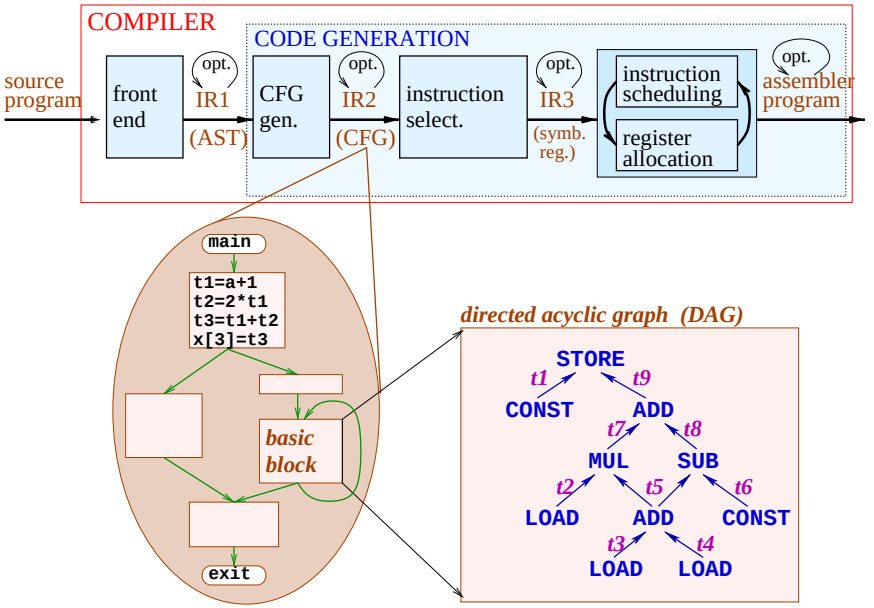
Optimal Integrated Code Generation for Clustered VLIW Architectures

Christoph Kessler and Andrzej Bednarski

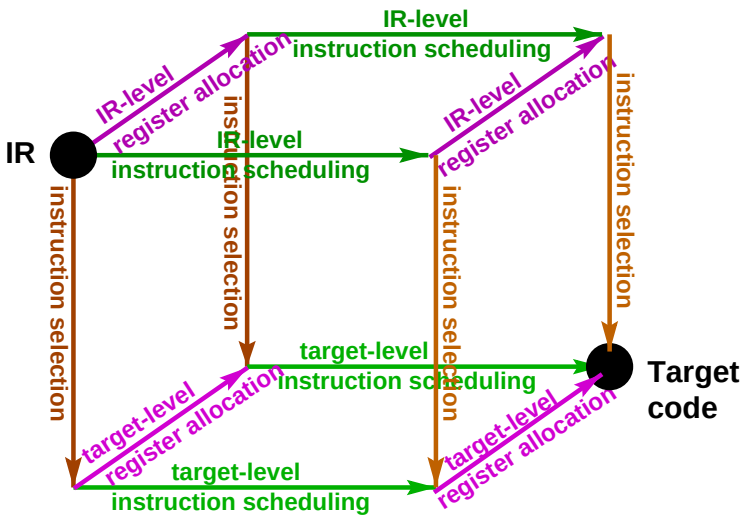
PELAB Programming Environments Laboratory
IDA – Computer Science Department
Linköpings Universitet
Linköping, Sweden

CENIIT 01.06
OPTIMIST

Compiler structure



Phase-decoupled code generation



Phase ordering problem

Register allocation ***before*** scheduling

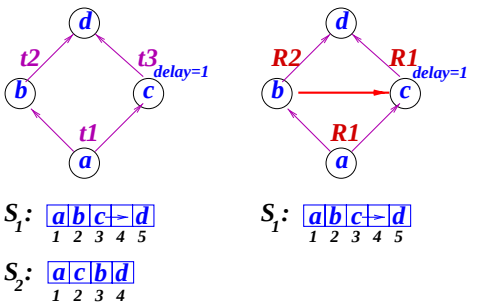
introduces additional data dependences

→ less parallelism / alternatives

Example:

$reg(t_1) = reg(t_3) = R1$ implies

"b before c", as c overwrites t_1 in $R1$.

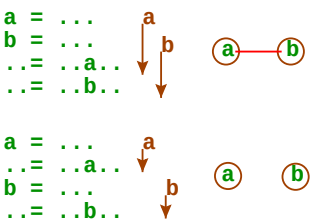


Register allocation ***after*** scheduling

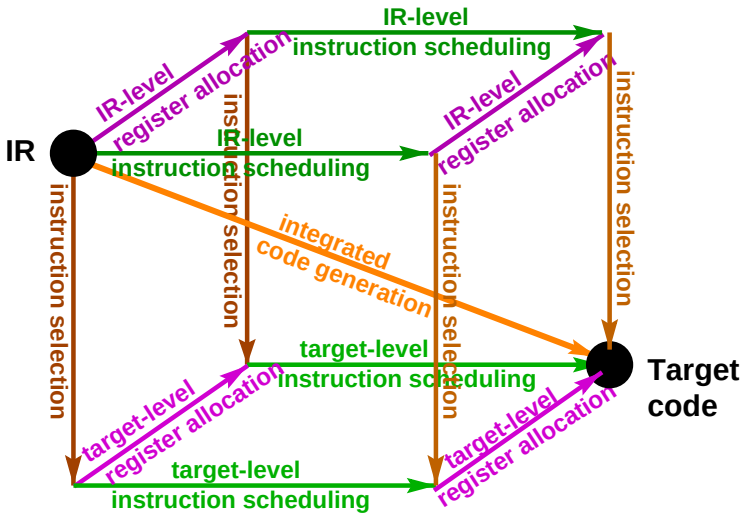
scheduling determines live ranges → interferences

spill code must be scheduled

→ may compromise quality of schedule!

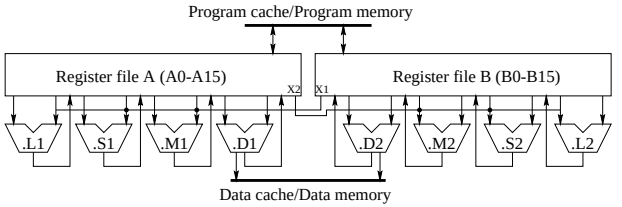


Integrated code generation



Code generation for DSPs: Partitioning ↔ Scheduling

Clustered VLIW architectures, e.g. TI C6201:



simultaneously e.g.

load on A
load on B
move A↔B

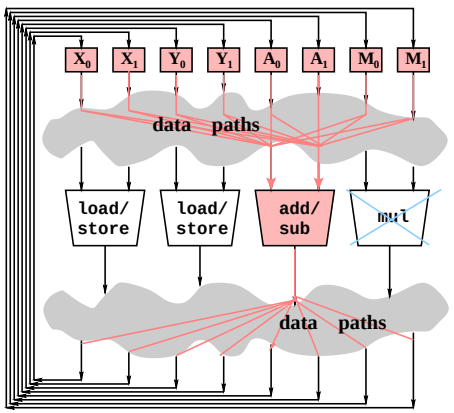
+ mapping instructions to clusters
  may profit from information about free copy slots in the schedule

+ instruction scheduling
  must generate copy instructions
    to match residence of operands and instructions

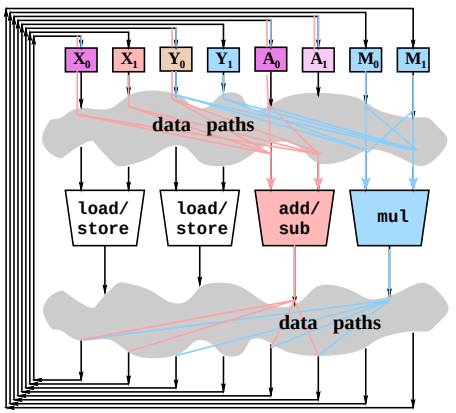
Heuristic [Leupers'00]: iterative optimization with simulated annealing

More phase ordering problems: Code generation for DSPs

Example: Hitachi SH3-DSP



`add + NOP`



`add + mul`

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.

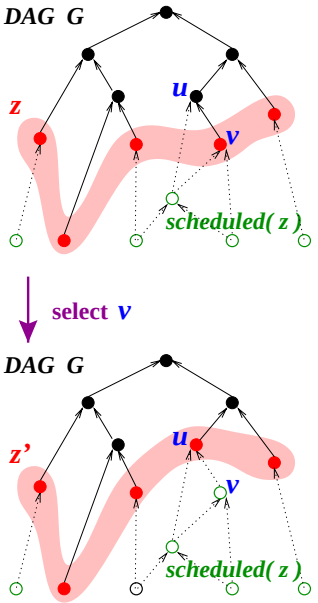
Towards integrated code generation

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

List Scheduling = Local Scheduling by Topological Sorting [Coffman'76]



Heuristics based on list scheduling

e.g. "deepest-level first"

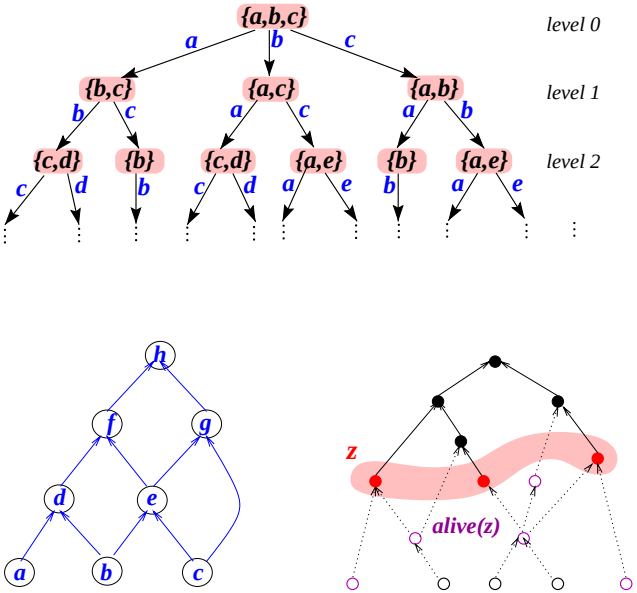
Integrate instruction selection

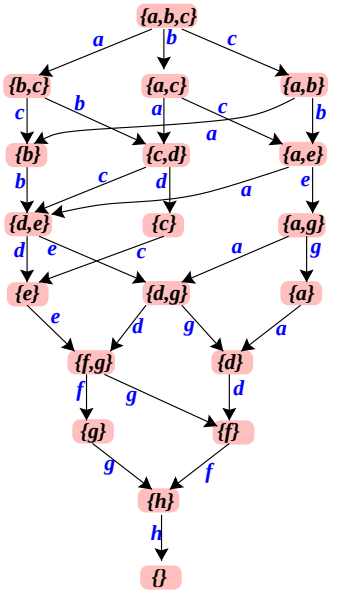
Keep track of register need

Optimal solution

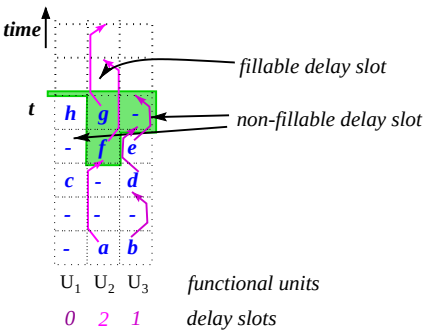
by complete enumeration of all alternatives ???

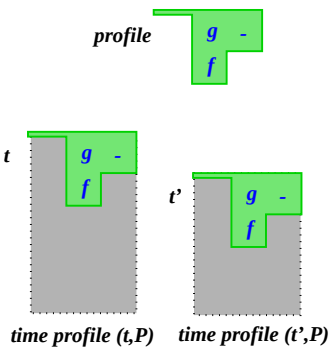
Compression of the space of partial solutions





Time profiles





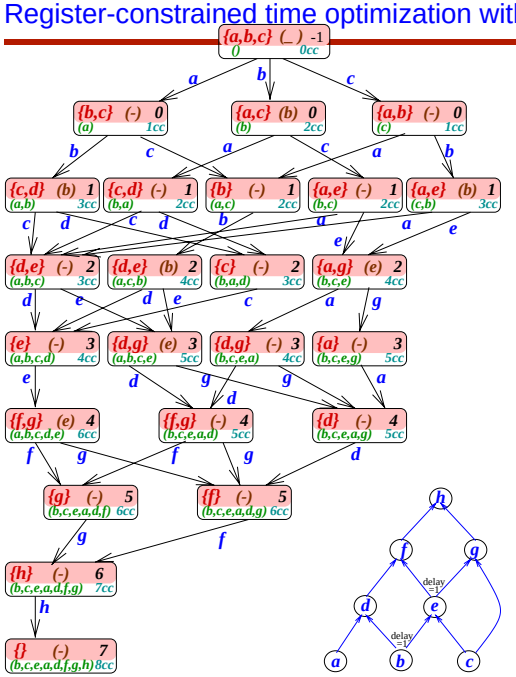
Time profile P: window of the instructions scheduled last for each unit that may still influence future scheduling decisions.

Extended selection node (z, t, P)

summarizes all schedules of $scheduled(z)$ that end with time profile (t, P).

Time-inferior extended selection nodes can be pruned.

Register-constrained time optimization with time profiles



Theorem:

All (prefix) schedules with same zeroind-set and same time profile are comparable.

It is thus sufficient to keep, per ext. selection node, only one, locally optimal, of them.

[K./Bednarski LCTES'01]

Example

for single-issue pipelined processor

Keeping track of value residence

Register class:

derived from operand residence constraints of instructions

Residence class:

register class or memory module

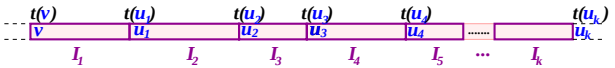
instruction selectable only if operands are in right residence class

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

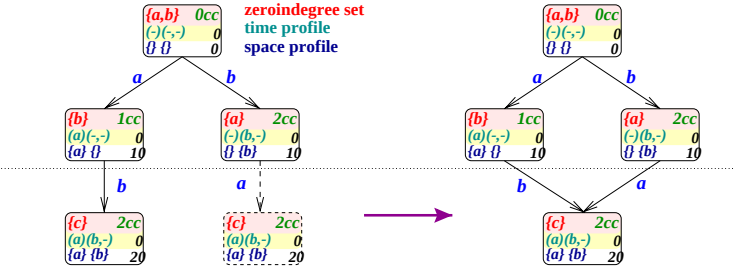


consider spilling to any residence class at any time

Space profiles:

keep track of residence classes of live variables

Comparability Theorem for Dynamic Programming



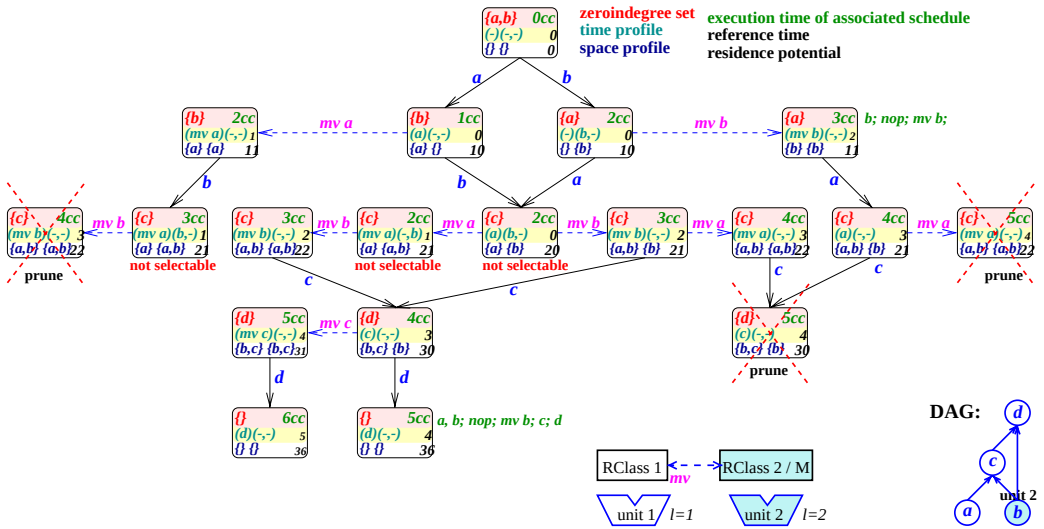
Theorem:

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_z
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.

Example

Solution space of extended selection nodes with time–space profiles



Structuring the space of partial solutions

Appending an instruction covering a DAG node v

+ increases time by 0cc or more
+ increases (IR-schedule) length

Appending a register transfer instruction MV R_1, R_2

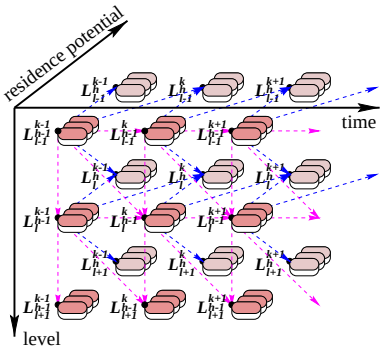
+ increases time by 0cc or more
+ does not change (IR-schedule) length
+ may change residence potential in Rclass(R_2)

$$rpot(\alpha, \zeta) = \sum_{v \in \zeta} \begin{cases} 1 \text{ if } v \text{ resides in } \alpha \\ 0 \text{ otherwise} \end{cases}$$

monotonic function describing residence potential:

$$RPot(z, l) = l \cdot (|P| \cdot |V| + 1) + \sum_{\alpha \in P} rpot(\alpha, alive(z))$$

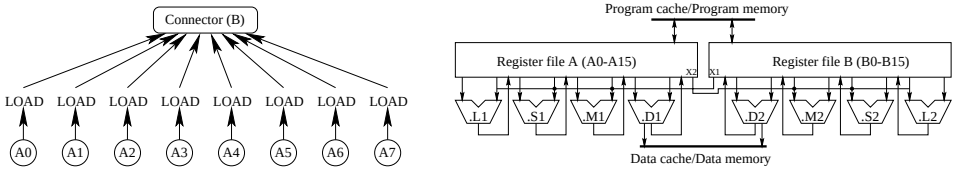
Structure the space of selection nodes as a grid of subsets $L_l^{k,h}$



⇓

partial order for construction

Leupers' example [Leupers'00]

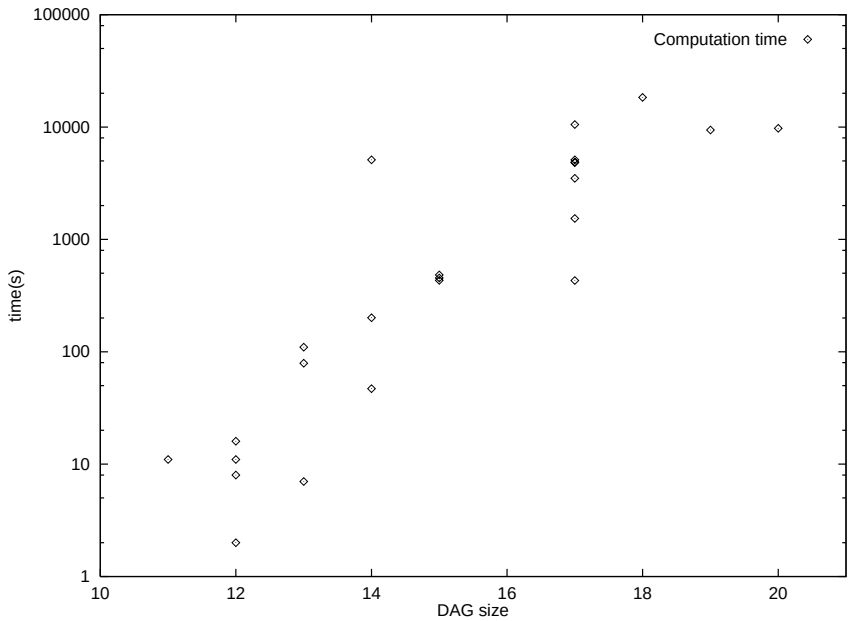


TI-C comp.	Schedule by Leupers' heuristic	Optimal schedule by OPTIMIST
LD *A4,B4	LD *A0,A8 || MV A1,B8	LD *A0,A8 || MV A1,B8
LD *A1,A8	LD *B8,B1 || LD *A2,A9 || MV A3,B10	LD *B8,B1 || LD *A2,A9 || MV A3,B10
LD *A3,A9	LD *B10,B3 || LD *A4,A10 || MV A5,B12	LD *B10,B3 || LD *A4,A10 || MV A5,B12
LD *A0,B0	LD *B12,B5 || LD *A6,A11 || MV A7,B14	LD *B12,B5 || LD *A6,A11 || MV A7,B14
LD *A2,B2	LD *B14,B7	LD *B14,B7 || MV A8,B0
LD *A5,B5	MV A8,B0	MV A9,B2
LD *A7,A4	MV A9,B2	MV A10,B4
LD *A6,B6	MV A10,B4	MV A11,B6
NOP	MV A11,B6	NOP
MV A8,B1		
MV A9,B3		
MV A4,B7		
12 cycles	9 cycles	9 cycles (15min)

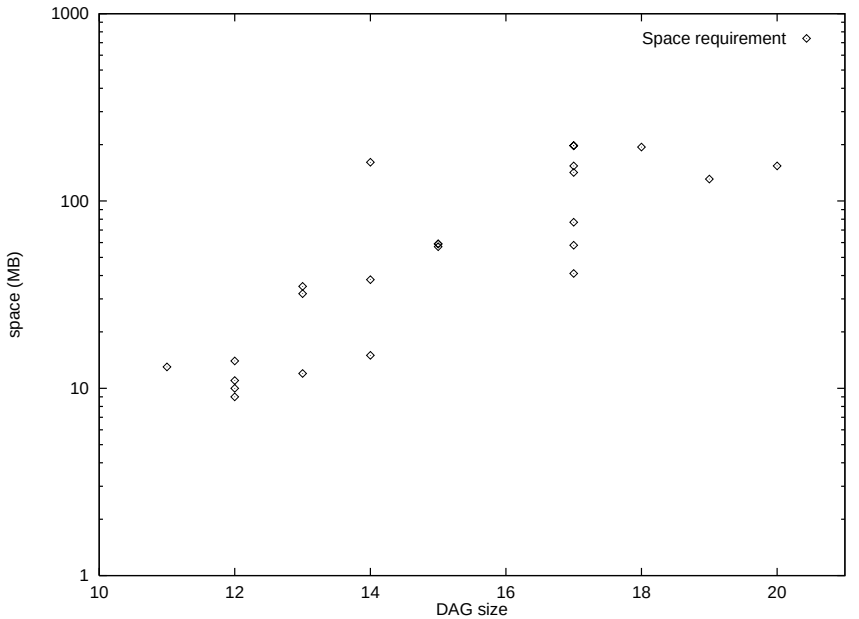
Results for TI-C62x variant: DSP benchmarks

Basic block	$\|V\|$	time[s]	space[MB]	#Impr.	#merged	#ESNodes
cplxinit	14	201	38	0	140183	44412
vectcopy init	12	16	11	26	61552	12286
vectcopy loop	14	47	15	58	149899	29035
matrixcopy loop	18	18372	194	1782	5172442	722012
vectsum loop	12	11	10	9	36715	8896
vectsum unrolled	17	1537	58	3143	1316684	198571
matrixsum loop	17	10527	154	2898	3502106	564058
dotproduct loop	17	3495	77	4360	2382094	345256
codebk_srch bb33	17	431	41	306	319648	64948
codebk_srch bb29	13	7	12	0	17221	6433
codebk_srch bb26	11	11	13	144	73761	19275
vecsum_c bb6	20	9749	154	5920	3744583	499740
vec_max bb8	13	79	35	0	99504	37254
...						

Time requirements (DSP benchmarks)



Space requirements (DSP benchmarks)



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>
    ...
    <transfer>
      <target id="MOVE" op0="r2" op1="r1">
        <use_fu="X2"/>
        <use_fu="L1"/>
      </target>
      ...
    </transfer>
  </instruction_set>
</architecture>
```