Optimal Integrated Code Generation for Clustered VLIW Architectures

Christoph Kessler and Andrzej Bednarski
PELAB Programming Environments Laboratory
IDA – Computer Science Department
Linköpings Universitet
Linköping, Sweden

Phase-decoupled code generation

Phase ordering problem

Register allocation before scheduling introduces additional data dependences
→ less parallelism / alternatives

Example:

\[ \text{reg}(t_1) := \text{reg}(t_3) = R1 \text{ implies } \]
"b before c", as c overwrites t_1 in R1.

Register allocation after scheduling
scheduling determines live ranges → interferences
spill code must be scheduled
→ may compromise quality of schedule!
Integrated code generation

Code generation for DSPs: Partitioning ↔ Scheduling

Clumped VLIW architectures, e.g. TI C6201:

- Mapping instructions to clusters
  
  - may profit from information about free copy slots in the schedule
- Instruction scheduling
  
  - must generate copy instructions to match residence of operands and instructions

Heuristic [Leupers‘00]: iterative optimization with simulated annealing

Towards integrated code generation

Heuristics

Integer Linear Programming

- [Wilson, Grewal, Henshall, Banerji‘95]
- SILP [Zhang‘96] [Kästner‘97,‘00] $O(n^2)$ vars, $O(n^2)$ inequalities
- OASIC [Gebotys/Elmasry‘92,‘93] [Kästner‘97,‘00]
  
  - $O(n^2)$ vars, exponential inequalities if register allocation integrated
  
  - versatile, but optimal solution only for small problem instances

Graph-based, dynamic programming algorithm

- problem-tailored solution strategy
- practical for typical problem sizes
- full integration of all phases
- retargetability: XML-based hardware description language (mixed mode)
- parallelizable

More phase ordering problems: Code generation for DSPs

Example: Hitachi SH3-DSP

Residence constraints on concurrent execution (load + mul, add + mul, ...)

Instruction scheduling and register allocation are not separable!

Phase-decoupled standard methods generate code of poor quality.
List Scheduling = Local Scheduling by Topological Sorting \[\text{[Coffman'76]}\]

\[\text{DAG } G \]

Heuristics based on list scheduling
e.g. “deepest-level first”

Integrate instruction selection

Keep track of register need

Optimal solution
by complete enumeration of all alternatives ???

Compression of the space of partial solutions

Heuristics based on list scheduling
e.g. “deepest-level first”

Optimal solution
by complete enumeration of all alternatives ???

Time profiles

Time profile \(P\): window of the instructions scheduled last for each unit
that may still influence future scheduling decisions.

Extended selection node \((z, t, P)\)
summarizes all schedules of \(\text{scheduled}(z)\) that end with time profile \((t, P)\).

Time-inferior extended selection nodes can be pruned.
Keeping track of value residence

Register class:
derived from operand residence constraints of instructions

Residence class:
register class or memory module
instruction selectable only if operands are in right residence class

Transfer instructions:
copy values to different residence classes: move, load, store
should increase the residence potential of live variables
consider migration (re-partitioning) at any time

Space profiles:
keep track of residence classes of live variables

Comparability Theorem for Dynamic Programming

Theorem:
For determining a time-optimal schedule, it is sufficient to keep just one locally optimal target-schedule \( s \) among all those target-schedules \( s' \) for the same subDAG \( G_s \) that have the same time profile and the same space profile.

Hence, \( s \) can be used as prefix for all schedules that could be created from these target schedules \( s' \) in a subsequent selection step.

Structuring the space of partial solutions

Appending an instruction covering a DAG node \( \nu \)
+ increases time by 0cc or more
+ increases (IR-schedule) length

Appending a register transfer instruction MV \( R_1, R_2 \)
+ increases time by 0cc or more
+ does not change (IR-schedule) length
+ may change residence potential in RClass(\( R_2 \))

\[
\text{rpot}(\alpha, z) = \sum_{\nu \in \alpha} \left\{ \begin{array}{ll} 1 & \text{if } \nu \text{ resides in } \alpha \\ 0 & \text{otherwise} \end{array} \right.
\]

monotonic function describing residence potential:

\[
\text{RPot}(\alpha, l) = l \cdot (|T| \cdot |V| + 1) + \sum_{\omega \in \mathcal{P}} \text{rpot}(\alpha, \text{alive}(\omega))
\]
### Results for TI-C62x variant: DSP benchmarks

<table>
<thead>
<tr>
<th>Basic block</th>
<th>(V)</th>
<th>time[s]</th>
<th>space[MB]</th>
<th>#Impr.</th>
<th>#merged</th>
<th>#ESNodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>cplxinit</td>
<td>14</td>
<td>201</td>
<td>38</td>
<td>0</td>
<td>140183</td>
<td>44412</td>
</tr>
<tr>
<td>vectcopy init</td>
<td>12</td>
<td>16</td>
<td>11</td>
<td>26</td>
<td>61552</td>
<td>12286</td>
</tr>
<tr>
<td>vectcopy loop</td>
<td>14</td>
<td>47</td>
<td>15</td>
<td>58</td>
<td>149899</td>
<td>29035</td>
</tr>
<tr>
<td>matrixcopy loop</td>
<td>18</td>
<td>18372</td>
<td>194</td>
<td>1782</td>
<td>5172442</td>
<td>722012</td>
</tr>
<tr>
<td>vectsum loop</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>36715</td>
<td>8896</td>
</tr>
<tr>
<td>vectsum unrolled</td>
<td>17</td>
<td>1537</td>
<td>58</td>
<td>3143</td>
<td>1316864</td>
<td>198571</td>
</tr>
<tr>
<td>matrixsum loop</td>
<td>17</td>
<td>10527</td>
<td>154</td>
<td>2898</td>
<td>3502106</td>
<td>564058</td>
</tr>
<tr>
<td>dotproduct loop</td>
<td>17</td>
<td>3495</td>
<td>77</td>
<td>4360</td>
<td>2382094</td>
<td>345256</td>
</tr>
<tr>
<td>codebk_srch bb33</td>
<td>17</td>
<td>431</td>
<td>41</td>
<td>306</td>
<td>319648</td>
<td>64948</td>
</tr>
<tr>
<td>codebk_srch bb29</td>
<td>13</td>
<td>7</td>
<td>12</td>
<td>0</td>
<td>17221</td>
<td>6433</td>
</tr>
<tr>
<td>vecsum_c bb6</td>
<td>20</td>
<td>9749</td>
<td>154</td>
<td>5920</td>
<td>3744583</td>
<td>497470</td>
</tr>
<tr>
<td>vec_max bb8</td>
<td>13</td>
<td>79</td>
<td>35</td>
<td>0</td>
<td>99504</td>
<td>37254</td>
</tr>
</tbody>
</table>

#### Time requirements (DSP benchmarks)

![Graph showing computation time vs. DAG size](image)

#### Space requirements (DSP benchmarks)

![Graph showing space requirement vs. DAG size](image)
Conclusion and outlook

+ Full integration, including static remapping
+ Generalized instruction selection with forest pattern matching
+ Dynamic programming algorithm: feasible for ≤ 20 IR operations
+ Considerable resources (time, space) available for optimization
+ An optimal solution allows to check the quality of fast heuristics

Future work

+ Decrease complexity:
  Exploit symmetry in programs, instruction set properties
+ Overlapping residence classes, “versatility”
+ Global code generation
+ Improved retargetability: ADML
+ Quantitative comparison with ILP methods

ADML

<architecture omega="8">
  <registers> ... </registers>
  <residenceclasses> ... </residenceclasses>
  <funits> ... </funits>
  <instruction_set>
    <instruction id="ADDP4" op="4407">
      <target id="ADD .L1" op0="A" op1="A" op2="A" use_fu="L1"/>
      <target id="ADD .L2" op0="B" op1="B" op2="B" use_fu="L2"/>
      ...
    </instruction>
    ...
  </instruction_set>
  ...
  <transfer>
    <target id="MOVE" op0="r2" op1="r1">
      <use_fu="X2"/>
      <use_fu="L1"/>
    </target>
    ...
  </transfer>
</architecture>