top-down traversal of the BBCG, it is as if each BB were split on the fly into a sequence of successive states. Every set A_k is ordered by the *same* priority encoding Π. Given a state s_k and an ordered set A_k, the scheduler selects the first operation $o_n \in A_k$ satisfying resource constraints.

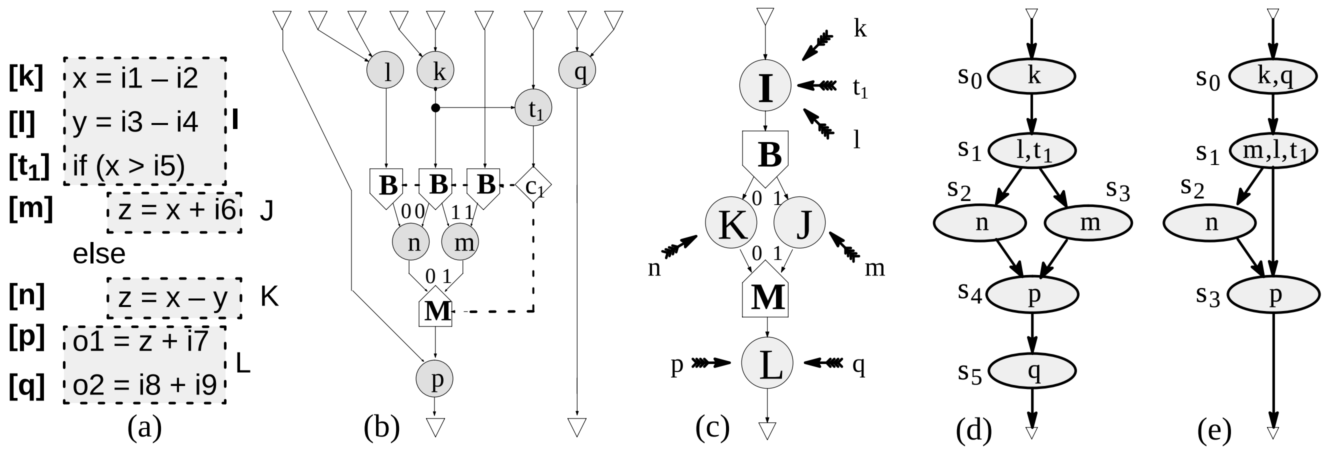

Figure 1: A behavioral description, its DFG, BBCG and resulting SMGs

3 Preventing inefficient code motions

Figure 2 shows a behavioral description, its BBCG and two SMGs obtained assuming 1 adder, 1 subtracter and 1 comparator. Since operations a and b depend only on the inputs, they are both available at state s_0 within BB I, i.e. $A_0 = \{a, b\}$. Solutions SMG_1 and SMG_2 are induced by Π_1 (with $b \prec_{\Pi_1} a$) and by Π_2 (with $a \prec_{\Pi_2} b$), respectively. Note that the schedule length of the left path is increased by the execution of b in the first state of SMG_1, as compared to SMG_2. The reason is that a and b can not be scheduled in a same state because $\tau(a) = \tau(b) = subtracter$. Given the encoding Π_1 in Figure 2c we have $A_0 = (b, a)$. If we *reorder* this set such that $A_0 = (a, b)$, the code motion of b that places it in state s_0 is prevented and solution SMG_1 (which is inferior) is not constructed. The example illustrates that the parallelism actually exploitable is hinted by a convenient interpretation of the links: given two available operations with same τ, the execution of the operation that is linked to the currently visited BB should have the priority over the other. However, this is valid only if we can guarantee that some operations must execute within a given BB, while others can be postponed. This requires a data-flow analysis technique which finds a so-called set of *lowest links* from the set of initial links [7], as summarized next.

3.1 The notion of lowest links

Given an operation o_m and its initial link, each lowest link is determined such that it points to the latest BB downwards on a given control path, where o_m can be legally executed. Our analysis checks if an operation o_m can be legally moved *down* from BB_i to BB_j, i.e. o_m should not move if it kills a value [7], no conditional should move past the point where the control decision is due and no operation should move through the sink, as formalized next:

Definition 1. Given the basic blocks BB_i and BB_j with $BB_i \overset{*}{\to} BB_j$, the operation o_m is *free* to move from BB_i to BB_j, written $free(o_m, BB_i, BB_j)$, iff: $\neg kill(o_m, BB_i, BB_j) \wedge (o_m \neq conditional) \wedge (BB_j \neq sink)$.

Algorithm 1 derives the set of *lowest* from the set of *initial* links. $CONS(o_m)$ represents the operations consuming the value produced by o_m. The DFG is visited by depth-first search. Procedure visit(o_m) propagates an operation o_m from its initial BB towards the initial BB of some consumer o_n. Procedure find_boundary(o_m, BB_i, BB_k) returns the latest BB to which o_m is free to move on control paths from BB_i to BB_k.

Algorithm 1. Finding the lowest links
```
procedure find_lowest_links ()
mark initial links;
foreach input $v_i$ of DFG
    foreach $o_m \in CONS(v_i)$
        visit ($o_m$);
delete initial links;

procedure visit ($o_m$)
if  ($o_m$ is visited) return;
mark $o_m$ as visited;
foreach $o_n \in CONS(o_m)$
    visit ($o_n$);
    foreach $BB_i$ with $o_m \overset{\lambda_i}{\to} BB_i$
        foreach $BB_k$ with $o_n \overset{\lambda_k}{\to} BB_k$
            $BB_j$ := find_boundary ($o_m, BB_i, BB_k$);
            create a new link $\lambda$ such that $o_n \overset{\lambda}{\to} BB_j$;
```

The lowest links capture the maximal freedom for moving operations downwards, as formalized next.

Theorem 1. If there is a path p from the source to BB_j such that operation o_n was not scheduled within any BB in path p, and if there exists a lowest link connecting operation o_n with BB_j, then o_n must be scheduled within BB_j.
Proof: This theorem is proven in [7].

3.2 A precedence relation based on the lowest links

If operation a is linked to the BB being visited, say BB_i, and operation b is linked to BB_j, with $BB_i \overset{*}{\to} BB_j$, then a *must* be scheduled in BB_i (Theorem 1), whereas the scheduling of b can be postponed, since b is free to move down to BB_j. This suggests that a should have the precedence over b. Precedence is defined only between operations with same module type, since only in that case their parallel execution might be impaired due to the lack resources, as follows:

Definition 2. Given the set of lowest links Λ and some $\lambda \in \Lambda$, let $\prec_{\Lambda,i}$ denote the *precedence relation induced by the set of links* Λ when the basic block BB_i is visited. We say that a precedes b at basic block BB_i, written $a \prec_{\Lambda,i} b$, iff: $(\tau(a) = \tau(b)) \wedge ((a = b) \vee (a \overset{\lambda}{\to} BB_i) \wedge (b \not\to BB_i)$.

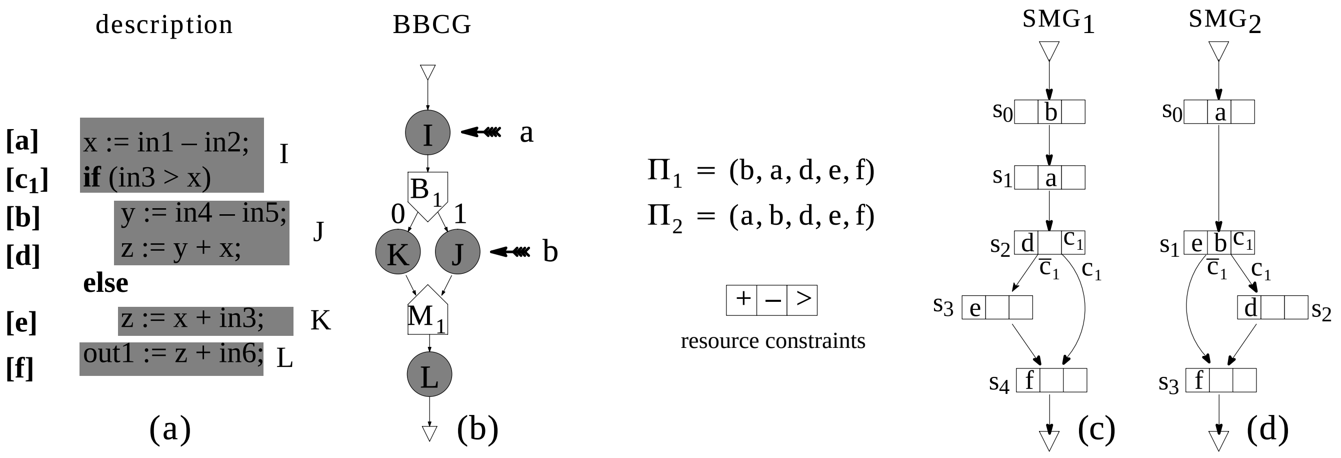

Figure 2: Example of code motion ahead of branch junction

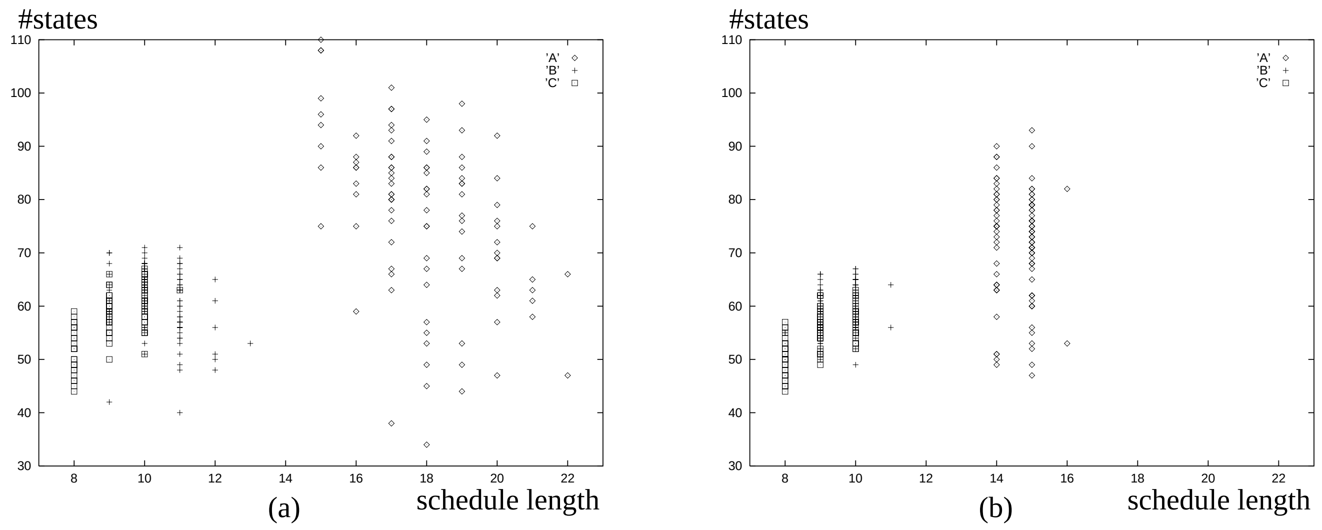

Figure 3: The search space without and with reordering, respectively

3.3 Reordering the sets of available operations

Given two operations a and b, our idea is to make the precedence relation $\prec_{\Lambda,i}$ prevail over the priority encoding either when $a \prec_{\Lambda,i} b$ or when $b \prec_{\Lambda,i} a$. Otherwise, the ordering implied by $\prec_{\Pi}$ is kept. This is formalized by a new linear ordering $\prec_{\Pi,i}$ for each BB_i, as follows:

$$a \prec_{\Pi,i} b \iff (a \prec_{\Lambda,i} b) \vee (\neg(b \prec_{\Lambda,i} a) \wedge (a \prec_{\Pi} b)). \quad (1)$$

The linear order $\prec_{\Pi,i}$ is the modification of the original linear order $\prec_{\Pi}$ so as to capture the results of our data-flow analysis. Suppose that BB_i is visited and, given the current state s_k within BB_i, let $a, b \in A_k$. Assume that both a and b map to module type τ and are such that $b \prec_{\Pi} a$, but $a \prec_{\Pi,i} b$. When we order the set A_k according to $\prec_{\Pi,i}$, instead of $\prec_{\Pi}$, the scheduler assigns a to state s_k and the code motion of b to BB_i is prevented if *no more resources of type τ are free* within BB_i. Thus, reordering A_k effectuates the pruning of code moves.

4 Experimental results

Our experiments are performed *with* and *without* reordering under largely unrestricted code motion and speculation and for several priority encodings generated randomly. Let (L_i, S_i) be the i^{th} solution in the search space, where L_i is the schedule length of the longest path in the SMG and S_i is the number of states. Let $\Delta L = max(L_i) - min(L_i)$ be the *range* of observed schedule lengths. We apply 100 priority encodings to example "s2r" and we construct 100 SMGs for different resource constraints (cases A, B and C in Table 1). The plot in Figure 3a, where each point represents a SMG, shows the effect in the search space *without* reordering. Observe that ΔL increases when the number of resources decreases. This means that exposed parallelism which is not accommodated leads to many inferior solutions. By repeating the same experiments *with* our reordering, the plot in Figure 3b is obtained. Compare Figures 3a and 3b, and note that inferior solutions are pruned, since ΔL is reduced from 3 to 2, for case C; from 4 to 3, for case B and from 7 to 2, for case A.

Table 1 reports statistics on L_i for several examples and resource constraints. The minimal value of L_i, its mean value and its standard deviation (σ) are given, along with the percentage of solutions with minimal L_i, i.e. the *density of optimal solutions*. Note that with reordering the mean value either decreases or remains the same and σ decreases for most examples. Therefore, our technique not only shortens schedules on average, but also spreads them over a smaller range of lengths. Observe that reordering increases the density of optimal solutions in the search space,

Table 1: Impact on schedule length L_i without and with reordering

example	case	resource constraints					min	without			with		
		alu	add	sub	mul	cmp	L_i	mean	σ[%]	density	mean	σ[%]	density
kim1	B	0	1	1	0	1	8	9.0	5.3	10%	8.8	4.6	23%
	C	0	2	1	0	1	6	7.0	6.6	7%	6.9	5.7	12%
rotor	A	1	0	0	0	0	11	13.6	12.8	12%	11.0	0.0	100%
	B	2	0	0	0	0	8	8.3	5.5	67%	8.0	0.0	100%
	E	1	0	0	2	0	9	10.6	5.4	2%	9.8	3.6	15%
	F	2	0	0	2	0	8	8.0	0.0	100%	8.0	0.0	100%
s2r	A	1	0	0	0	0	14	18.0	9.5	1%	14.7	3.5	41%
	B	2	0	0	0	0	8	10.4	7.8	1%	9.5	6.1	16%
	C	3	0	0	0	0	8	9.1	9.0	41%	8.9	8.7	62%
	E	1	0	0	2	0	12	15.0	8.4	1%	13.1	5.7	25%
	F	2	0	0	2	0	8	10.3	8.2	2%	10.0	7.6	16%
	G	3	0	0	2	0	8	9.7	11.4	31%	9.4	11.0	45%
kim2	A	0	1	1	1	1	49	64	6.2	0%	59	4.5	0.1%
	C	0	1	2	1	1	49	63	5.1	0%	59	4.3	0.1%
	D	0	1	1	2	1	47	61	5.9	0%	58	5.0	0.1%

except for a few examples where it remains the same.

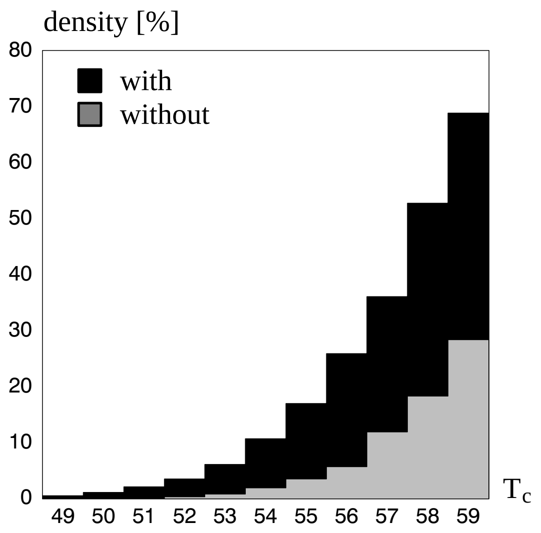

Figure 4: Density of solutions satisfying a given T_c

For the example "kim2", case D, Figure 4 shows the density of solutions satisfying a given time constraint T_c, i.e. the percentages obtained by adding the densities of all solutions with $L_i \leq T_c$. Note that, when $T_c = 56$, 5% of the solutions without reordering satisfy the time constraint, against about 25% with reordering. About 70% of the solutions observed without reordering have $L_i > 59$, whereas only 30% of such inferior solutions are observed with reordering. In summary, without reordering, more solutions have to be explored on average to meet a given T_c.

5 Related Work and Conclusions

Our method is more conscious of data-flow properties than related methods such as [8] [6], which assign priorities to operations according to their *initial* BBs, but do not check if the operations could move further down. In [6] the precedence of operations linked to BBs on a same path is enforced, *regardless of which BB is currently visited*. In our technique, we use distinct precedence relations for each visited BB depending on the *current* links. Tree-Based Scheduling (TBS) [5] casts the control flow into trees. Operations are propagated towards the leaves within a *single* tree. Unlike TBS, our method is *global*: downward code motion is not restricted to nested conditional trees. TBS propagates operations down to remove redundant operations on a path, instead of our more general notion of maximal freedom for downward code motion, which also supports the execution of operations more than once on a same path [7]. Path-Based Scheduling has been extended to relieve the required fixed order [3], but its reordering is performed inside BBs only, improving the utilization of parallelism within a BB, but limiting the exploitation of parallelism with complex control flow, since speculative code motions are not allowed.

Our method is designed to avoid the greed of classical built-in scheduling heuristics. Although optimal solutions may still be overlooked, it increases the density of solutions satisfying a time constraint. Our reordering seems affordable: in Equation 1, the test $a \prec_{\Pi} b$ takes constant time and the test $a \prec_{\Lambda,i} b$ may require the enumeration of all links from a and b, typically a small fraction of the total number of operations. Since our reordering improves the quality of global scheduling and relies on an efficient test on precedence, it seems a viable option for a HLS tool.

References

[1] A. Aiken et al., "Resource-Constrained Software Pipelining", IEEE Trans. Parallel and Distributed Syst., vol. 6(12), pp. 1248-1270, Dec. 1995.

[2] U. Banerjee et al., "Automatic Program Parallelization", Proc. of the IEEE, vol. 81(2), pp. 211-243, Feb. 1993.

[3] R. Bergamaschi et. al.,"Control-Flow Versus Data-Flow Based Scheduling: Combinining Both Approaches in an Adaptive Scheduling System", IEEE Trans. on VLSI Systems, vol. 5, no.1, pp.82-100, March 1997.

[4] J. van Eijndhoven and L. Stok, "A Data Flow Exchange Standard", Proc. Europ. Conf. Design Automation, pp. 193-199, 1992.

[5] S.Huang et al.,"A tree-based scheduling algorithm for control dominated circuits", Proc. ACM/IEEE Design Automation Conference, pp. 578-58, 1993.

[6] S.-M. Moon and K. Ebcioglu, "An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors", Proc. Int. Simp. on Microarchitecture, pp. 55-71, 1992.

[7] L. C. V. dos Santos, "Exploiting instruction-level parallelism: a constructive approach", PhD Thesis, Eindhoven University of Technology, The Netherlands, November, 1998.

[8] M. Smith et al., "Efficient Superscalar Performance Through Boosting", Proc. Int. Conf. Archit. Support for Prog. Lang. and Operating Syst., pp. 248-259, 1992.