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 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). 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 BCG appears in Figure 1c.

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

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 subtractor 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 \( o_k \) is mapped to a single module type \( \tau(o_k) \) (e.g. \( \tau(m) = \text{adder} \) in Figure 1e). 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 \( \Pi \) is a permutation of the operations in the DFG.

When operation \( o_n \) precedes operation \( o_m \) in a permutation \( \Pi \), written \( o_n <_\Pi o_m \), then \( o_n \) has the priority over \( o_m \).

In our approach, solutions are encoded by priority encodings. An explorer creates priority encodings \( \Pi \) and a constructor builds a solution for each \( \Pi \) 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 \( T \). 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 BCG, 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 \( \Pi \).

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.
3 Preventing inefficient code motions

Figure 2 shows a behavioral description, its BBCG and two SMGs obtained assuming 1 adder, 1 subtractor and 1 comparator. Since operations a and b depend only on the inputs, they are both available at state s0 within BB I, i.e. A₀ = {a, b}. Solutions SMG₁ and SMG₂ are induced by Π₁ (with b <ₜ₂ a) and by Π₂ (with a <ₜ₂ b), respectively. Note that the schedule length of the left path is increased by the execution of b in the first state of SMG₁, as compared to SMG₂. The reason is that a and b cannot be scheduled in a same state because τ(a) = τ(b) = subtractor. Given the encoding Π₁ in Figure 2c we have A₀ = (b, a). If we reorder this set such that A₀ = (a, b), the code motion of b that places it in state s₀ is prevented and solution SMG₂ (which is inferior) is not constructed. The example illustrates that the parallelism actually exploitable is limited 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 oᵢ and its initial link, each lowest link is determined such that it points to the latest BB downwards on a given control path, where oᵢ can be legally executed. Our analysis checks if an operation oᵢ can be legally moved down to BBj to BBj, i.e. oᵢ 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 BBᵢ and BBj with BBᵢ ⋾ BBj, the operation oᵢ is free to move from BBᵢ to BBj, written free(oᵢ, BBᵢ, BBj), iff: ¬kill(oᵢ, BBᵢ, BBj) ∨ (oᵢ conditional) ∨ (BBj ≠ sink).

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

Algorithm 1. Finding the lowest links

procedure find_lowest_links()
mark initial links;
foreach input νᵢ of DFG
    foreach oᵢ ∈ CONS(νᵢ)
        visit(oᵢ);
        delete initial links;
procedure visit(oᵢ)
if (oᵢ is visited) return;
mark oᵢ as visited;
foreach oᵢ ∈ CONS(oᵢ) visit(oᵢ);
foreach BBᵢ with oᵢ ⋾ BBᵢ
    foreach BBk with oᵢ ⋾ BBk
        BBj := find_boundary(oᵢ, BBᵢ, BBk);
        create a new link λ such that oᵢ ⋾ BBj;

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 BBj such that operation oᵢ was not scheduled within any BB in path p, and if there exists a lowest link connecting operation oᵢ with BBj, then oᵢ must be scheduled within BBj.

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 BBᵢ, and operation b is linked to BBj, with BBᵢ ⋾ BBj, then a must be scheduled in BBᵢ (Theorem 1), whereas the scheduling of b can be postponed, since b is free to move down to BBj. 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 λ ∈ Λ, let a ∽ₜ₂ b denote the precedence relation induced by the set of links Λ when the basic block BBᵢ is visited. We say that a precedes b at basic block BBᵢ, written a ∽ₜ₂ b, iff: (τ(a) = τ(b)) ∧ ((a = b) ∨ (a ⋾ BBᵢ) ∧ (b ⋾ BBᵢ)).
3.3 Reordering the sets of available operations

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

\[
a \prec_{\Omega,\ell} b \iff (a \prec_{\lambda} b) \lor (b \prec_{\Omega} a) \lor (a \prec_{\Omega} b).
\]  

The linear order \( \prec_{\Omega,\ell} \) is the modification of the original linear order \( \prec_{\Omega} \) 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 \( \tau \) and are such that \( b \prec_{\Omega} a \), but \( a \prec_{\Omega,\ell} b \). When we order the set \( A_k \) according to \( \prec_{\Omega,\ell} \), instead of \( \prec_{\Omega} \), 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 \( \tau \) 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 “sz2” 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 \( \Delta L \) 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 \( \Delta L \) 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 (\( \sigma \)) 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 \( \sigma \) 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

<table>
<thead>
<tr>
<th>example</th>
<th>case</th>
<th>resource constraints</th>
<th>without</th>
<th>with</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>resource constraints</td>
<td>$L_i$</td>
<td>mean</td>
<td>density</td>
</tr>
<tr>
<td></td>
<td>alu</td>
<td>add</td>
<td>sub</td>
<td>mul</td>
</tr>
<tr>
<td>kml1</td>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>rotor</td>
<td>A</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s2r</td>
<td>A</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>G</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>kml2</td>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

except for a few examples where it remains the same.

![Image of density graph]

**Figure 4:** Density of solutions satisfying a given $T_c$.

For the example “kml2”, 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 $L_i > 50$, 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 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 $a \prec_b b$ takes constant time and the test $a \prec_{<,i} b$ 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