The offset assignment problem and its variants

Peter Marwedel
TU Dortmund
Informatik 12
Germany

2009/01/17

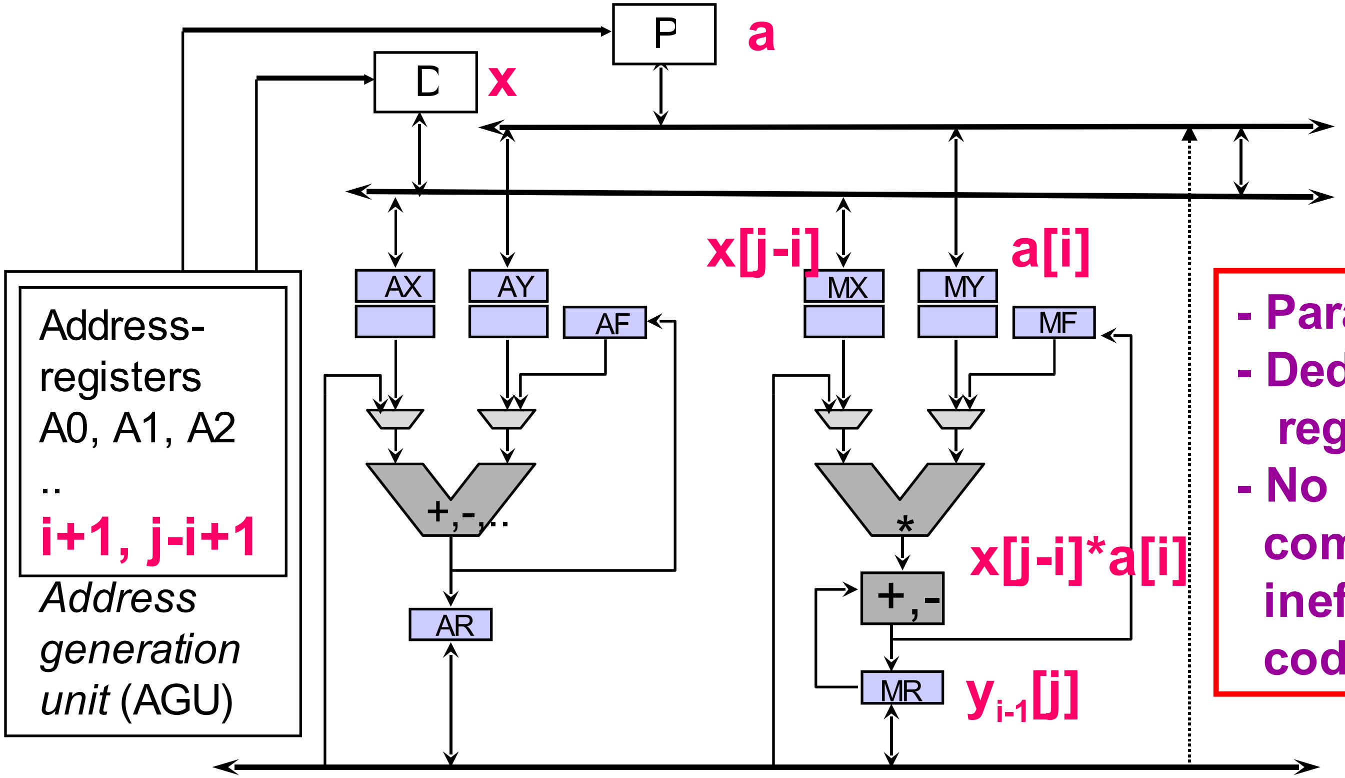
Graphics: © Alexandra Nolte, Gesine Marwedel, 2003

Reason for compiler-problems: Application-oriented Architectures

Application: u.a.: $y[j] = \sum_{i=0}^{n} x[j-i]*a[i]$

$\forall i: 0 \leq i \leq n: y_i[j] = y_{i-1}[j] + x[j-i]*a[i]$

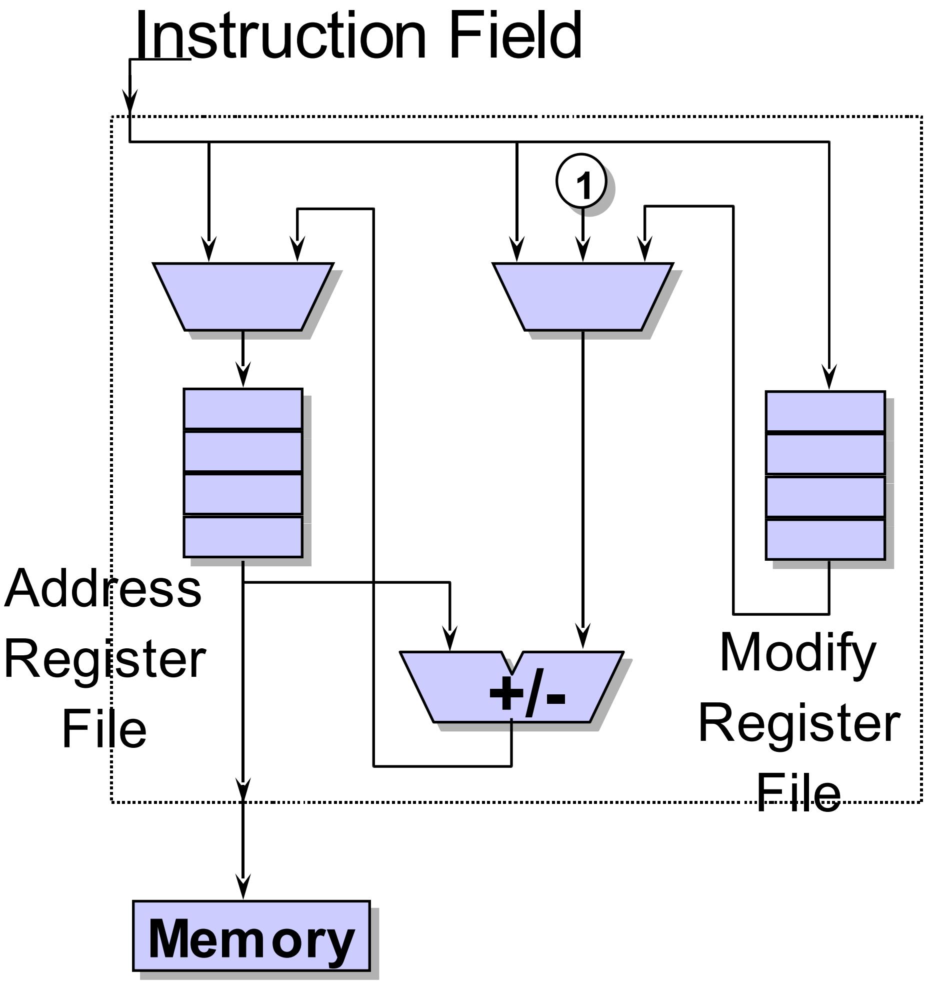
Architecture: Example: Data path ADSP210x



- Parallelism
- Dedicated registers
- No matching compiler ⇒ inefficient code

Exploitation of parallel address computations

Generic address generation unit (AGU) model



Parameters:

k = # address registers

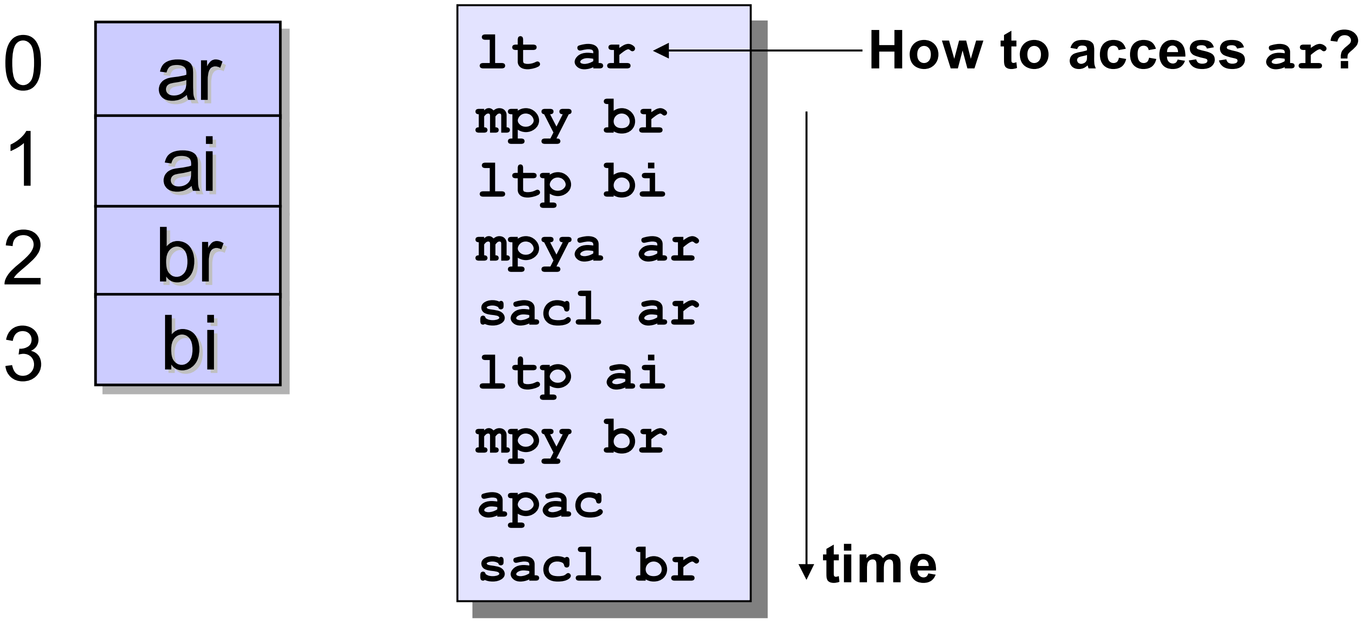
m = # modify registers

Cost metric for AGU operations:

Operation	cost
immediate AR load	1
immediate AR modify	1
auto-increment/ decrement	0
AR += MR	0

Address pointer assignment (APA)

Given: Memory layout + assembly code (without address code)



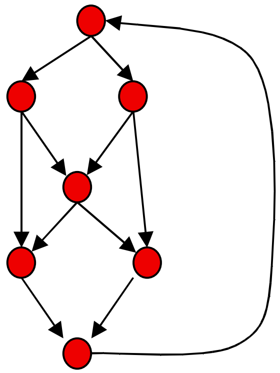
0	ar
1	ai
2	br
3	bi

```
lt ar
mpy br
ltp bi
mpya ar
sacl ar
ltp ai
mpy br
apac
sacl br
```

How to access `ar`?

Address pointer assignment (APA) is the sub-problem of finding an allocation of address registers for a given memory layout and a given schedule.

General approach: Minimum Cost Circulation Problem



Let $G = (V,E,u,c)$, with (V,E): directed graph

- u: $E \rightarrow \mathbb{R}_{\geq 0}$ is a capacity function,
- c: $E \rightarrow \mathbb{R}$ is a cost function; $n = |V|$, $m = |E|$.

Definition:

1. g: $E \rightarrow \mathbb{R}_{\geq 0}$ is called a **circulation** if it satisfies :

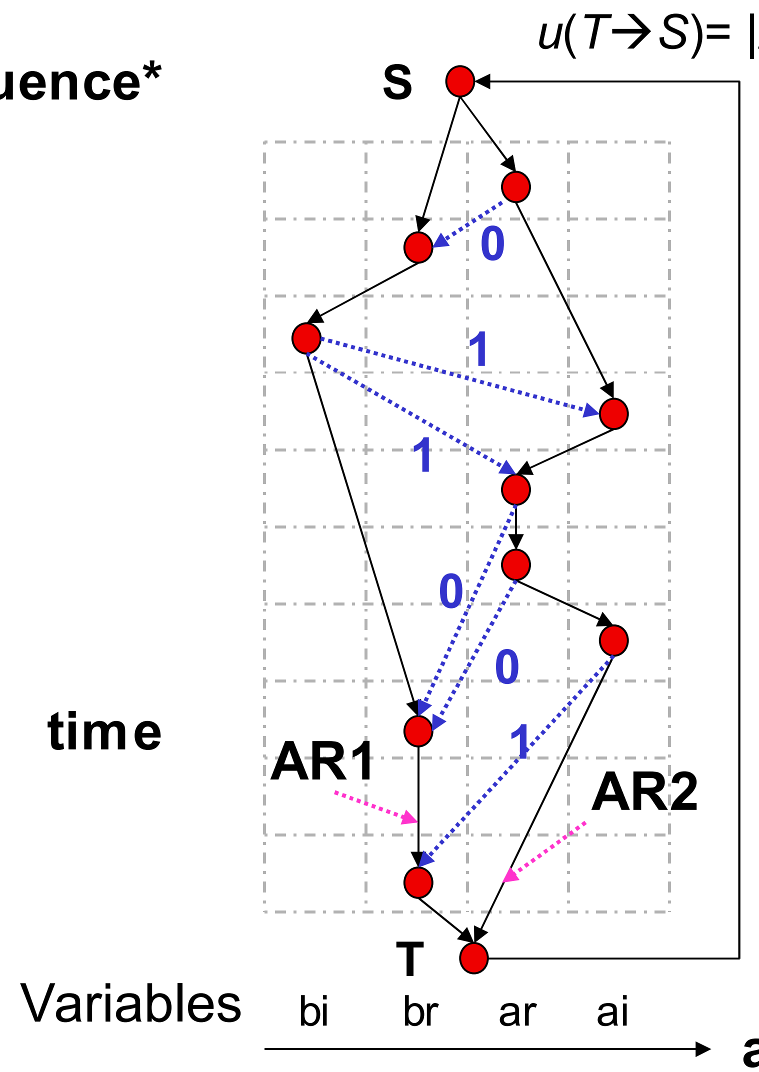
   $\forall\ v \in V: \sum_{w \in V:(v,w) \in E} g(v,w) = \sum_{w \in V:(w,v) \in E} g(w,v)$ (flow conservation)
2. g is **feasible** if $\forall (v,w) \in E$: $g(v,w) \leq u(v,w)$ (capacity constraints)
3. The cost of a circulation g is $c(g) = \sum_{(v,w) \in E} c(v,w)\, g(v,w)$.
4. There may be a lower bound on the flow through an edge.
5. The **minimum cost circulation problem** is to find a feasible circulation of minimum cost.

[K.D. Wayne: A Polynomial Combinatorial Algorithm for Generalized Minimum Cost Flow, http://www.cs.princeton.edu/ ~wayne/ papers/ ratio.pdf]

Mapping APA to the Minimum Cost Circulation Problem

Assembly sequence*

```
lt ar
mpy br
ltp bi
mpy ai
mpya ar
sacl ar
ltp ai
mpy br
apac
sacl br
```



Flow into and out of variable nodes must be 1. Replace variable nodes by edge with lower bound=1 to obtain pure circulation problem

↓ **circulation selected**

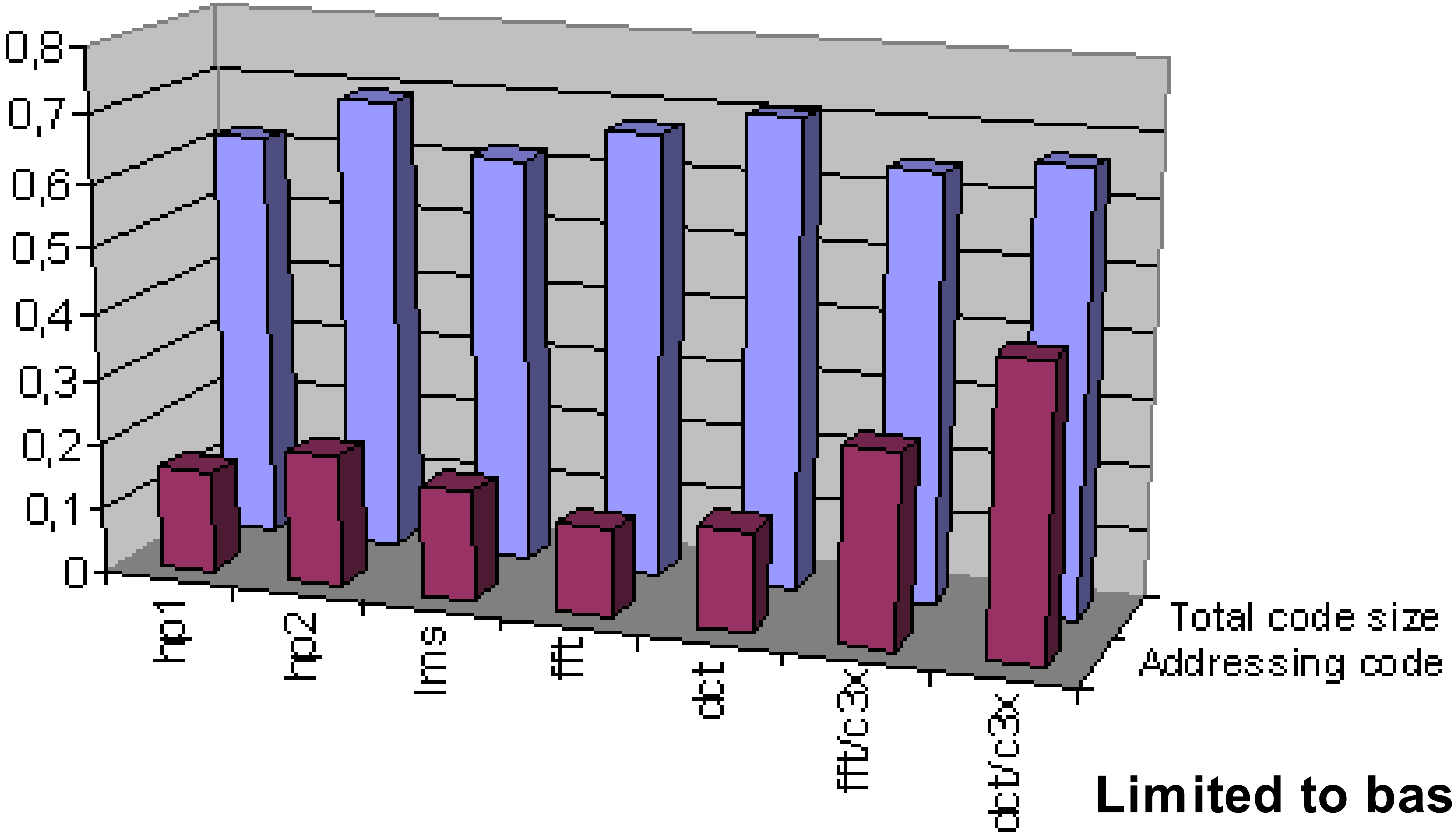
⇣ **additional edges of original graph (only samples shown)**

* C2x processor from ti

[C. Gebotys: DSP Address Optimization Using A Minimum Cost Circulation Technique, *ICCAD*, 1997]

Results according to Gebotys

$$\frac{\text{Optimized code size}}{\text{Original code size}}$$



Limited to basic blocks

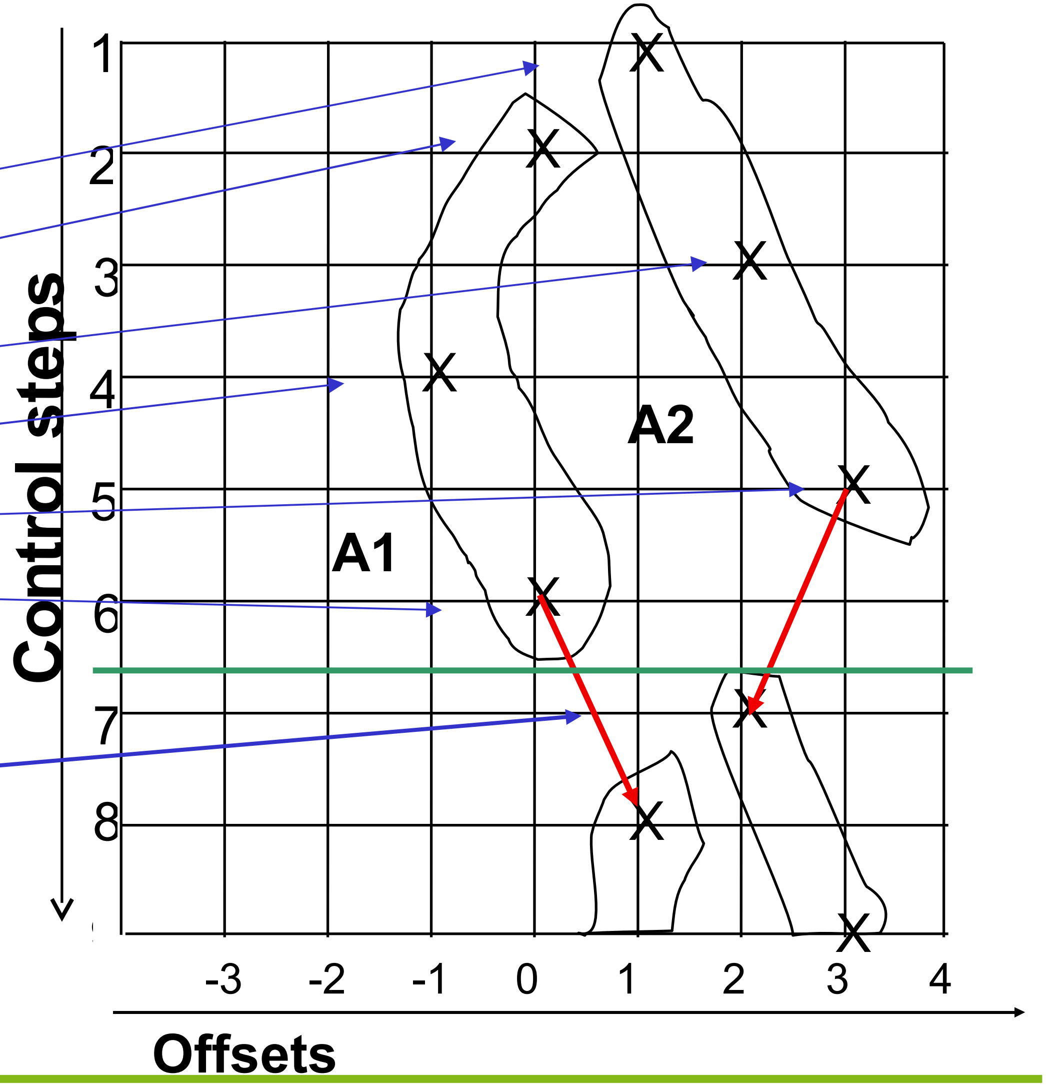
Beyond basic blocks:
- handling array references in loops -

Example:

```
for (i=2; i<=N; i++)
{ .. B[i+1]     /*A2++   */

  .. B[i]       /*A1--   */
  .. B[i+2]     /*A2++  */
  .. B[i-1]     /*A1++  */
  .. B[i+3]     /*A2--   */
  .. B[i] .. }  /*A1++  */
```

Cost for crossing loop boundaries considered.



Reference: A. Basu, R. Leupers, P. Marwedel: Array Index Allocation under Register Constraints, Int. Conf. on VLSI Design, Goa/India, 1999

Offset assignment problem (OA)
- Effect of optimised memory layout -

Let's assume that we can modify the memory layout

- ☞ offset assignment problem.

(*k*,*m*,*r*)-OA is the problem of generating a memory layout which minimizes the cost of addressing variables, with

- ☞ *k*: number of address registers
- ☞ *m*: number of modify registers
- ☞ *r*: the offset range

The case (1,0,1) is called simple offset assignment (SOA),

the case (*k*,0,1) is called general offset assignment (GOA).

☞ SOA example
- Effect of optimised memory layout -

Variables in a basic block: Access sequence:

V = {*a*, *b*, *c*, *d*} *S* = (*b*, *d*, *a*, *c*, *d*, *c*)

0	a
1	b
2	c
3	d

```
Load AR,1  ;b
AR += 2    ;d
AR -= 3    ;a
AR += 2    ;c
AR ++      ;d
AR --      ;c
```

cost: 4

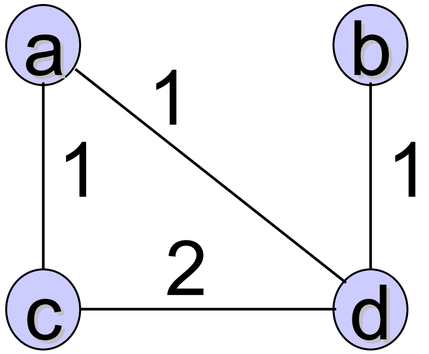
0	b
1	d
2	c
3	a

```
Load AR,0  ;b
AR ++      ;d
AR +=2     ;a
AR --      ;c
AR --      ;d
AR ++      ;c
```

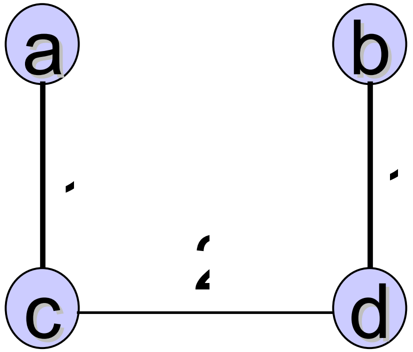
cost: 2

SOA example: Access sequence, access graph and Hamiltonian paths

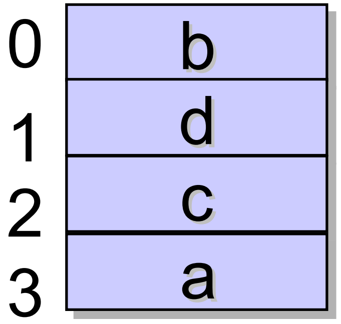
access sequence: b d a c d c



access graph



maximum weighted path= max. weighted Hamilton path covering (MWHC)



0	b
1	d
2	c
3	a

memory layout

SOA used as a building block for more complex situations

➡ significant interest in good SOA algorithms

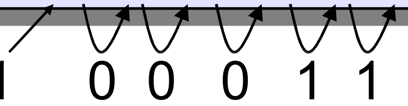
[Bartley, 1992; Liao, 1995]

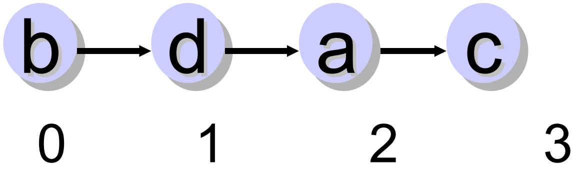
Naïve SOA

Nodes are added in the order
in which they are used in the program.

Example:

Access sequence: *S* = *(b, d, a, c, d, c)*





0	b
1	d
2	a
3	c

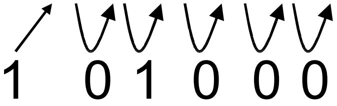
memory layout

Liao's algorithm

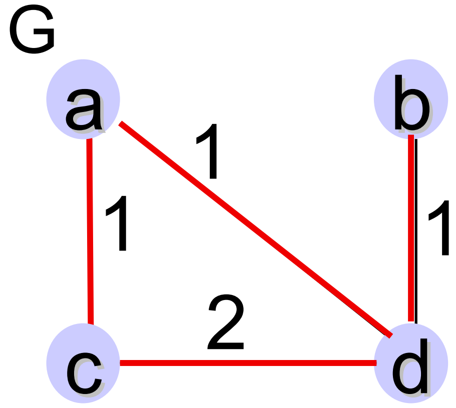
Similar to Kruskal's spanning tree algorithms:

1. Sort edges of access graph G=(V,E) according to their weight
2. Construct a new graph G'=(V',E'), starting with E'=0
3. Select an edge *e* of G of highest weight; If this edge does not cause a cycle in G' and does not cause any node in G' to have a degree > 2 then add this node to E' otherwise discard *e*.
4. Goto 3 as long as not all edges from G have been selected and as long as G' has less than the maximum number of edges (|V|-1).

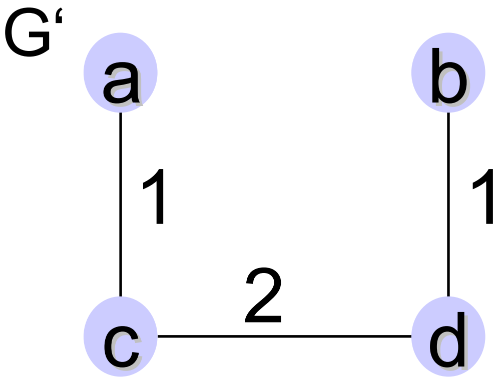
Example: Access sequence: *S=(b, d, a, c, d, c)*



Implicit edges of weight 0 for all unconnected nodes.



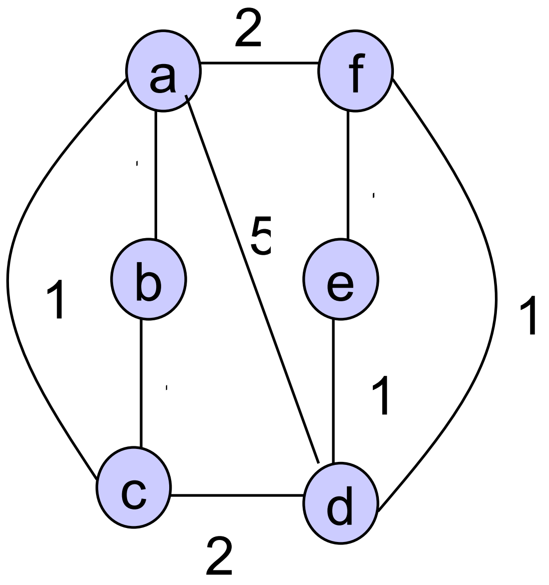
2 (c,d)
1 (a,c)
1 (a,d)
1 (b,d)



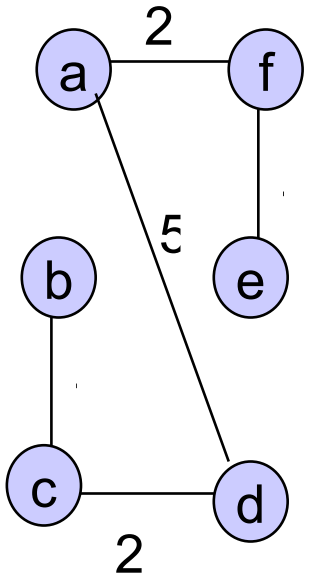
Liao’s algorithm
on a more complex graph

a b c d e f a d a d a c d f a d

G



G’

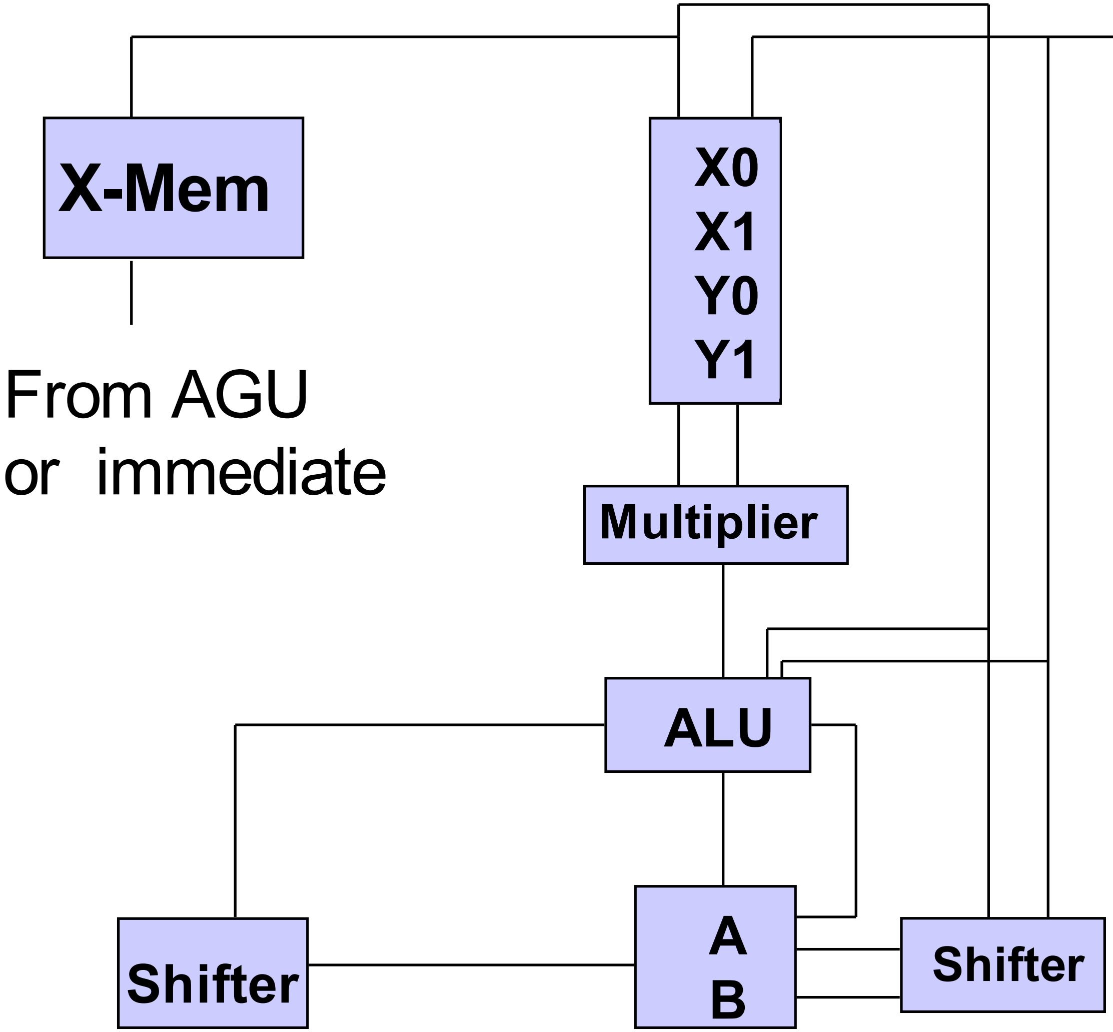


Additional compiler optimizations

Peter Marwedel
TU Dortmund
Informatik 12
Germany

Graphics: © Alexandra Nolte, Gesine Marwedel, 2003

Multiple memory banks
- Sample hardware -



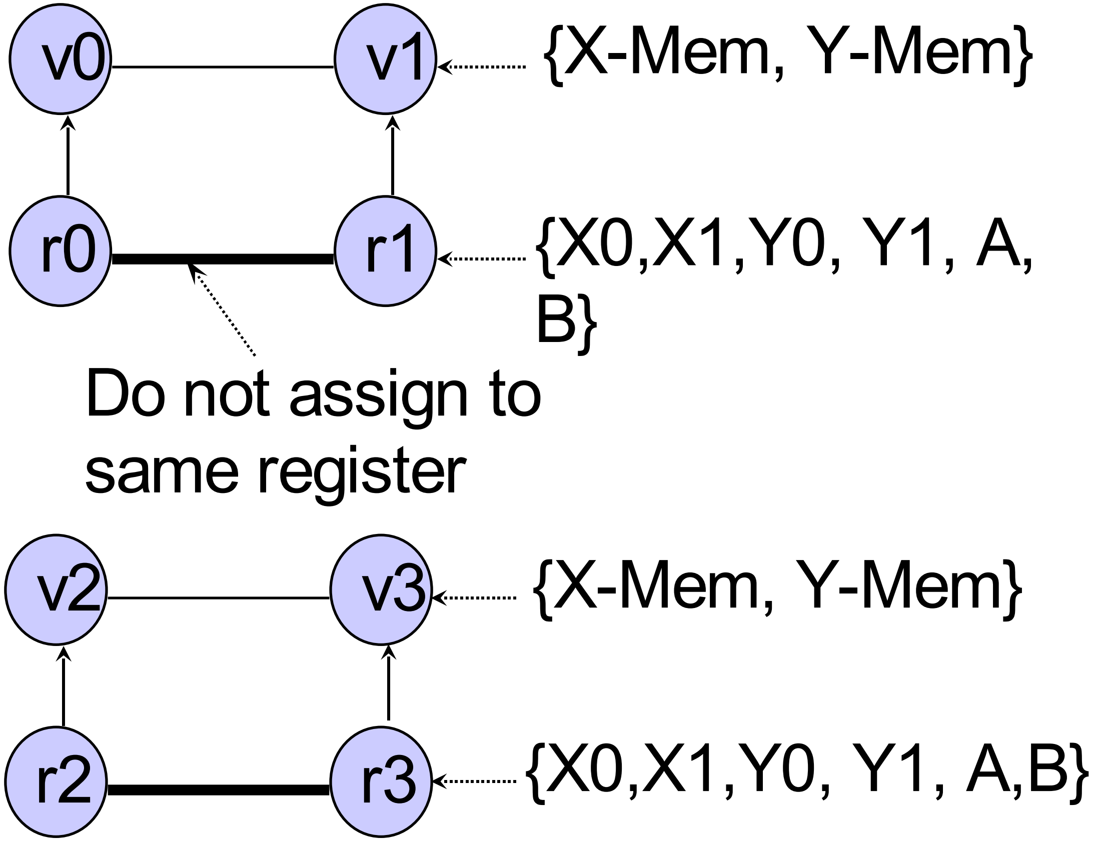
Parallel moves possible if using different memory banks

Multiple memory banks
- Constraint graph generation -

Precompacted code (symbolic variables and registers)

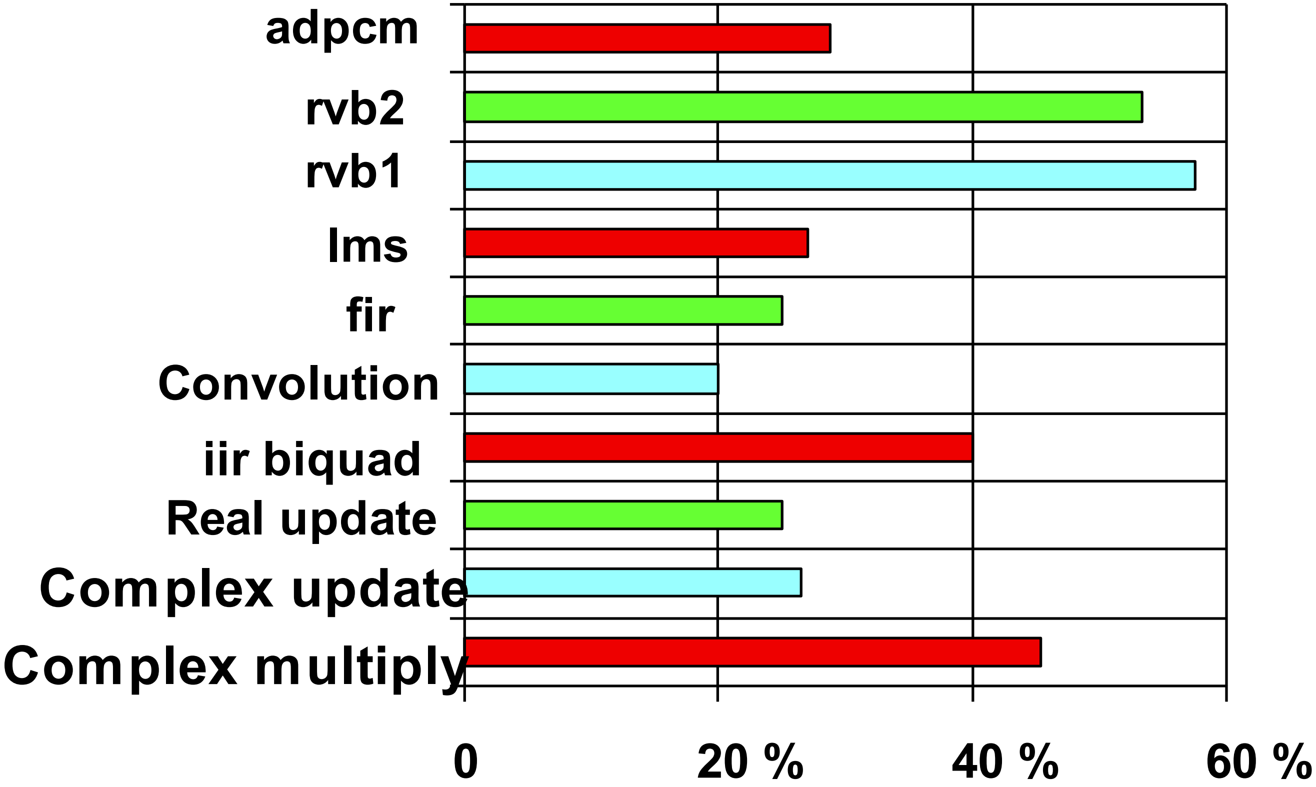
Move v0,r0 v1,r1
Move v2,r2 v3,r3

Constraint graph



Links maintained, more constraints ...

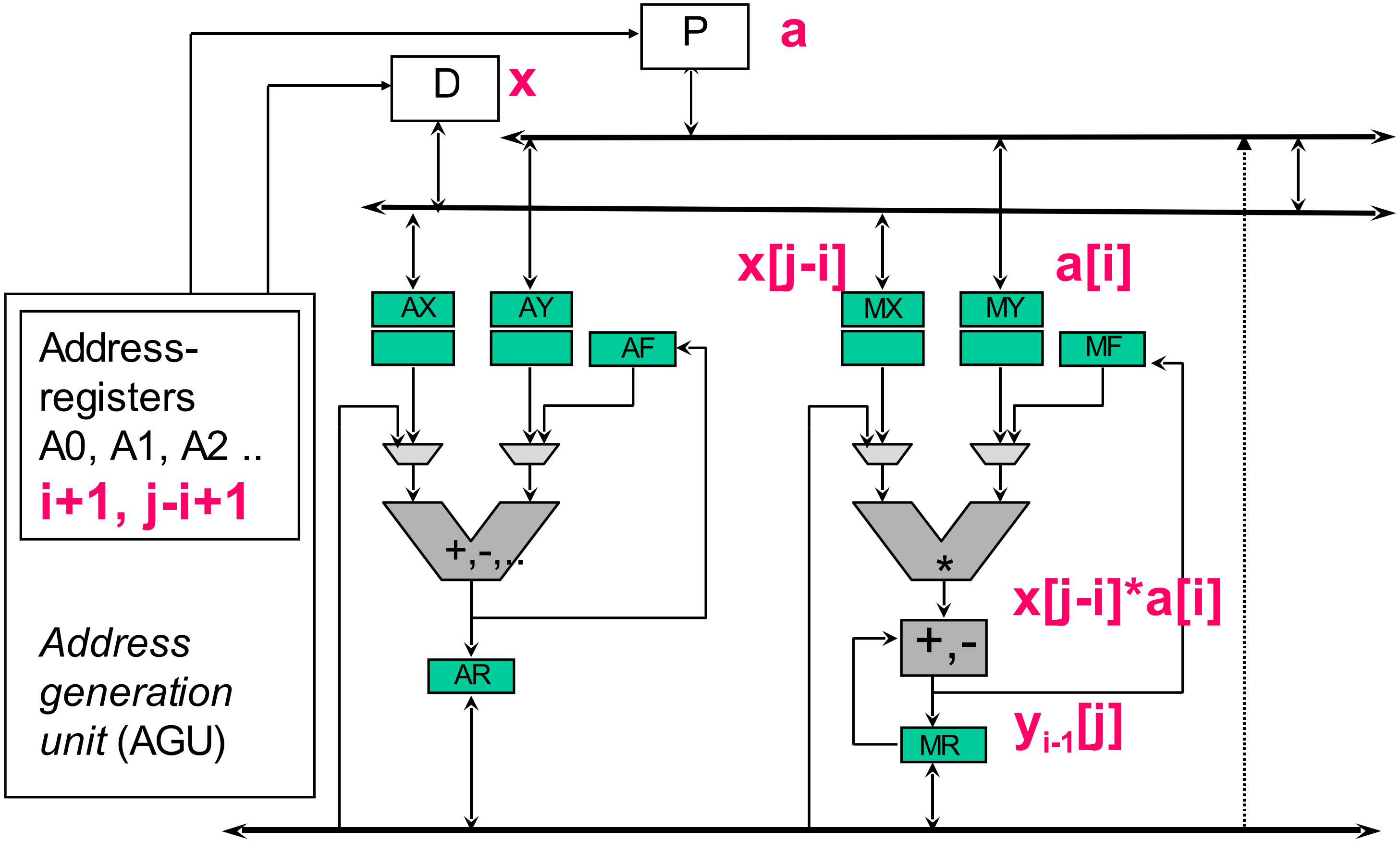
Multiple memory banks
Code size reduction through simulated annealing



[Sudarsanam, Malik, 1995]

Exploitation of instruction level parallelism (ILP)

Several transfers in the same cycle:



Exploitation of instruction level parallelism (ILP)

```
1: MR := MR+(MX*MY);
2: MX:=D[A1];
3: MY:=P[A2];
4: A1- -;
5: A2++;
6: D[0]:= MR;
.....
```

```
1´: MR := MR+(MX*MY), MX:=D[A1],
     MY:=P[A2], A1- -, A2++;
2´: D[0]:= MR;
```

Modelling of possible parallelism using n-ary compatibility relation, e.g. ~(1,2,3,4,5)

Generation of integer programming (IP)- model (max. 50 statements/model)

Using standard-IP-solver to solve equations

Exploitation of instruction level parallelism (ILP)

- From 9 to 7 cycles through compaction -

```
u(n) = u(n - 1) + K0 × e(n) + K1 × e(n - 1);
e(n - 1)= e(n)
```

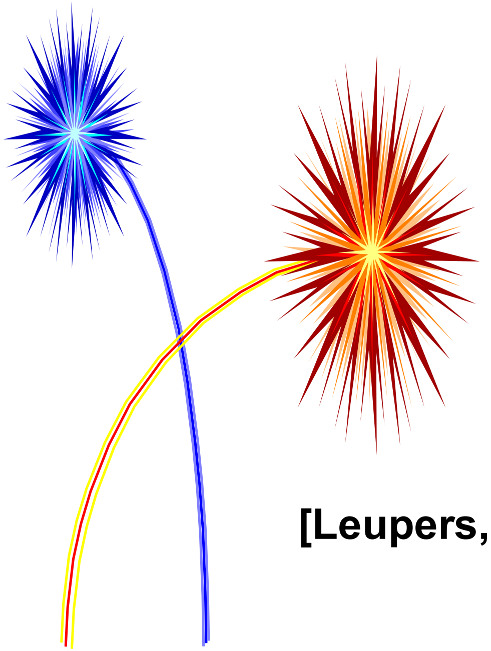
```
ACCU   := u(n - 1)
TR     := e(n - 1)
PR     := TR × K1
TR     := e(n)
e(n - 1)  := e(n)
ACCU   := ACCU + PR
PR     := TR × K0
ACCU   := ACCU + PR
u(n)   := ACCU
```

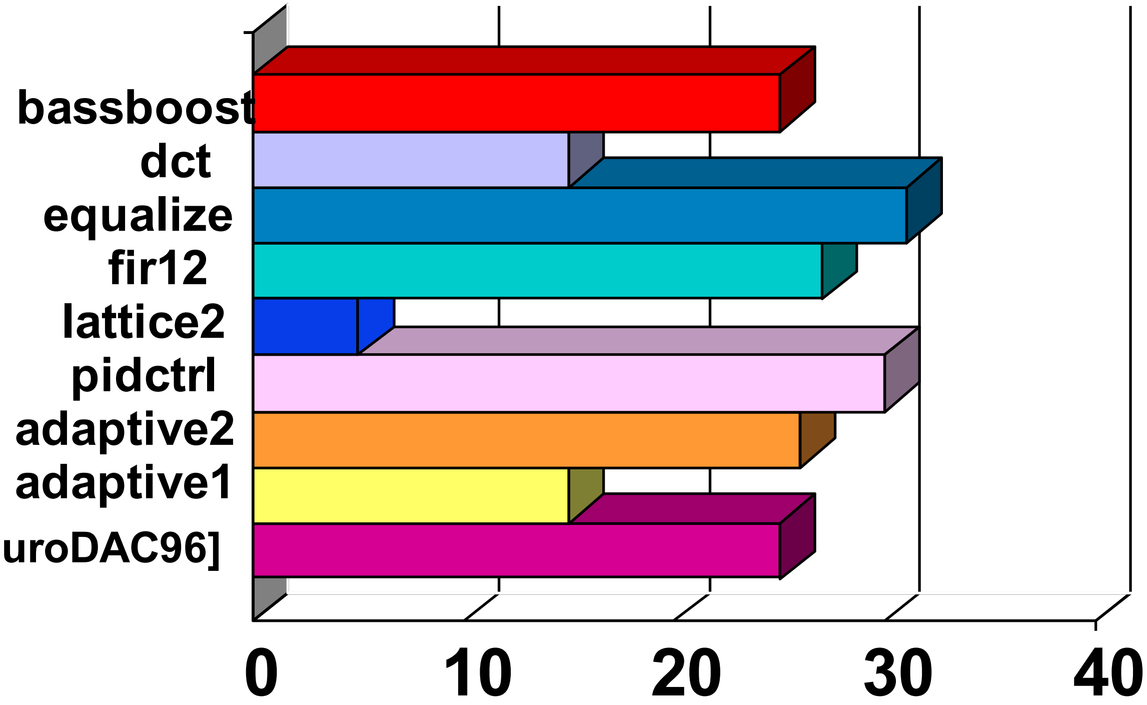
```
ACCU:= u(n - 1)
TR     := e(n - 1)
PR     := TR × K1
e(n - 1):= e(n) || TR:= e(n)  ||
                ACCU:= ACCU + PR
PR     := TR × K0
ACCU:= ACCU + PR
u(n)    := ACCU
```

Exploitation of instruction level parallelism (ILP)

Results obtained through integer programming:

Code size reduction [%]





[Leupers, EuroDAC96]

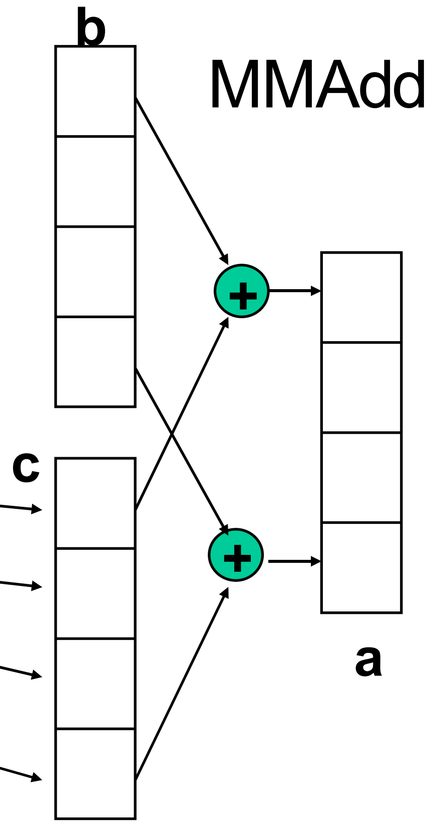
Compaction times: 2 .. 35 sec

Exploitation of Multimedia Instructions

```
FOR i:=0 TO n DO
 a[i] = b[i] + c[i]
```

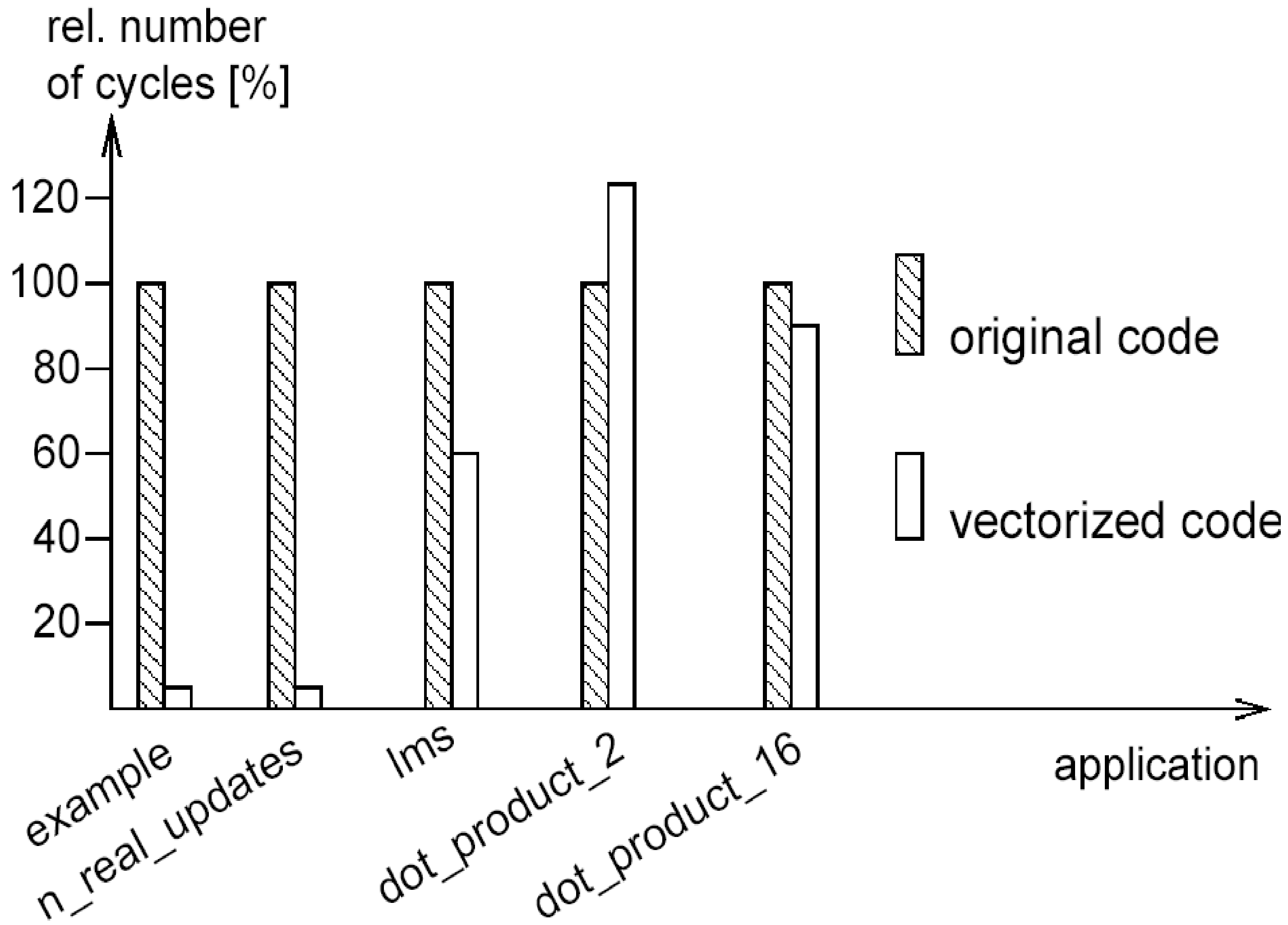
⇩

```
FOR i:=0 STEP 4 TO n DO
 a[i  ]=b[i  ]+c[i ];
 a[i+1]=b[i+1]+c[i+1];
 a[i+2]=b[i+2]+c[i+2];
 a[i+3]=b[i+3]+c[i+3];
```



MMAdd (4 x 8/16 bit)

Improvements for M3 DSP due to vectorization

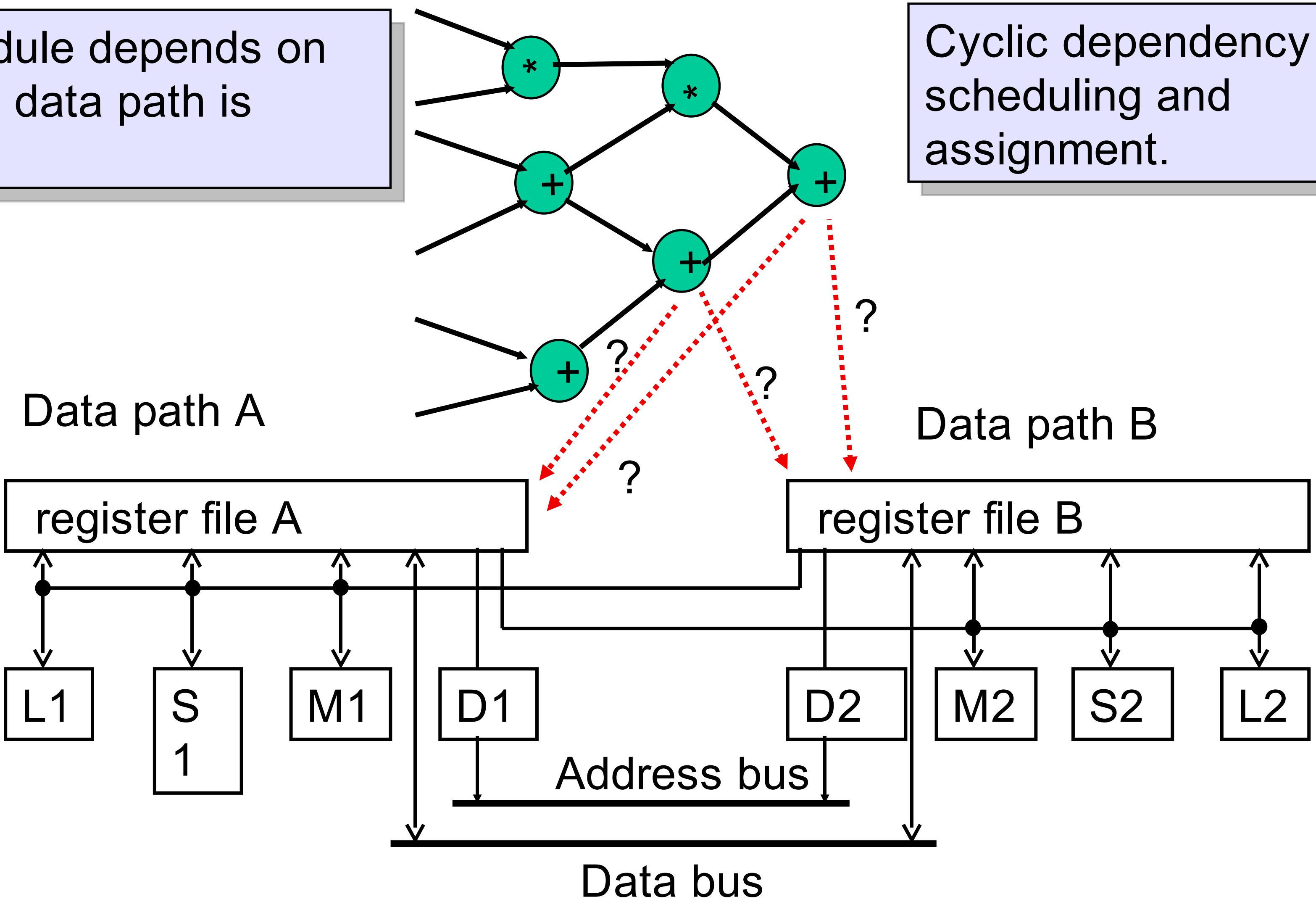


Scheduling for partitioned data paths

Schedule depends on which data path is used.

Cyclic dependency of scheduling and assignment.

‘C6x:

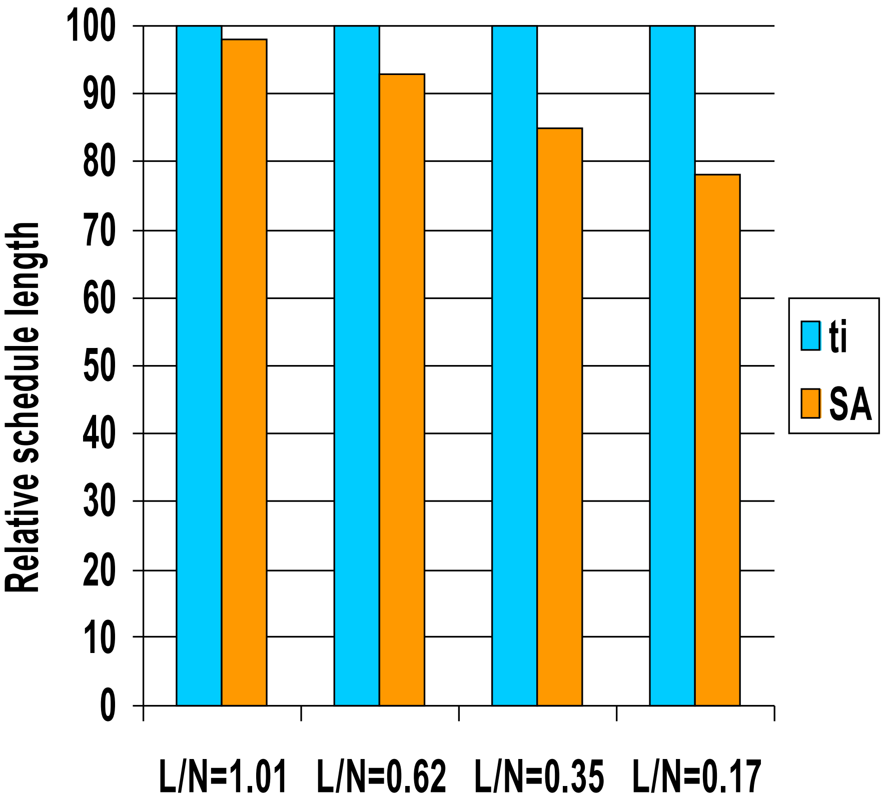


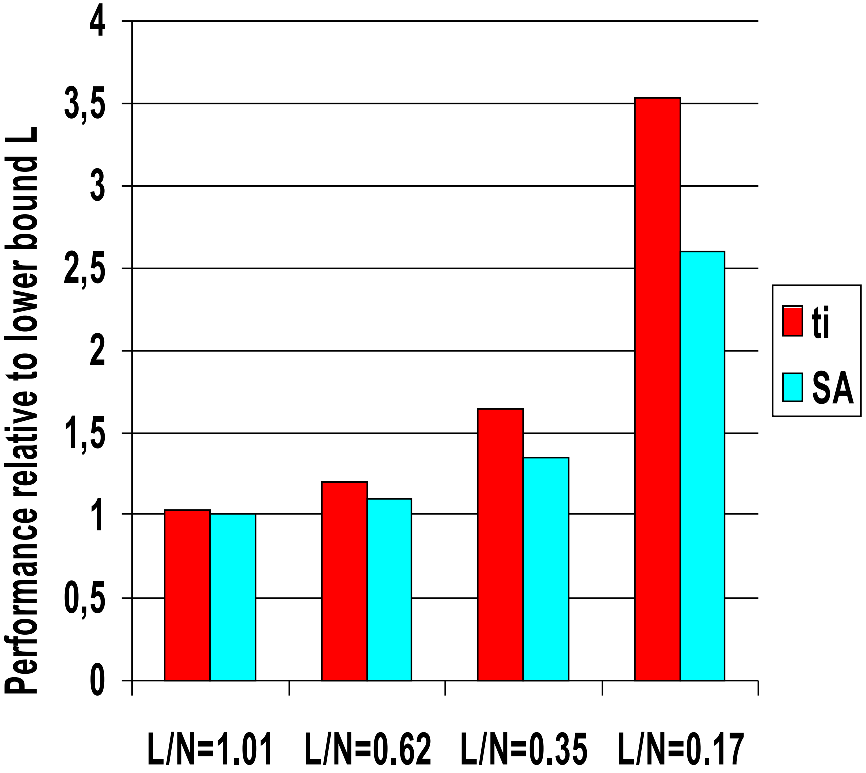
Integrated scheduling and assignment using Simulated Annealing (SA)

```
algorithm Partition
input DFG G with nodes;
output: DP: array [1..N] of 0,1 ;
var int i, r, cost, mincost;
 float T;
 begin
   T=10;
   DP:=Randompartitioning;
   mincost :=
    LISTSCHEDULING(G,D,P);
   WHILE_LOOP;
   return DP;
 end.
```

```
WHILE_LOOP:
while T>0.01 do
 for i=1 to 50 do
  r:= RANDOM(1,n);
  DP[r] := 1-DP[r];
  cost:=LISTSCHEDULING(G,D,P);
  delta:=cost-mincost;
  if delta <0 or
    RANDOM(0,1)<exp(-delta/T)
    then mincost:=cost
    else DP[r]:=1-DP[r]
   end if;
  end for;
  T:= 0.9 * T;
end while;
```

Results: relative schedule length as a function of the "width" of the DFG



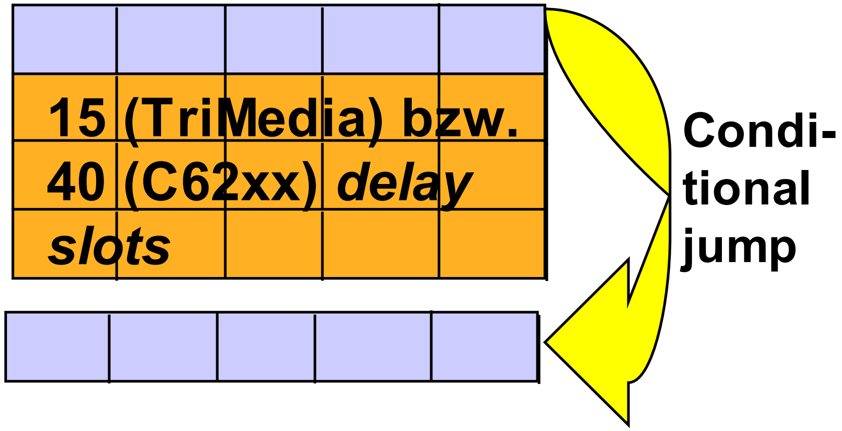


SA approach outperforms the ti approach for "wide" DFGs (containing a lot of parallelism)

For wide DFGs, SA algorithm is able of "staying closer" critical path length.

VLIW (very long instruction word) DSPs

Large *branch delay penalty:*



Avoiding this penalty:
predicated execution:

[c] *instruction*

c=*true*: instruction executed

c=*false*: effectively NOOP

Realisation of *if-statements*

with conditional jumps or with *predicated execution:*

```
if (c)
{ a = x + y;
  b = x + z;
}
else
{ a = x - y;
  b = x - z;
}
```

Cond. instructions:

```
   [c]  ADD x,y,a
|| [c]  ADD x,z,b
|| [!c] SUB  x,y,a
|| [!c] SUB  x,z,b
```

1 cycle

Cost of implementation methods for IF-Statements

Sourcecode: if (c1) {t1; if (c2) t2}

No precondition (no enclosing IF or enclosing IFs implemented with cond. jumps)

1. Conditional jump:

```
BNE c1, L;
t1;
L: ...
```

2. Conditional Instruction:

```
[c1] t1
```

Precondition (enclosing IF not implemented with conditional jumps)

3. Conditional jump :

```
[c1] c:=c2
[~c1] c:=0
BNE c, L;
t2;
L: ...
```

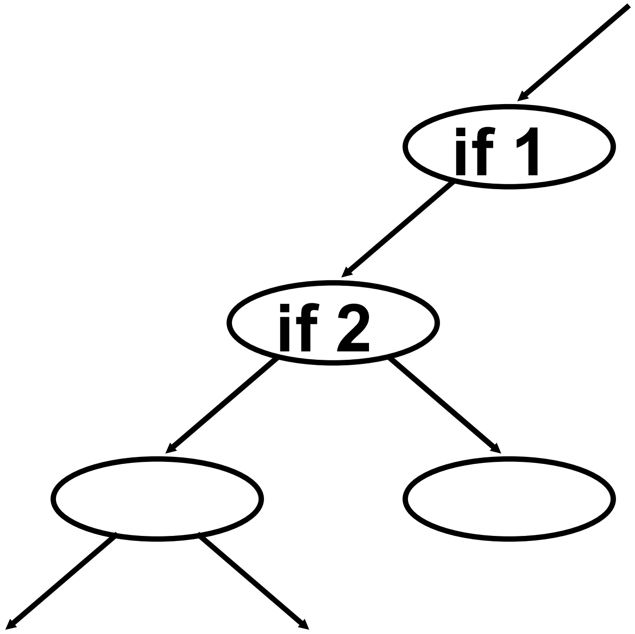
4. Conditional Instruction :

```
[c1] c:=c2
[~c1] c:=0
[c] t2
```

Additional computations to compute effective condition c

Optimization for nested IF-statements

Goal: compute fastest implementation for all IF-statements



- Selection of fastest implementation for if-1 requires knowledge of how fast if-2 can be implemented.
- Execution time of if-2 depends on setup code, and, hence, also on how if 1 is implemented
- cyclic dependency!

Dynamic programming algorithm (phase 1)

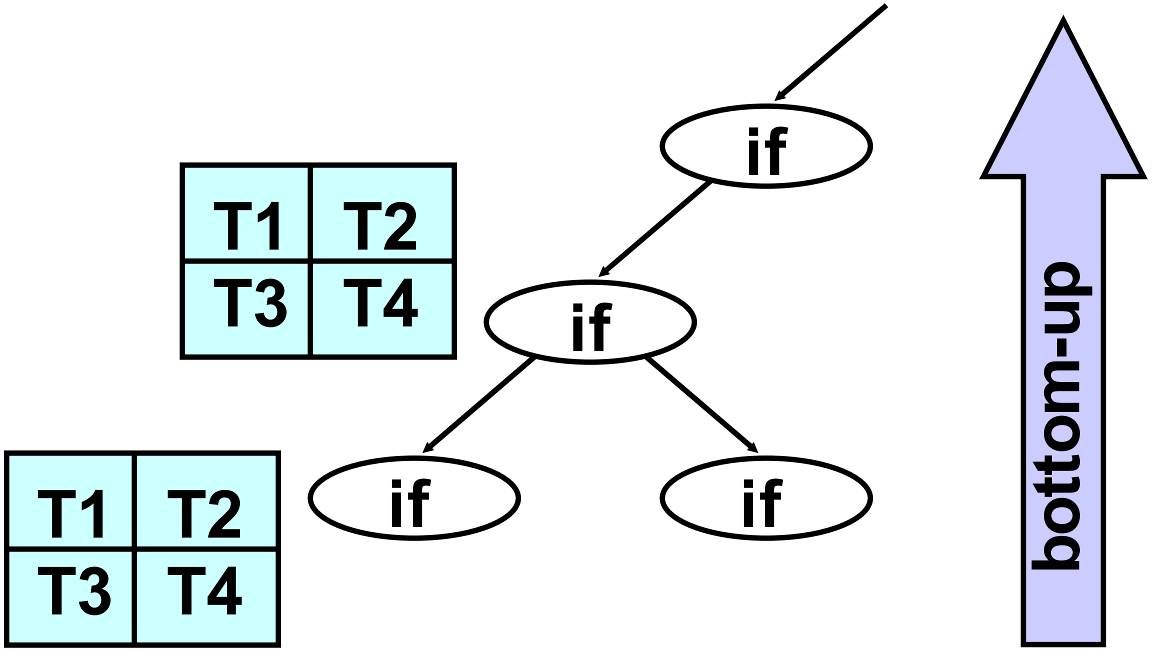
For each if-statement compute 4 cost values:

T1 : cond. jump, no precondition

T2 : cond. instructions, no precondition

T3 : cond. jump, with precondition

T4: cond. instructions, with precondition

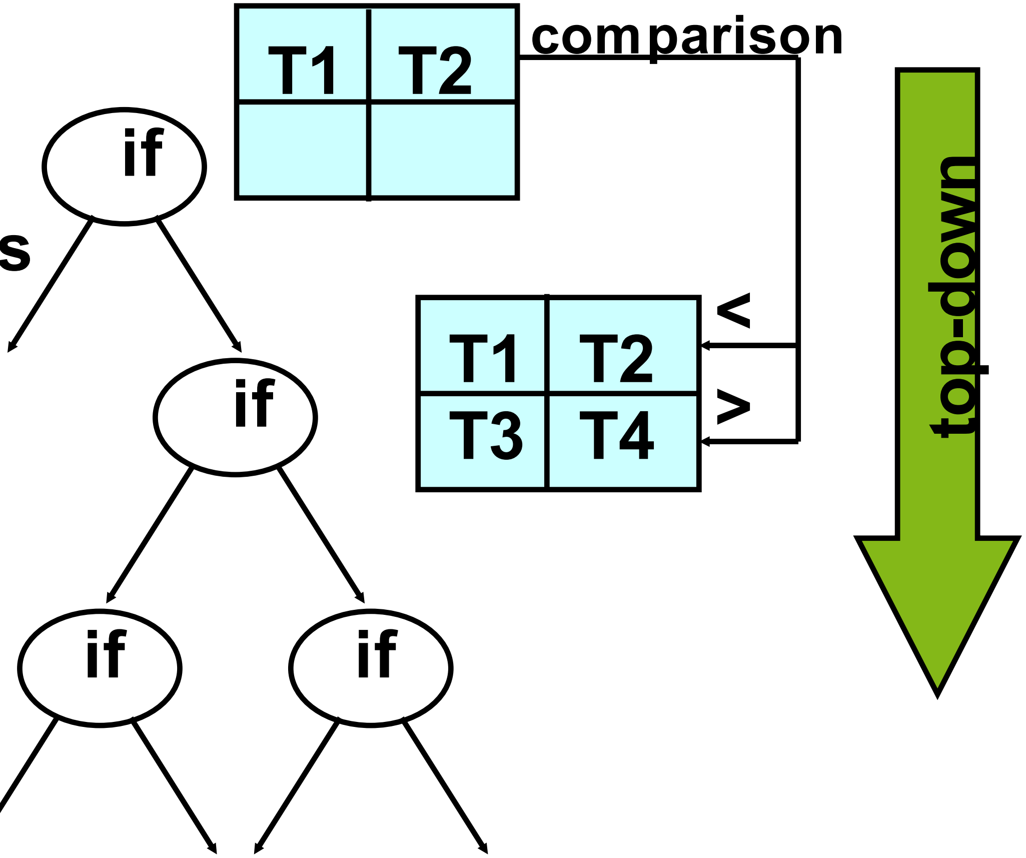


Dynamic programming (phase 2)

No precondition for top-level IF-statement. Hence, comparison „T1 < T2“ suffices.

T1 < T2:
cond. branch faster, no precondition for nested IF-statements

T1 > T2:
cond. instructions faster, precondition for nested IF-statements



Results: TI C62xx

Runtimes (max) for 10 control-dominated examples

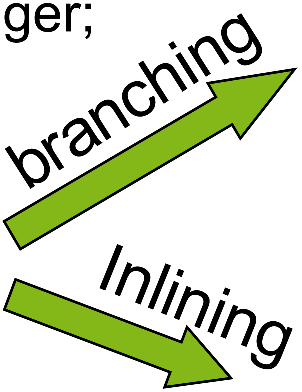
Example	Conditional jumps	Conditional instructions	Dynamic program.	Min (col. 2-5)	TI C compiler
1	21	11	11	11	15
2	12	13	13	12	13
3	26	21	22	21	27
4	9	12	12	9	10
5	26	30	24	24	21
6	32	23	23	23	30
7	57	173	49	49	51
8	39	244	30	30	41
9	28	27	27	27	29
10	27	30	30	27	28

Average gain: 12%

Function inlining: advantages and limitations

Advantage: low calling overhead

```
Function
sq(c:integer)
 return:integer;
begin
 return c*c
end;
....
a=sq(b);
....
```



```
push PC;
push b;
BRA sq;
 pull R1;
 mul R1,R1,R1;
 pull R2;
 push R1;
 BRA (R2)+1;
pull R1;
ST R1,a;
```

```
....
LD R1,b;
MUL R1,R1,R1;
ST R1,a
```

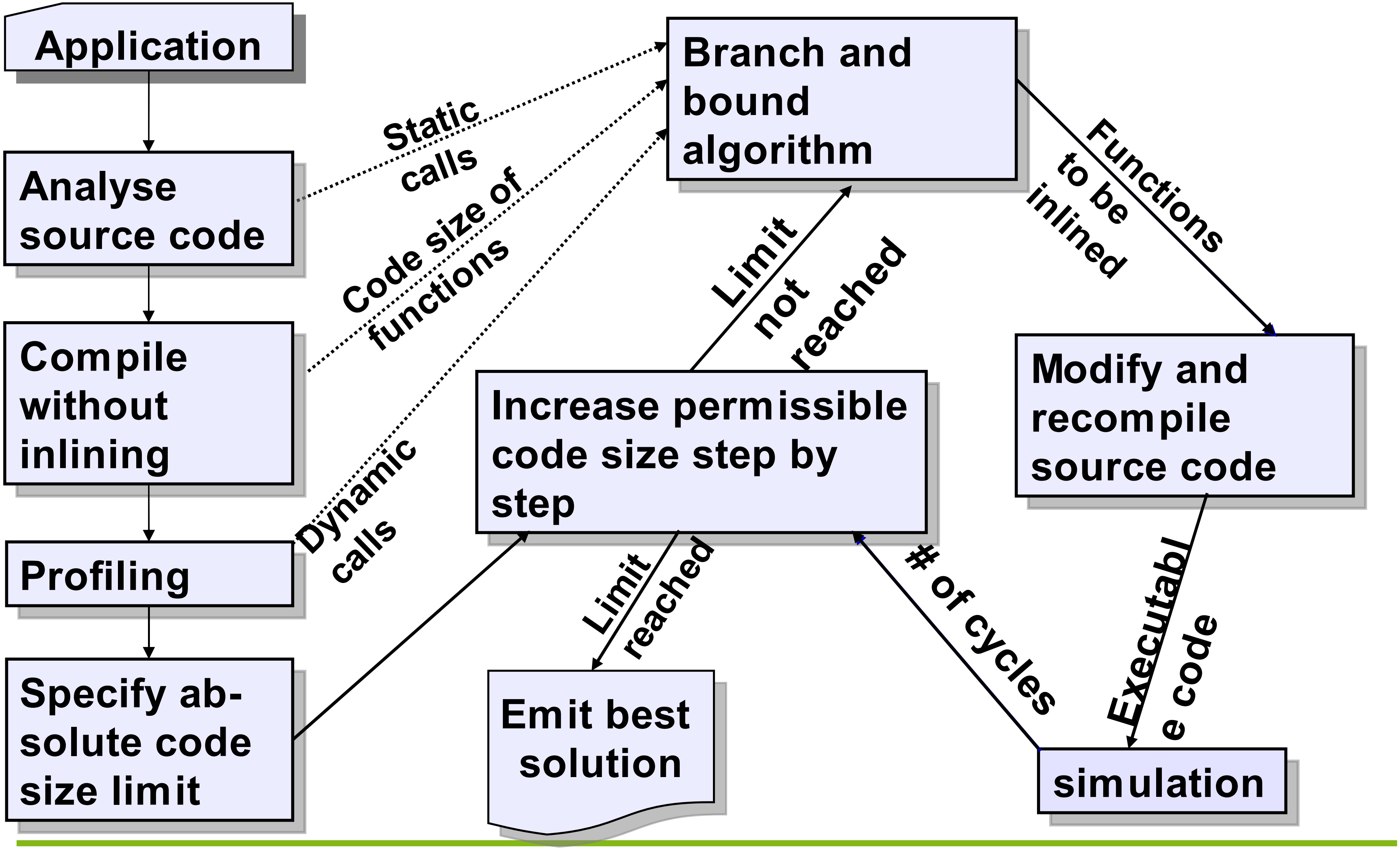
Limitations:

- Not all functions are candidates.
- Code size explosion.
- Requires manual identification using ‘inline’ qualifier.

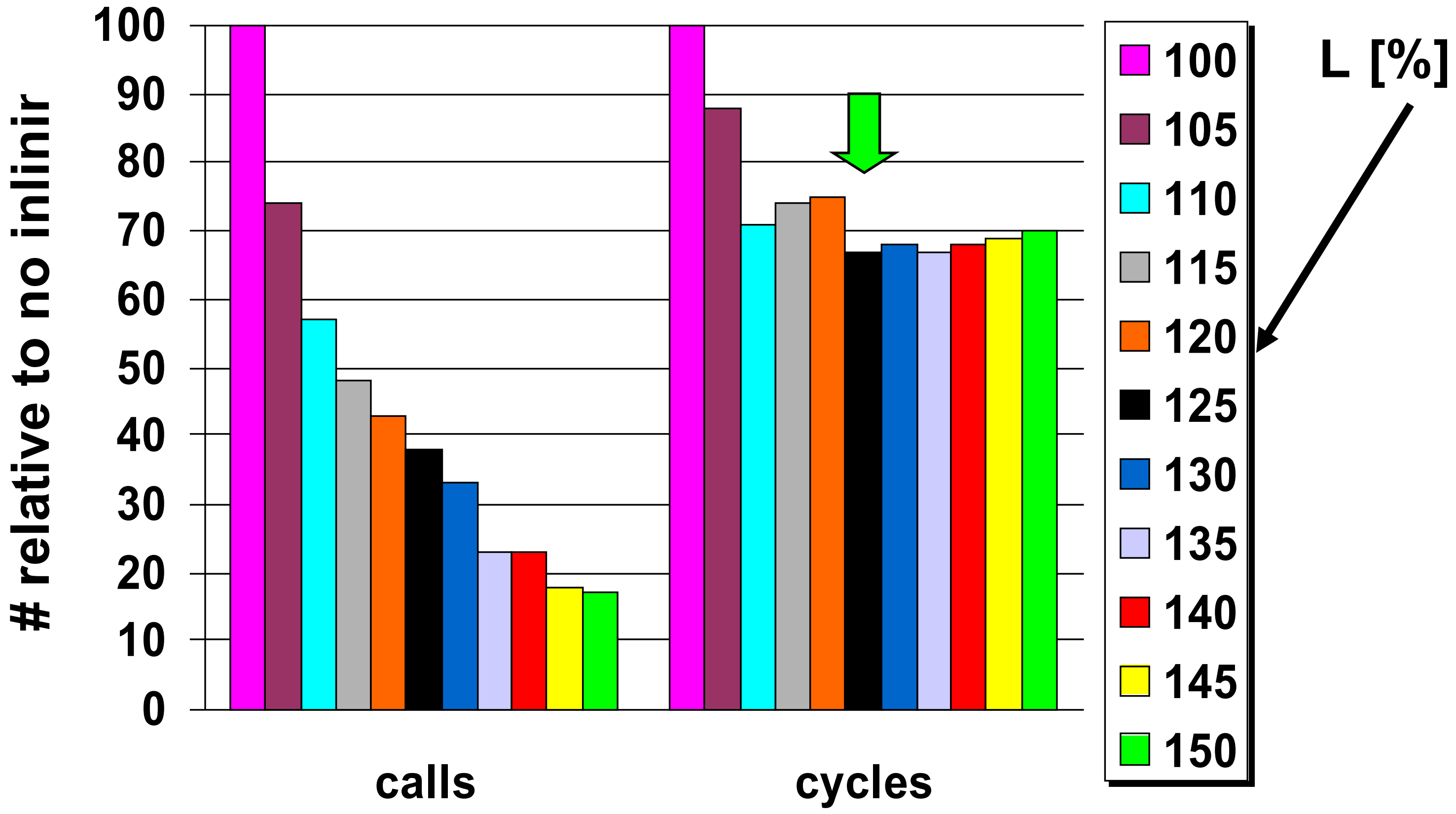
Goal:

- Controlled code size
- Automatic identification of suitable functions.

Design flow


Application
Analyse source code
Compile without inlining
Profiling
Specify ab-solute code size limit
Static calls
Code size of functions
Dynamic calls
Branch and bound algorithm
Functions to be inlined
Modify and recompile source code
Executabl e code
simulation
# of cycles
Increase permissible code size step by step
Limit not reached
Limit reached
Emit best solution

Results for GSM speech and channel encoder: #calls, #cycles (TI 'C62xx)



33% speedup for 25% increase in code size.
# of cycles not a monotonically decreasing function of the code size!

Inline vectors computed by B&B algorithm

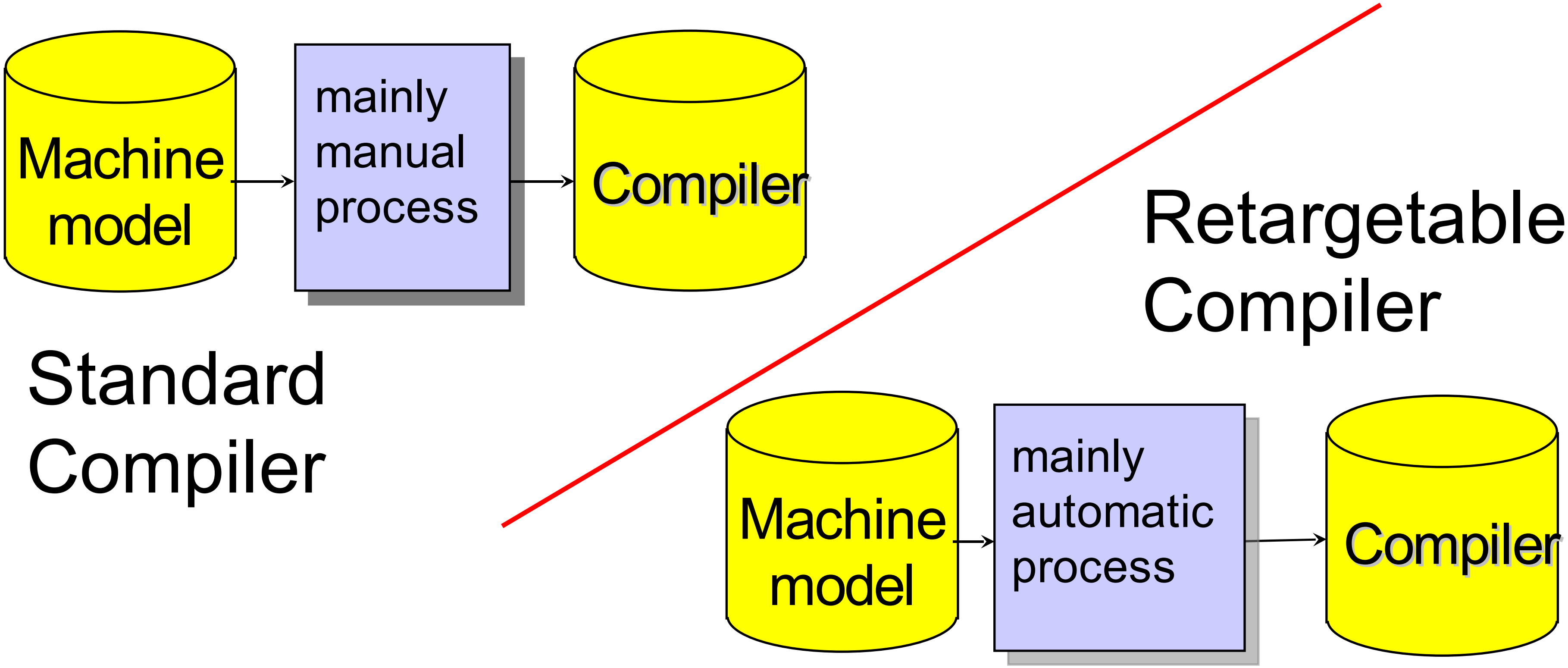
size limit (%)	inline vector (functions 1-26)
100	00000000000000000000000000
105	00100000001100001110111111
110	10111001011100001111111111
115	10110000000001001000111001
120	10110100101000100110111101
125	10110000001010000100111101
130	00110000000010100100111000
135	10110010001110101110111101
140	10111011111110101111111111
145	10110110101010100110111101
150	10110110000010110110111101

Major changes for each new size limit. Difficult to generate manually.

References:

- J. Teich, E. Zitzler, S.S. Bhattacharyya. 3D Exploration of Software Schedules for DSP Algorithms, CODES'99
- R. Leupers, P.Marwedel: Function Inlining under Code Size Constraints for Embedded Processors ICCAD, 1999

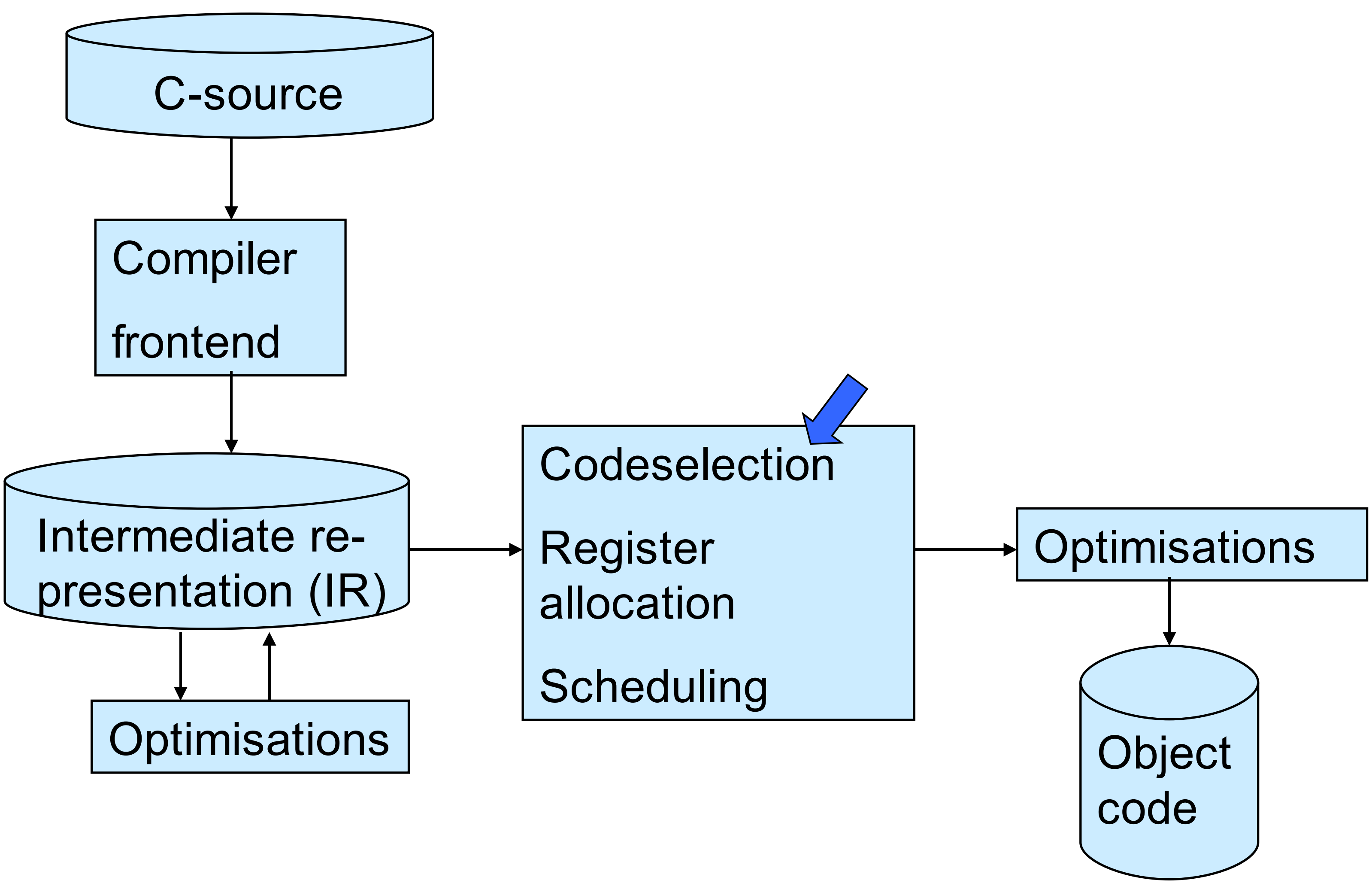
Retargetable Compilers vs. Standard Compilers



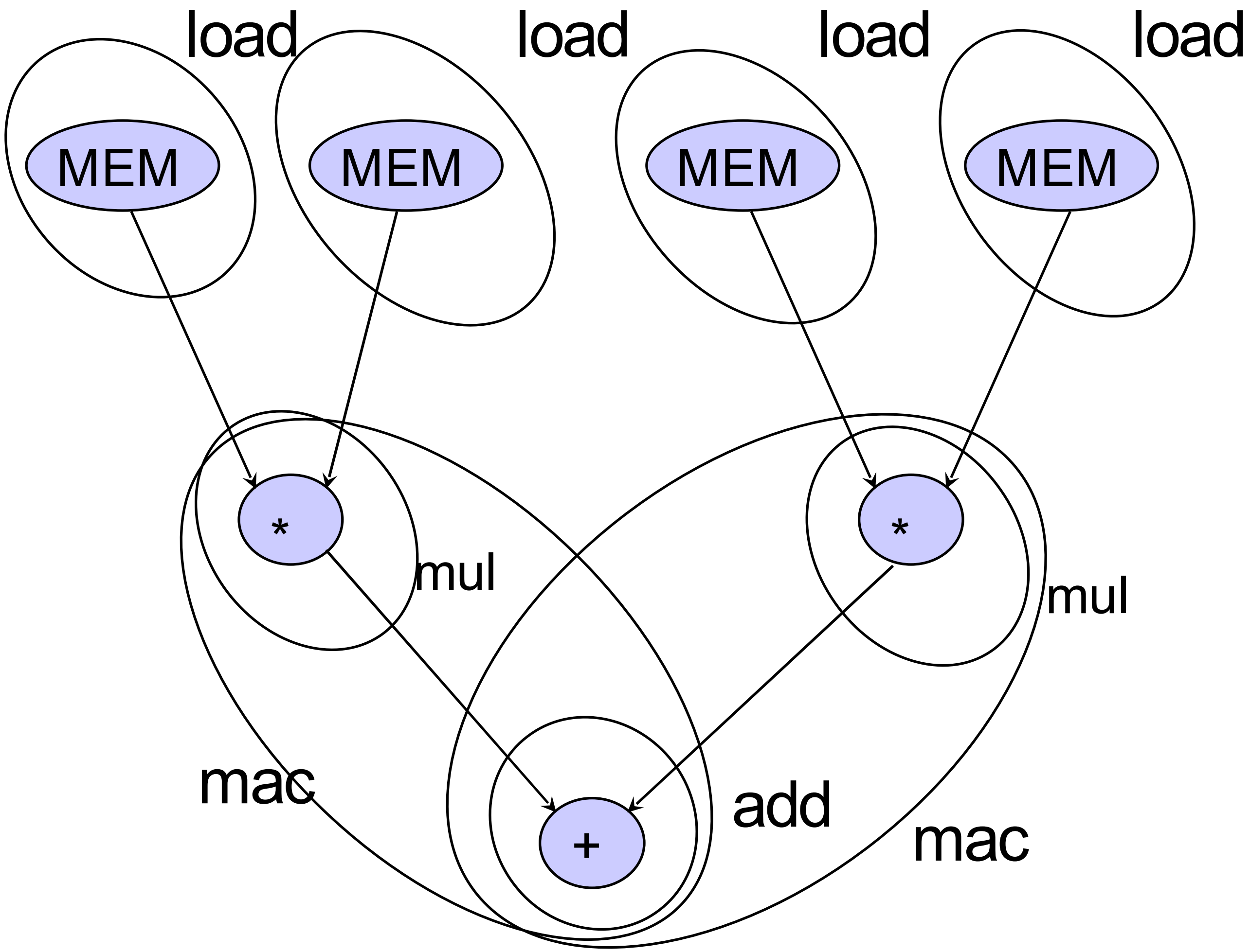
Developer retargetability: compiler specialists responsible for retargeting compilers.

User retargetability: users responsible for retargeting compiler.

Compiler structure



Code selection = covering DFGs



Does not yet consider data moves to input registers.

Code selection by tree parsing (1)

Specification of grammar for generating a iburg parser*:

terminals: {MEM, *, +}

non-terminals: {reg1,reg2,reg3}

start symbol: reg1

rules:

“add” (cost=2): reg1 -> + (reg1, reg2)

“mul” (cost=2): reg1 -> * (reg1,reg2)

“mac” (cost=3): reg1 -> + (*(reg1,reg2), reg3)

“load” (cost=1): reg1 -> MEM

“mov2”(cost=1): reg2 -> reg1

“mov3”(cost=1): reg3 -> reg1

Code selection by tree parsing (2)

- nodes annotated with (register/pattern/cost)-triples -

“load”(cost=1):

reg1 -> MEM

“mov2”(cost=1):

reg2 -> reg1

“mov3”(cost=1):

reg3 -> reg1

“add” (cost=2):

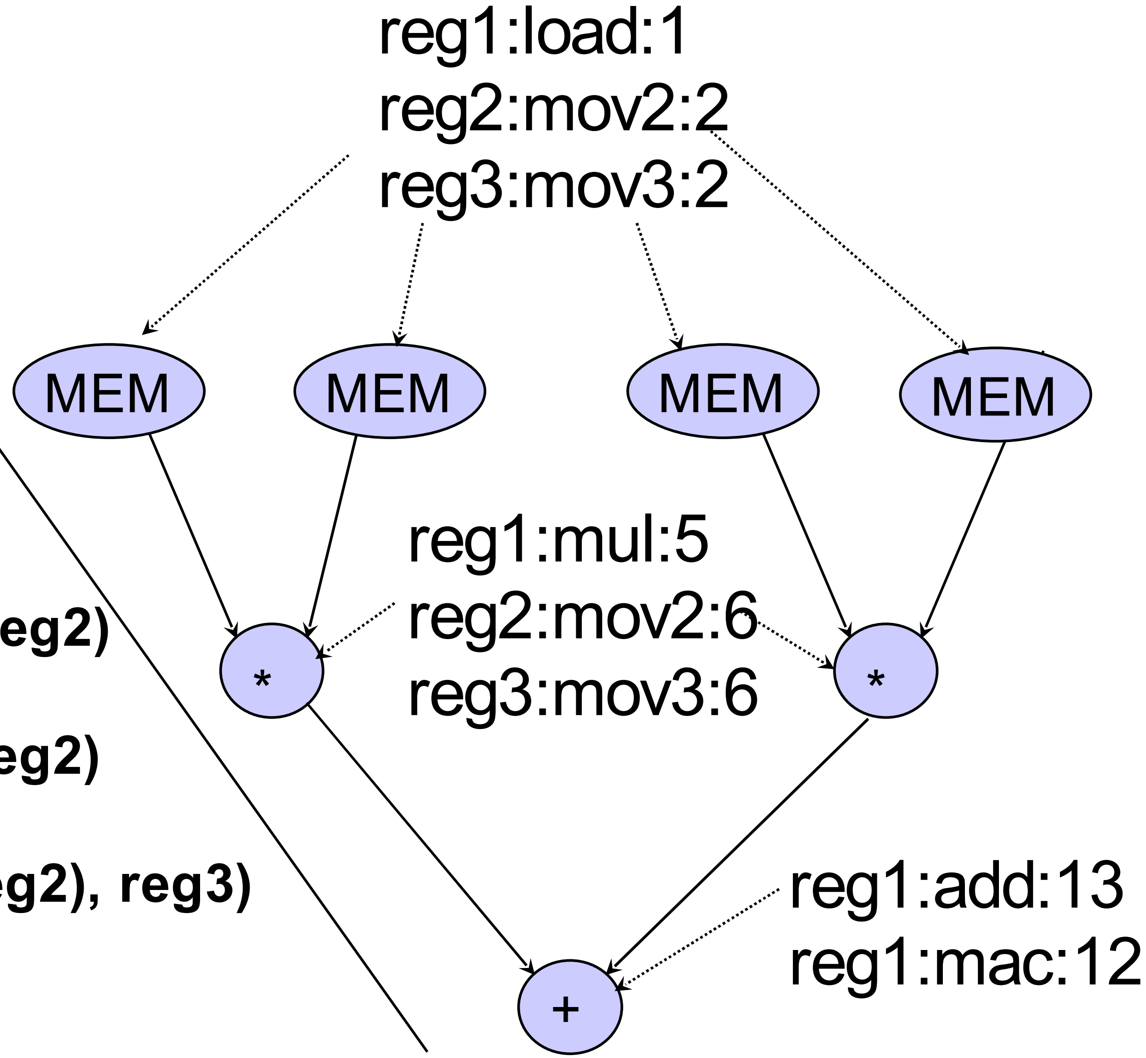
reg1 -> +(reg1, reg2)

“mul” (cost=2):

reg1 -> *(reg1,reg2)

“mac” (cost=3):

reg1->+(*(reg1,reg2), reg3)



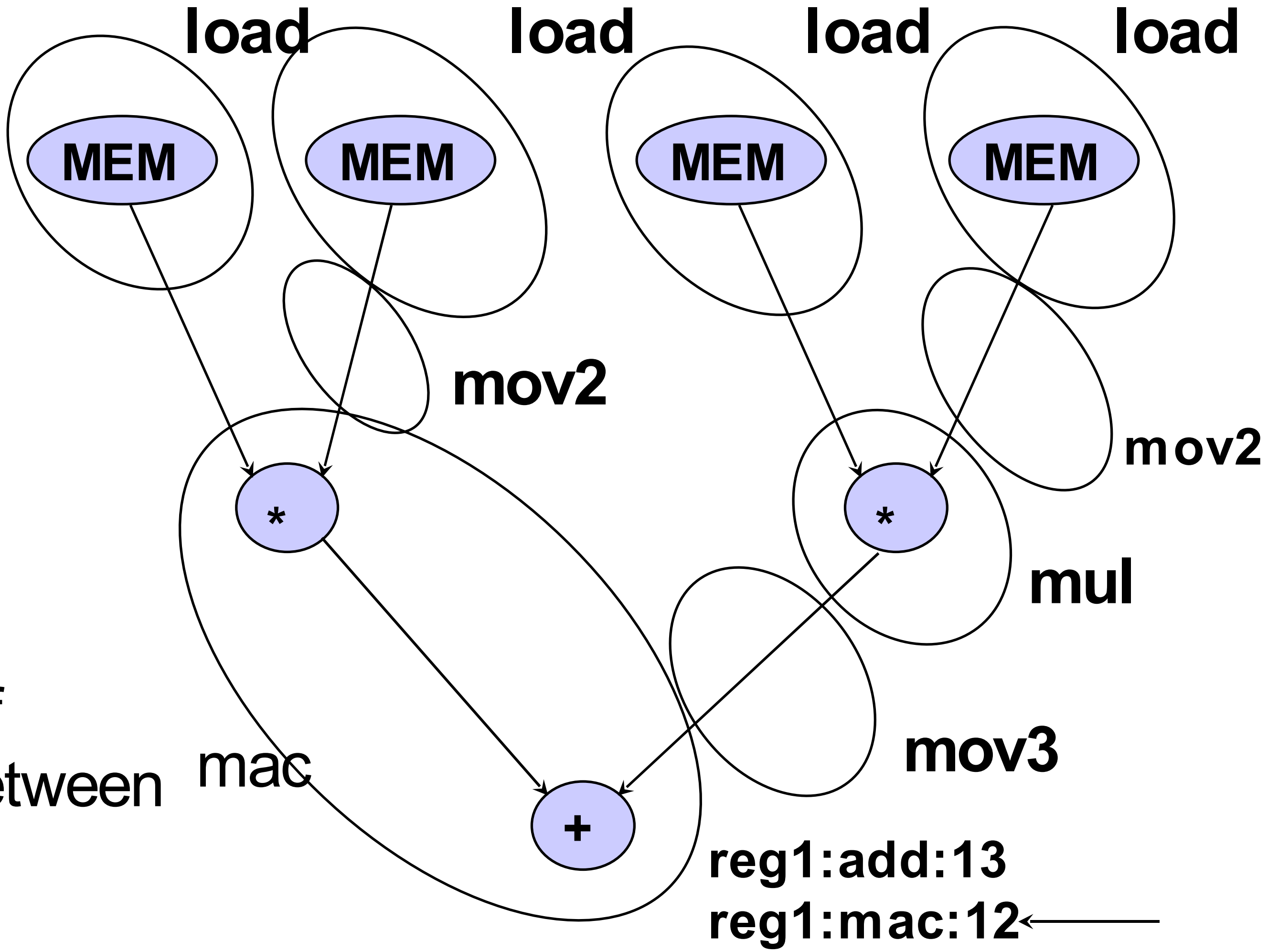
reg1:load:1
reg2:mov2:2
reg3:mov3:2

reg1:mul:5
reg2:mov2:6
reg3:mov3:6

reg1:add:13
reg1:mac:12

Code selection by tree parsing (3)

- final selection of cheapest set of instructions -



Includes routing of values between various registers!

Summary

- The offset assignment problem
- Additional compiler optimizations
  - Using multiple memory banks
  - Exploiting parallelism
  - Multimedia instructions
  - Predicated execution
  - Inlining
- Retargetability