A Constructive Method for Exploiting Code Motion

Luiz C. V. dos Santos\textsuperscript{1}, M.J.M. Heijligers, C.A.J. van Eijk, J.T.J. van Eijndhoven and J.A.G. Jess

Design Automation Section, Eindhoven University of Technology
P.O.Box 513, 5600 MB Eindhoven, The Netherlands

Abstract

In this paper we address a resource–constrained optimization problem for behavioral descriptions containing conditionals. In high–level synthesis of ASICs or in code generation for ASIPs, most methods use greedy choices in such a way that the search space is limited by the applied heuristics. For example, they might miss opportunities to optimize across basic block boundaries when treating conditional execution. We propose an approach based on local search and present a constructive method to allow unrestricted types of code motion, while keeping optimal solutions in the search space. A code–motion pruning technique is presented for cost functions optimizing schedule lengths. A technique for treating concurrent flows of execution is also described.

1. Introduction

In the high–level synthesis of an application–specific integrated circuit (ASIC) or in the code generation for an application–specific instruction set processor (ASIP), four main difficulties have to be faced when conditionals are present in the behavioral description:

\begin{enumerate}
  \item the NP–completeness of the resource–constrained scheduling problem itself.
  \item the limited parallelism of operations enclosed by basic blocks, as they might not use all available resources completely.
  \item the possibility of state explosion because the number of paths may explode in the presence of conditionals.
  \item the possibility of state explosion due to limited resource sharing of mutually exclusive operations (which is limited by the timely availability of test results).
\end{enumerate}

Most methods apply different heuristics for each subproblem (BB scheduling, code motion, code size reduction) as if they were independent. An heuristic is used to decide the order of the operations during scheduling (like the many flavors of priority lists), another to decide whether a particular code motion is worth doing [6, 17], yet another for a reduction on the number of states [18]. As a result, these approaches might miss optimal solutions.

We propose a formulation to encode potential solutions for the interdependent subproblems. We show that optimal solutions are kept in the search space. The formulation abstracts from the linear–time model and allows us to concentrate on the order of operations and availability of resources. Figure 1 shows the outline of our approach. We have a solution explorer which is based on a local search algorithm [15]. The constructor is driven by a permutation \( \Pi \) of the operations in the data flow graph. The explorer handles encoded solutions and uses a solution constructor to evaluate their cost. A code–motion pruning is embedded in the constructor to reduce the search space.

![Figure 1 – An outline of the approach](image)

2. The Problem

\textbf{Definition 1:} A control data flow graph \( \text{DFG} = (U, E) \) is a directed graph where the nodes represent operations and the edges represent the dependencies between them. We assume that the DFG contains conditional constructs.

\textbf{Definition 2:} An execution condition is associated with every operation, represented as a boolean function, hence called a predicate. An execution instance (EXI) is a set of operations executed under a given predicate.

\textbf{Definition 3:} A basic block (BB) is a set of operations which all have the same execution condition.

\textbf{Definition 4:} A basic block control flow graph \( \text{BBCG} = (V, F) \) is a directed graph where the nodes represent basic blocks and the edges represent the flow of control. We allow that an operation initially associated with a given BB be "moved" to another BB; this is called code motion.

\textbf{Definition 5:} Each traversal in the BBCG from an input node to an output node such that only BBs which execute under a same predicate are visited defines a path. A path defines a sequence of BBs which enclose all the operations belonging to an EXI of the DFG. Each path in the BBCG corresponds to exactly one EXI in the DFG.

\textbf{Optimization problem (OP):} Given a number \( K \) of functional units and an acyclic DFG, find a control sequence represented by a state machine graph, in which precedence constraints of the DFG are obeyed and the resource constraints are satisfied for each functional unit type, such that a cost function \( C \) is minimized.
3. Related Work

In path-based scheduling (PBS) [3, 2] a so-called as-fast-as-possible (AFAP) schedule is found for each path independently, provided that a fixed order of operations be chosen in advance. Due to the fixed order and to the fact that scheduling is cast as a clique covering problem on an interval graph, code motions resulting in speculative execution are not allowed. Thus, the method has limited capability of exploiting parallelism with complex control flow [13]. This limitation is released in tree-based scheduling (TBS) [10], by allowing boosting and duplication code motions.

Condition vector list scheduling (CVLS) [18] allows code motion and supports speculative execution. However, the underlying mutual exclusion representation is limited [1].

In Trace-scheduling (TS) [6] a main path (trace) is chosen to be scheduled first and independently of other paths, then another trace is chosen and scheduled, and so on. TS doesn’t allow certain types of code motion across the main trace.

In [17] an approach is presented where code-motions are exploited. BBs are scheduled using a list scheduler and then code motions are allowed. One priority function is used in the BB scheduler and another for code motion. Code motion is allowed only inside windows containing a few BBs to keep runtime low, but then iterative improvement is needed not to restrict too much the kind of code motions allowed.

Among those methods, only PBS is exact, but it solves a partial problem where speculative execution is not allowed. TBS and CVLS address BB scheduling and code motions simultaneously, but use classical list scheduler heuristics. TS combines both subproblems in a per trace basis, but main-trace-first heuristics are applied. In [17] a different heuristic is applied to each subproblem. All those methods may exclude optimal solutions from the search space.

In [16], an exact symbolic technique for control dependent scheduling is presented. However, restrictions are imposed on the speculative execution model. Besides, the use of an exact method in early (more iterative) phases of a design is unlikely, especially because no pruning is presented to cope with the larger search space due to code motions.

A method could be envisaged where no restriction is imposed neither on the kind of code motion, nor on the order the operations are taken to be scheduled.

4 The method

4.1. Outline of the method

The solution constructor takes a permutation of the operations and generates a solution. Techniques borrowed from the constructive topological sorted scheduler [8] are used, because it has the important property that there always exists a permutation which results in an optimal solution. A schedule is constructed out of the permutation as follows. An operation to be scheduled is selected among ready operations (unscheduled operations whose predecessors are all scheduled) following the order in the permutation. Each selected operation is attempted to be scheduled at the as early as possible time where a free resource is available.

In figure 2, a linear-time sequence is constructed for a given permutation. See [5] for an explanation of the symbols used. The topological sorted scheduler is applied on purpose in a very straightforward way (figure 2b). No information about mutual exclusion is used. When such information is used, b can be scheduled at the second step by sharing an adder with operation c (figure 2c). We are assuming that the outcome of i is not available inside the first step to allow a and b to conditionally share a resource. The resulting schedule length is reduced to 5 steps for both EXIs, even though the EXI (t, b, g) could be scheduled on its own in only two steps. Information about mutual exclusion is clearly not enough and the limitation is the linear-time model. To allow a more efficient solution, some mechanism has to split the linear-time sequence by exposing a flow of control. Our mechanism is based on initial links, as we will explain in the next subsections.
4.2. Initial links

In our method we want to capture the freedom for code motions without restrictions and for this purpose we introduce the notion of a link. A link connects an operation \( u \) in the DFG with a BB \( v \) in the BBCG. Its interpretation is that \( u \) may be executed under the predicate which defines the execution of operations in \( v \). A same operation can be linked to several mutually exclusive BBs. Figure 3 illustrates the link concept. A merge node (M) represents data selection and a branch node (B) represents control selection.

Figure 3 – Paths and execution instances

Initial links: We will encode the freedom for code motion by using a set of initial links. First, we look for each operation whose result has to be available before a control or data selection point in a given EXI. Such an operation is said to be a terminal. Each terminal is linked to the BB which precedes the corresponding selection point in the BBCG. In figure 3, initial links are shown for terminals \( c \) and \( d \) (due to data selection) and \( t \) (due to control selection). Then, we link each ancestor of a terminal to the same BB to which the latter is linked. Operation \( a \) is not a terminal and will have initial links (not shown in figure) to both BB2 and BB3; \( b \) will have a single initial link pointing to BB3. Each link points to the latest BB in a given path where the respective operation can still be executed. This means that each operation is free to be executed inside any preceding BB on the same path as soon as data precedence and resource constraints allow (the only control dependency to be satisfied is the need to execute the operation at the latest inside the BB pointed by the initial link). The underlying idea is to traverse the BBCG in topological order trying to schedule operations in traversed BBs. If operation \( u \) is given an initial link to BB \( v \) and \( v \) is reached in the traversal, then \( u \) must be scheduled inside it. We say that the assignment of \( u \) to BB \( v \) is compulsory.

Figure 4 – Linking unconditional operations

Final assignments: Each link \( \alpha \) will be called here an operation assignment (from now on called simply assignment) when it obeys precedence constraints and it doesn’t imply the need for more than the available functional units. Assignments which might increase registers and/or interconnect usage are included in the search space. Each assignment from an operation \( u \) into a BB \( v \) is given the following attributes:

- a) condition (execution condition of \( u \) when its predicate is completely evaluated); b) begin (operation starting time of \( u \) in BB \( v \)); c) end (completion time of \( u \) in BB \( v \)). Assignments represent a relative-time encoding. The absolute time is given by the instant BB \( v \) starts execution plus the value in attribute begin.

Freedom for code motion: Operations may be redundant for some paths in a behavioral description [10]. TBS uses tree optimization to remove redundancies by propagating each operation to the latest BB where it is to be used. CVLS eliminates them by using extended condition vectors [18]. Even though they remove redundancies, those methods don’t succeed to encode the freedom for code motion. Tree optimization works well for (nested) conditional trees, but in the case of parallel flows of execution, the quality of a solution may depend on the description. In figure 4, after tree optimization, the unconditional operation \( f \) would remain associated to the BB wherein it was originally described.

Merging concurrent flows of execution: Our initial links do not only handle redundancies, but also encode freedom for code motion. In figure 4, \( f \) may be linked to BB4, to both BB3 and BB2 or to BB1. \( \alpha \), is the initial link, because the only control dependency to be satisfied is that \( f \) execute before the output is available. As operation \( f \) can be executed in BB1 or in any preceding BB as soon as resource and data dependency constraints are satisfied, unrestricted code motions can be exploited, even for concurrent flows of execution.

4.3. The solution constructor

Traversing in topological order: The solution constructor follows the flow of tokens in the DFG while the BBCG is traversed in topological order. An operation can be assigned to any traversed BB, as soon as data precedence and resource constraints allow. If more than one operation satisfies these constraints, an operation will be chosen based on the order in the permutation. Such an assignment is not compulsory as long as the BB to which the operation was initially linked is not reached. As a result, an initial link \( u \rightarrow v \) might become a final assignment, but it will be revoked if \( u \) succeeds to be scheduled inside any ancestor of \( v \), inducing a code motion.

Splitting the linear–time sequence: Only when an operation is compulsory in a BB, it is allowed to “allocate” extra time steps inside that BB. This will make room for scheduling non–compulsory operations in idle resources. We claim that the underlying pruning associated with this criterion to split the linear–time sequence doesn’t discard any better solutions (see section 2 and proof in Appendix A).

Example: In figure 5 the same example as in figure 2 is scheduled to illustrate the method. It is shown in figure 5b how each EXI could be scheduled independently. EXI1 = \{t, a, c, e, d, f, g\} is scheduled in five steps and EXI2 = \{t, b, g\} is scheduled in 2 steps. It is not possible to overlap those sequences, because \( a \) and \( b \) cannot share the adder (the outcome of test \( t \) is not available inside the first step). Even though each path can be AFAP scheduled for the given \( \Pi \), there is a conflict between them so that if one sequence is chosen, the other will be imposed an extra step. Figure 5c shows the initial links. Figures 5d to 5k show the evolution.
allow speculative execution to the first step represents speculative execution. If we don’t impose an extra step. When a conflict happens, at that point no other ready operations can be scheduled in that BB, as they would require the allocation of extra steps. Then, another BB is taken (figure 5b) and so on. Figure 5k shows the final result. It is the same as obtained by scheduling EXI1 independently (figure 5b), but EXI2 was imposed an extra step, resulting in schedule lengths of 3 and 6.

The solution constructor is summarized in algorithm 1.

Algorithm 1 – The solution constructor

Runtime complexity: Let \( n \) be the number of operations in \( \Pi \), \( b \) the number of BBs, \( p \) the number of paths and \( c \) the number of conditionals. The search for the first ready operation in \( \Pi \) takes \( O(\lg n) \). As this search may be repeated for each operation and for each BB, the worst case complexity of algorithm 1 is \( O(b n \lg n) \). The runtime efficiency of our approach doesn’t depend on \( p \) (which can grow exponentially in \( c \)), as opposed to path-based methods.

4.4. Exploiting unrestricted code motions

In this section we summarize the relationship between initial links, assignments and code motions. A detailed analysis of code motions can be found in [17].
5. Experimental results

The method has been implemented in the NEAT system [7]. A genetic algorithm is used as solution explorer in the current implementation. For the representation of predicates we are using the BDD package developed by Geert Janssen.

### 5.1 Results for Wakabayashi’s example

<table>
<thead>
<tr>
<th>method</th>
<th>case</th>
<th>#alu</th>
<th>#add</th>
<th>#sub</th>
<th>chain</th>
<th>lengths</th>
</tr>
</thead>
<tbody>
<tr>
<td>ours</td>
<td>a</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4.4,7</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3.4,7</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3.4,6</td>
</tr>
<tr>
<td>TBS</td>
<td>a</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3.4,7</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3.4,7</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3.4,6</td>
</tr>
<tr>
<td>HRA</td>
<td>a</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4.4,7</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3.4,7</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3.5,6</td>
</tr>
<tr>
<td>CVLS</td>
<td>a</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4.5,7</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3.6,7</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3.5,6</td>
</tr>
</tbody>
</table>

### 5.2 Results for maha and parker

<table>
<thead>
<tr>
<th>method</th>
<th>maha</th>
<th>parker</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>ours</td>
<td>4/1</td>
<td>2.25</td>
<td>2.00</td>
</tr>
<tr>
<td>CVLS</td>
<td>4/1</td>
<td>2.38</td>
<td>2.38</td>
</tr>
<tr>
<td>[16]</td>
<td>4/-</td>
<td>2.25</td>
<td>2.13</td>
</tr>
</tbody>
</table>

In table 1, our method is compared with others for the example in [18]. Results were collected from [10], [18] and [12]. Our solution for case a is as good as TBS and HRA [12] and better than CVLS. In case b our method, TBS and HRA reach the same results which are better than PBS. For case c, both our method and TBS are better than HRA and PBS. In table 2 we compare our results for benchmarks maha and parker used in [19] and [16]. Our method reaches the same average values for those methods during exploration (2.25, 2.38 and 2.13), but a better average value (2.00) is found.

Although we certainly need to perform more experiments, these first results are encouraging. They seem to confirm that our method is able to find the code motions which induce the better solutions.

Figures 8 and 9 show the impact of our code-motion pruning. For the experiment, 50 solutions were constructed.
induced by randomly generated permutations. Figure 8 shows results with (black) and without (gray) pruning. The height of a bar represents the number of solutions counted under different assumptions. In cases (3) (waka(3) and maha(3)), solutions are distinguished by comparing the length of each path. In cases (1) and (2), solutions are distinguished only by their overall cost. Case (1) uses max $T$, as a cost function and case (2) uses $\sum T$. In waka(3) for example, 32 different solutions are identified without pruning, but only 5 with pruning. This reduction will lead to an effective reduction in search space, which depends on the chosen cost function (compare waka(1) and (2)).

**Figure 8 – Reduction of the number of solutions**

Not only the number of solutions is reduced, but also the range of cost values. The cost range ratio “no pruning/pruning” is 2.5 for waka(1) and maha(1), 3.5 for waka(2), and 1.55 for maha(2). These ratios are shown in figure 9, normalized with respect to the “no pruning” cost range. This compaction on cost range suggests a higher probability of reaching (near) optimal solutions during local search.

6. Conclusions and Future Work

We have cast the resource–constrained OP for descriptions with conditionalities into a local search problem. The permutation–driven constructor deals with code motions constructively while keeping optimal solutions in the search space. The approach can be extended to include other issues.

Most methods for conditionals either break loops or allow limited optimization between iterations. Our method could accommodate one of such approaches, but we prefer to investigate loops as a further topic to prevent limitation on the search space. We also intend to consider time constraints.

**Appendix A. Proof of the pruning technique**

**Theorem:** Let $S_m$ be a solution of the described OP constructed with algorithm 1 for a given $\Pi$ and consider an operation $o$ assigned to BB $j$ in such a solution. Let $\delta$ be $[\text{delay}(o)]$. If a solution $S_n$ is obtained by moving $o$ from BB $j$ into BB $i$ and it allocates exactly $\delta$ cycle steps inside BB $i$, then $\text{cost}(S_n) \geq \text{cost}(S_m)$, where $\text{cost}$ is a monotonically increasing function in terms of schedule lengths.

**Proof:** Let $(k), (k) \in \mathbb{N}$ be the schedule lengths of a BB $k$ before and after the motion, respectively. Let $p, q, r, s$ be BBs forming path $p \rightarrow q \rightarrow r$ and path $p \rightarrow s \rightarrow r$. Let $li_i$ and $lo_i$ be, respectively, the schedule lengths of path $li_i$ before and after motion.

1. $o$ was assigned to $q$ and allocates $\delta$ steps inside $p \rightarrow L(p)=l(p)+\delta$.
2. No operation assigned to $r$ can be moved in the allocated steps.

$\therefore L(q) \geq l(q)+\delta+L(p)=l(p)+\delta$.

b) $o$ was assigned to $r$ and allocates $\delta$ steps inside both $q$ and $s$.

$\therefore L(q)=l(q)+\delta, L(s)=l(s)+\delta, L(r) \geq l(r)+\delta \Rightarrow L_i \geq l_i$ and $L_j \geq l_j$.

c) $o$ was assigned to $r$ and allocates $\delta$ steps inside $p \Rightarrow L(p)=l(p)+\delta \Rightarrow o$ doesn’t depend but has resource conflicts with operations assigned to $q$ and $s$ (topological sorted construction).

$\therefore L(q)=l(q), L(s)=l(s), L(r) \geq l(r)+\delta \Rightarrow L_i \geq l_i$ and $L_j \geq l_j$.

**For a given $\Pi$, solution $S_n$ have path lengths greater than or equal to those in $S_m$. As compound code motions can be built out of these basic code motions, and $\text{cost}$ is monotonically increasing, we can conclude without loss of generality that $\text{cost}(S_n) \geq \text{cost}(S_m)$.**

**Figure 9 – Compaction on cost range**

References


