Summary of the paper "Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator", written by Silvina Hanono and Srinivas Devadas, both members of Department of EECS, MIT.

Summary written by David Landén, D4B

Introduction

AVIV is a retargetable code generator, which means that it can generate code for different target machines. The program takes a description of the target processor, written in ISDL (Instruction Set Description Language), and a application code file, written in C or C++, as input. AVIV optimizes for minimum code size, since it is intended to be used to generate code for embedded systems, where the size of on-chip ROM is important. AVIV focuses on VLIW (Very Long Instruction Word) architectures, with many functional units, where many instructions can be executed in parallel and thus a clever optimization algorithm can reduce the code size by squeeze as much as possible into every VLIW. AVIV solves the problems instructions selection, resource allocation and scheduling by creating a Split-Node Dag of the application’s all basic blocks. This representation is used to set up a heuristic branch-and-bound step that performs functional unit assignment, operation grouping, register bank allocation, and scheduling concurrently.

Code generation in AVIV

The Split-Node Dag describes all possible ways that any operation can be performed on the target processor. In Figure 1, there is description of a target architecture containing three functional units, which can perform different operation, but only on at a time. The processor has also an instruction memory and a data memory. In figure 2, there is basic block and it’s representation as a dag. This dag is split over the functional units and transformed to the split-node dag in figure 3. The number of children for every split node is the same as the number of functional units that can perform it’s instruction. For instance, the subtraction node is split into two node since two functional units can perform subtraction. Transfer nodes are inserted between split nodes and instructions node. The code generation covers the split node dag in five steps, using heuristic functions in each step. Finally, detailed register allocation and peephole optimization are performed.

Figure 1: Target architecture

Figure 2: Basic block, and it’s Dag.

Figure 3: The Split-Node Dag.
A. Exploring the Split-Node Functional Unit Assignments

In the first step, a functional unit must be selected to cover each and every split node. But this will be all too CPU-intensive, since even a small basic block corresponds to a split-node dag with many split-nodes. The split node dag is covered, using a branch-and-bound, and only a few split-nodes are covered in each step and explored in further detail. A heuristic function, that measure the cost in number of instruction, and prunes the split-node dag. The number of registers required from each register bank are estimated, and loads and spills are inserted if the available resources are exceeded.

B. Adding Required Transfers

Data transfer nodes are added for every split-node assignment. Where only one possible data transfer path exists, this step is straightforward. In the case of multiple transfer paths, the heuristic is applies again and select the transfer node, but this time the choice is based only on parallelism.

C Maximal Groupings

All functional unit assignment, will now be grouped into as few as possible VLIW instructions. A N x N matrix is created, where N is the number of nodes in the split node dag. A position in the matrix contain 0 if it’s row and column number (nodes) can be executed in parallel, and 1 otherwise. Maximal cliques are created in the matrix, choosing nodes that are executed on different functional units and where no direct path exists between the nodes. But generating maximal cliques is also very CPU-intensive and therefore, another heuristic is used in this step, that will not group nodes together which lies on very different levels in the split-node dag. This heuristic maintains the quality of the results. Some generated cliques may not correspond to legal VLIW instruction and thus have to be split into smaller VLIW instructions until all parts can be executed on the target machine.

D. Selecting a Minimum-Cost Set of Maximal Cliques

When the maximal cliques have been generated, the next step is to select the minimum-cost set that cover all the nodes in the assignment.

The algorithm starts with an empty set, and selects the clique that covers the largest number of remaining uncovered nodes, whose children have all been covered. After selecting the clique, the remaining cliques are shrunk so that they no longer contains any of the covered nodes. This is repeated until all nodes a covered.

E. The Covering Solution

The assignment that required the minimum-cost of cliques to cover all its nodes is selected as the final assignment. Now, unit assignment has been made, operations and data transfers are grouped into VLIW instructions, register bank allocation has been performed, and a schedule has been determined.

Finally, detailed register allocation is performed, using graph coloring algorithms. If any spill and loads are not required, peephole optimization is performed. This might improve the schedule (reducing the code size), but it not always the case, since the remaining parts of a VLIW instruction could not be spilt into other such instructions.
Experiments

Running a few examples on a Sun Microsystems Ultra-30/300 resulted in the data in table 1. The column "by hand" corresponds to optimal solutions. AVIV:s solutions where close to optimum, but none of the examples consisted of very many nodes.

<table>
<thead>
<tr>
<th>Basic Block</th>
<th>Original DAC #Nodes</th>
<th>Split-Node DAC #Nodes</th>
<th>#Regs per Rev File</th>
<th>#Splitted Injected</th>
<th>#Nodes in Solution</th>
<th>CPU Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ex1</td>
<td>8</td>
<td>20</td>
<td>4</td>
<td>0</td>
<td>7</td>
<td>0.1 (0.2)</td>
</tr>
<tr>
<td>Ex2</td>
<td>13</td>
<td>56</td>
<td>4</td>
<td>0</td>
<td>10</td>
<td>0.4 (37.2)</td>
</tr>
<tr>
<td>Ex3</td>
<td>11</td>
<td>55</td>
<td>4</td>
<td>0</td>
<td>13</td>
<td>0.9 (1223)</td>
</tr>
<tr>
<td>Ex4</td>
<td>15</td>
<td>81</td>
<td>4</td>
<td>0</td>
<td>16</td>
<td>8.2 (41,466)</td>
</tr>
<tr>
<td>Ex5</td>
<td>16</td>
<td>106</td>
<td>4</td>
<td>0</td>
<td>14</td>
<td>10.7 (88,337)</td>
</tr>
<tr>
<td>Ex6</td>
<td>15</td>
<td>91</td>
<td>2</td>
<td>2</td>
<td>18</td>
<td>6.9 (29,072)</td>
</tr>
<tr>
<td>Ex7</td>
<td>16</td>
<td>106</td>
<td>2</td>
<td>1</td>
<td>15</td>
<td>9.9 (61,748)</td>
</tr>
</tbody>
</table>

Table 1

Code Generation Experiments for the Example Target Architecture