technische universität
dortmund

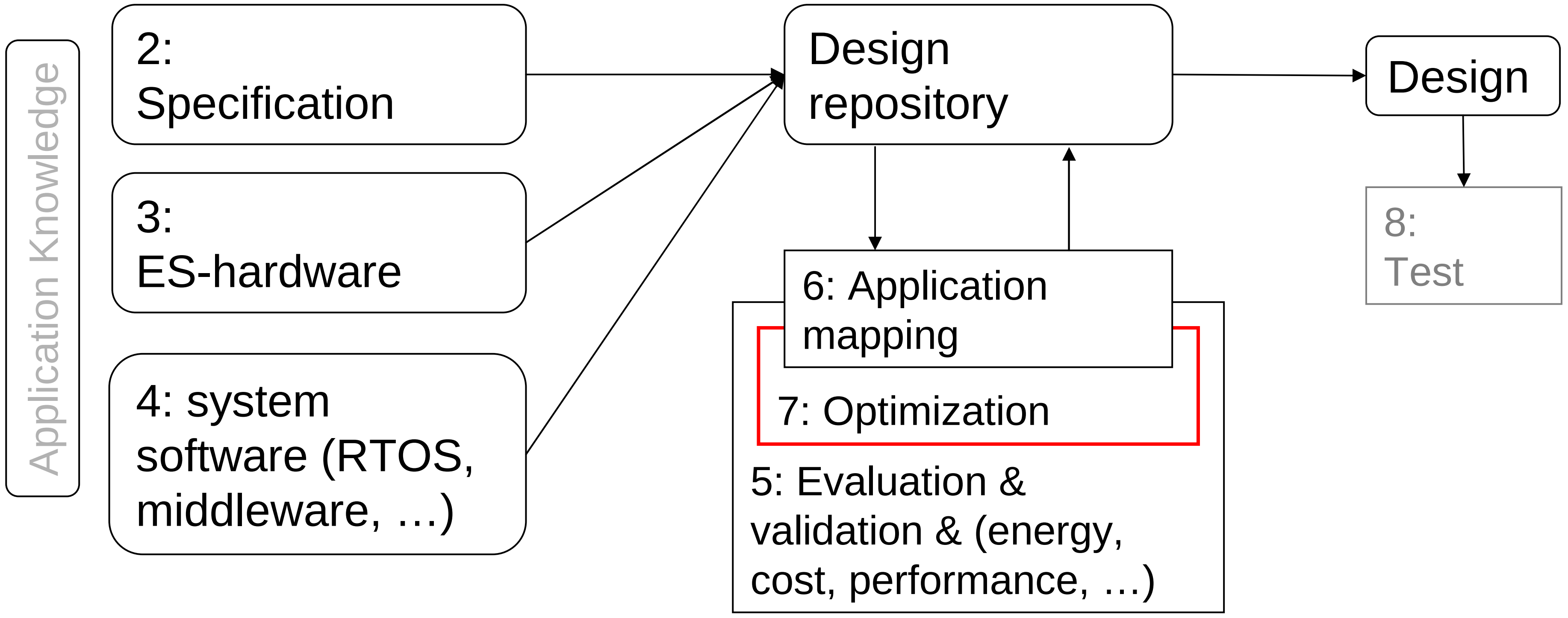
fakultät für informatik
informatik 12

Optimizations
- Compilation for Embedded Processors -

Peter Marwedel
TU Dortmund
Informatik 12
Germany

2011年 01 月 25 日

Structure of this course



Numbers denote sequence of chapters

Reconciling compilers and timing analysis

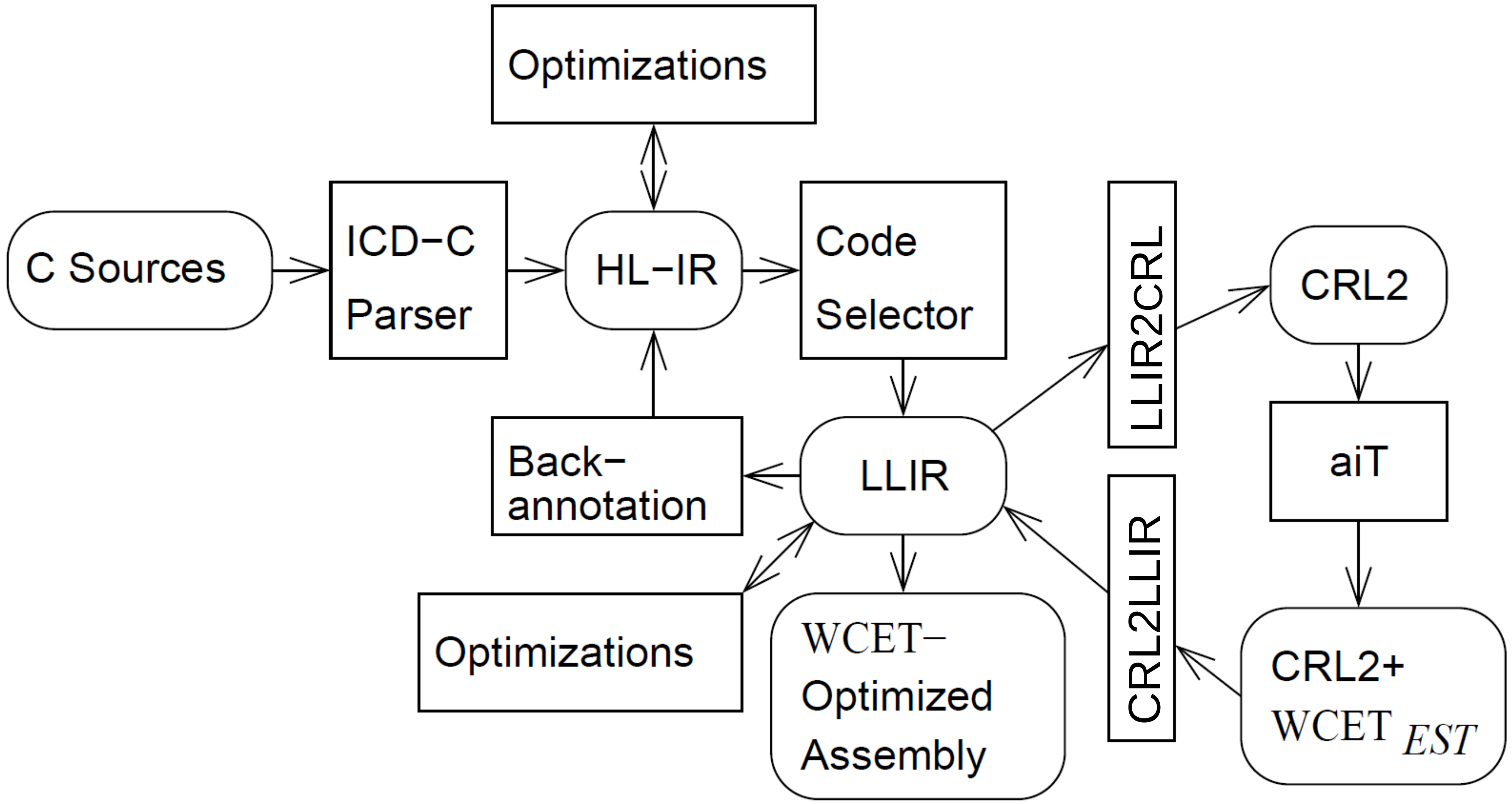
Compilers mostly unaware of execution times

- Execution times are checked in a “trial-and-error” loop: {try: compile – run – check – error: change}*
- Timing analysis based on measurements is not safe

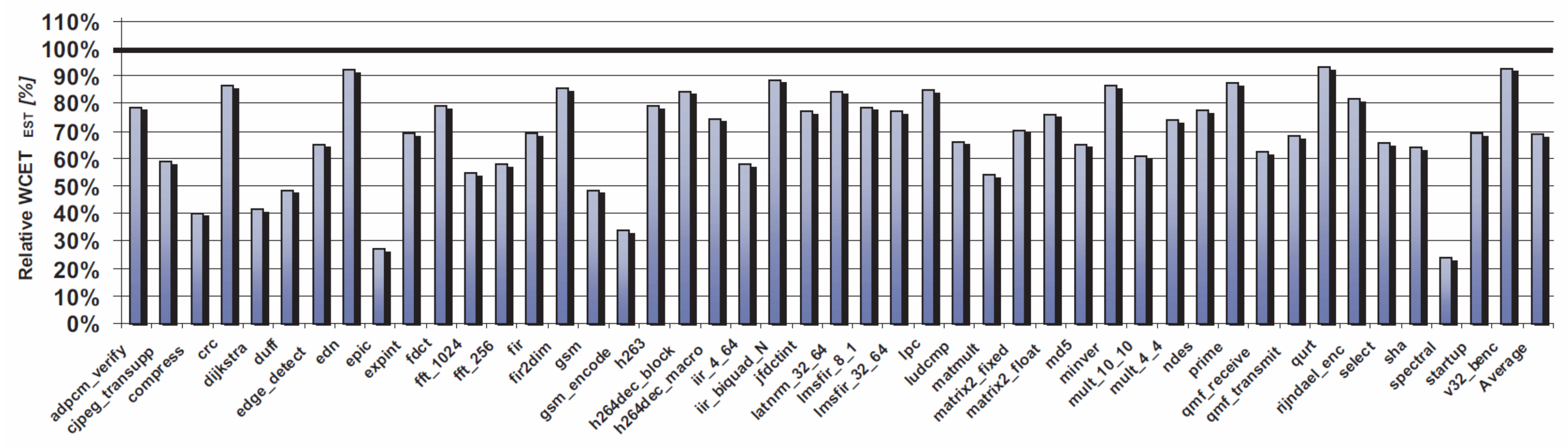
☞ Integration of safe, static timing analysis into compiler

- Getting rid of loops (if everything works well)
- Safe timing verification
- Potential for optimizing for the WCET

Structure of WCC (WCET-aware C-compiler)



Results for WCET-aware register allocation



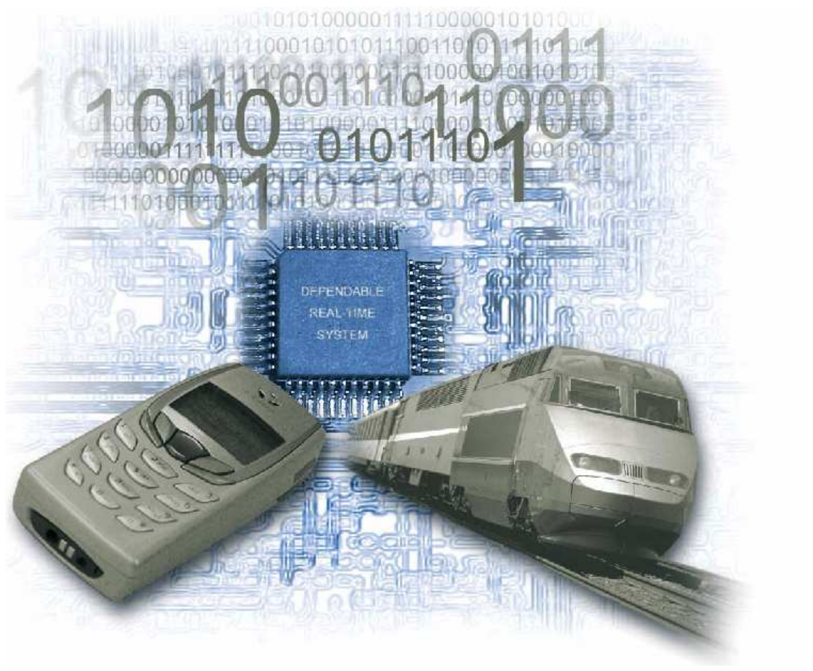
technische universität
dortmund

fakultät für informatik
informatik 12

The offset assignment problem

Peter Marwedel
TU Dortmund
Informatik 12
Germany

2010/01/13

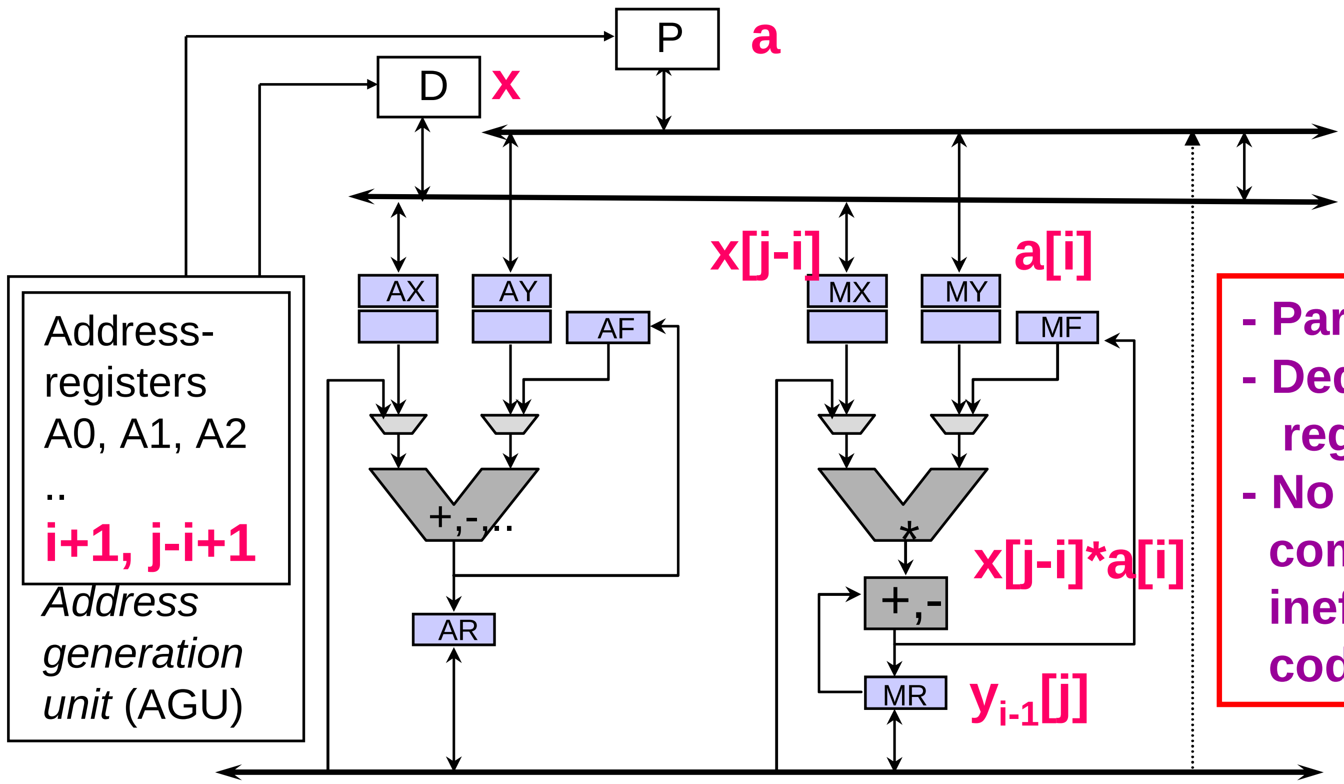


Reason for compiler-problems: Application-oriented Architectures

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

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

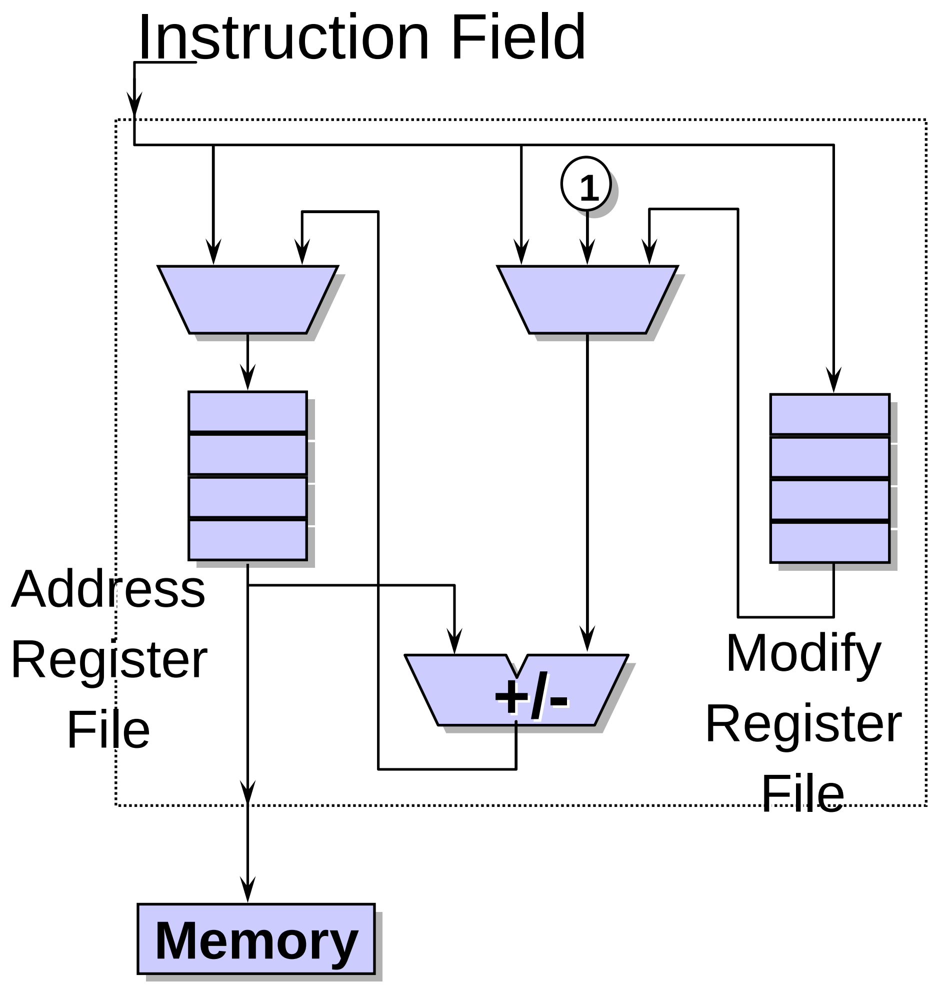
Architecture: Example: Data path ADSP210x



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

Exploitation of parallel address computations

Generic address generation unit (AGU) model



Parameters:

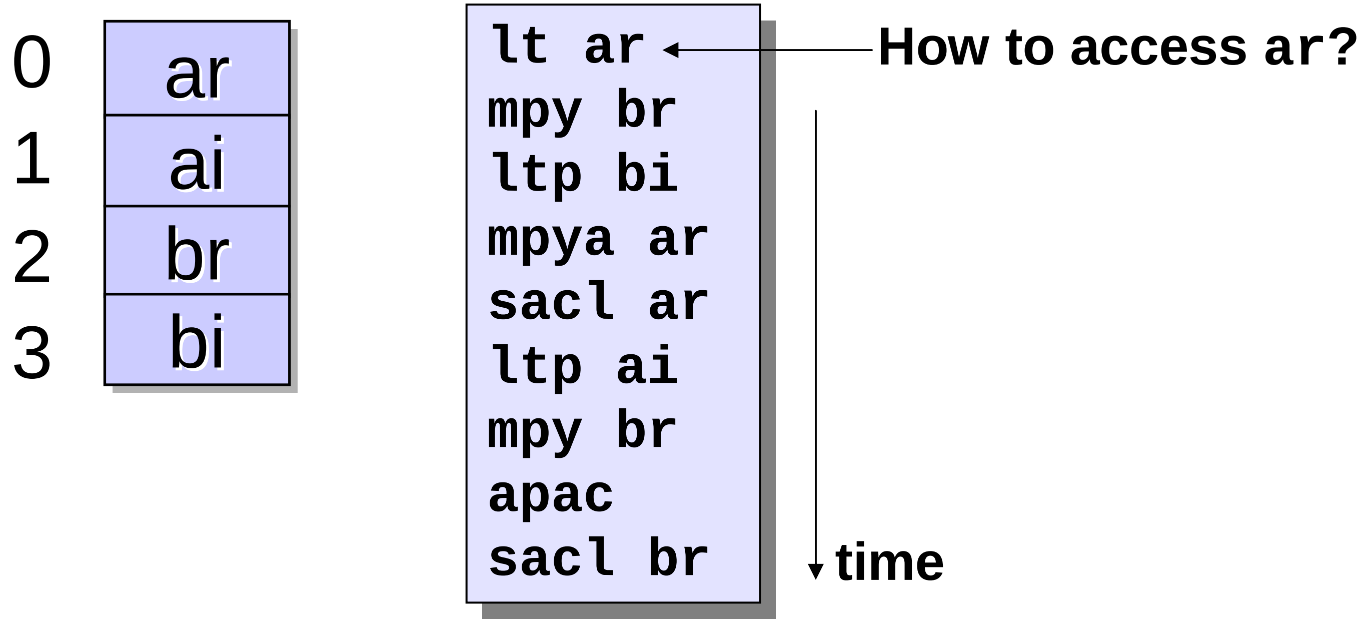
k	=	# address registers
m	=	# modify registers

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)

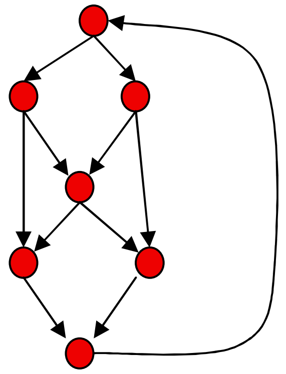
Given: Memory layout + assembly code (without address code)



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

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 :

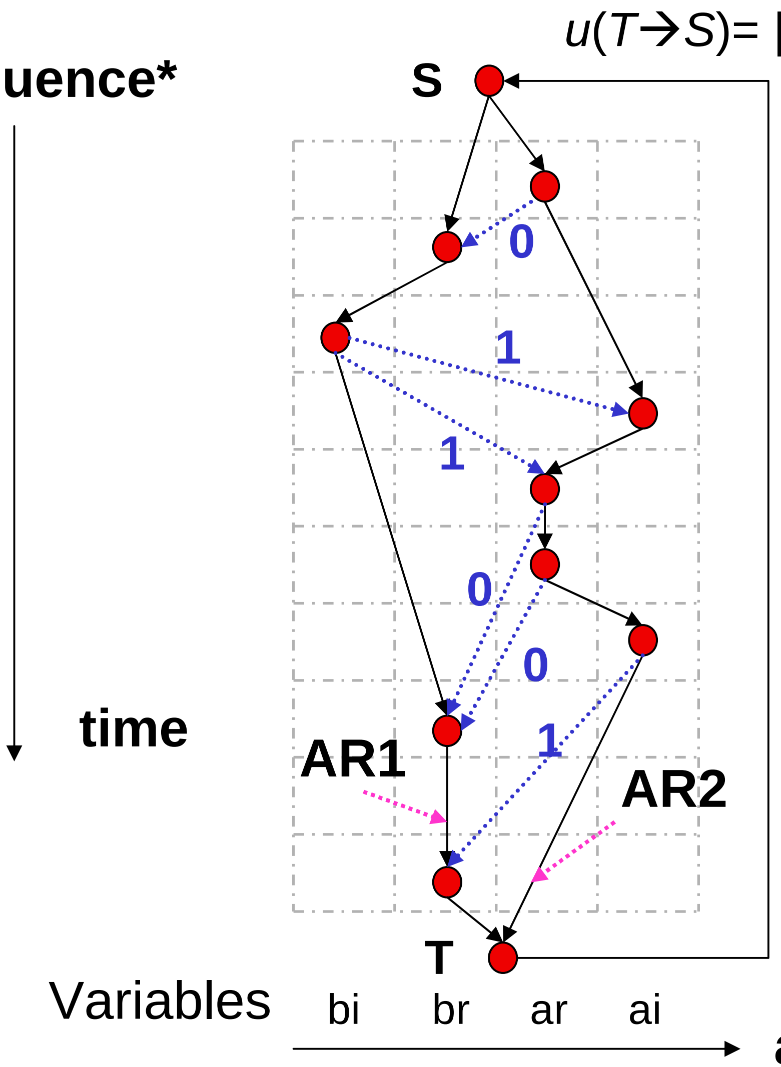
   $\forall\, v \in V: \sum_{w \in V:(v,w) \in E} g(v,w) = \sum_{w \in V:(w,v) \in E} g(w,v)$ (flow conservation)
2. g is **feasible** if $\forall (v,w) \in E$: $g(v,w) \leq u(v,w)$ (capacity constraints)
3. The cost of a circulation g is $c(g) = \sum_{(v,w) \in E} c(v,w)\, g(v,w)$.
4. There may be a lower 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

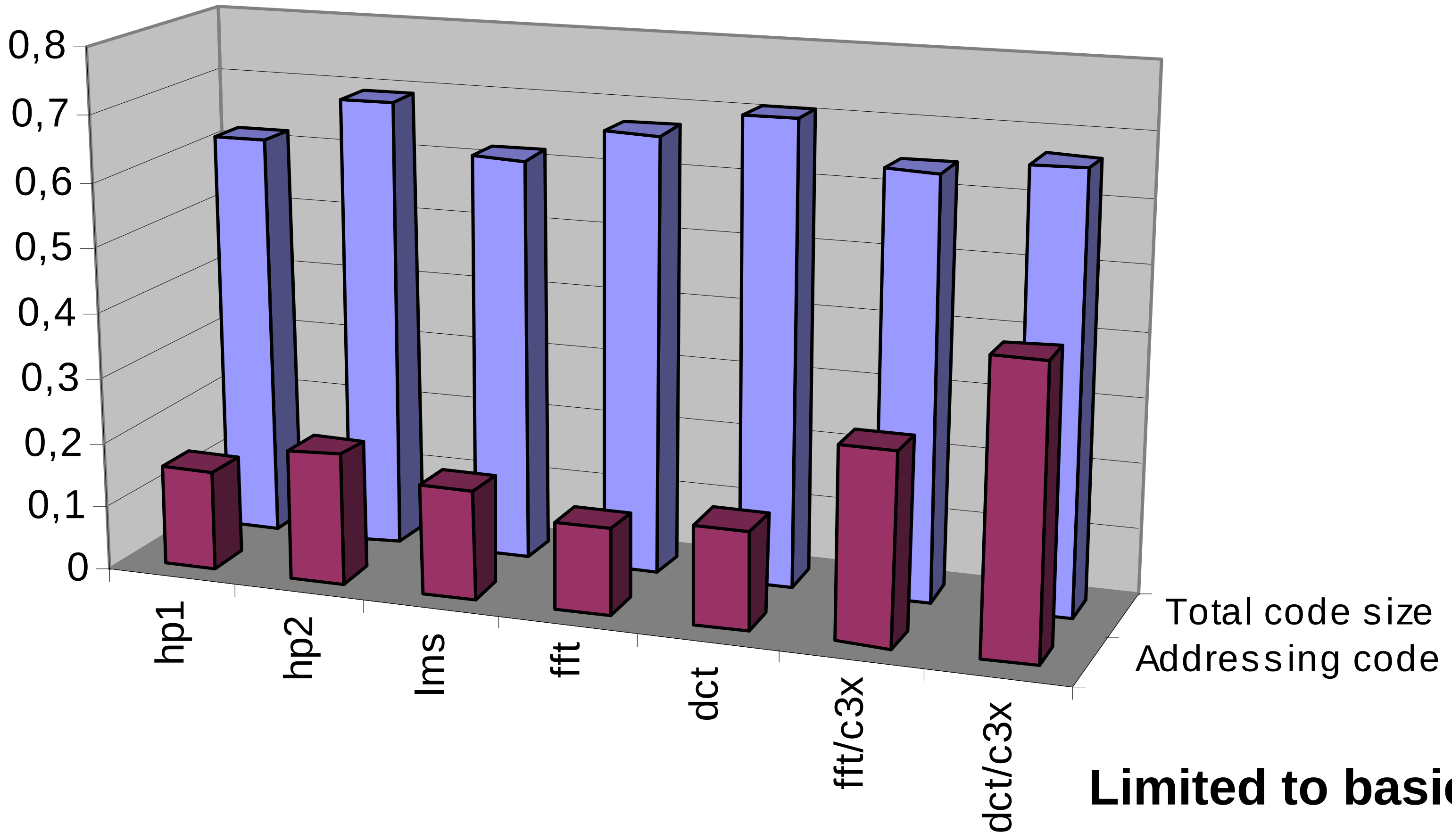
additional edges of original graph (only samples shown)

* C2x pro-cessor 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

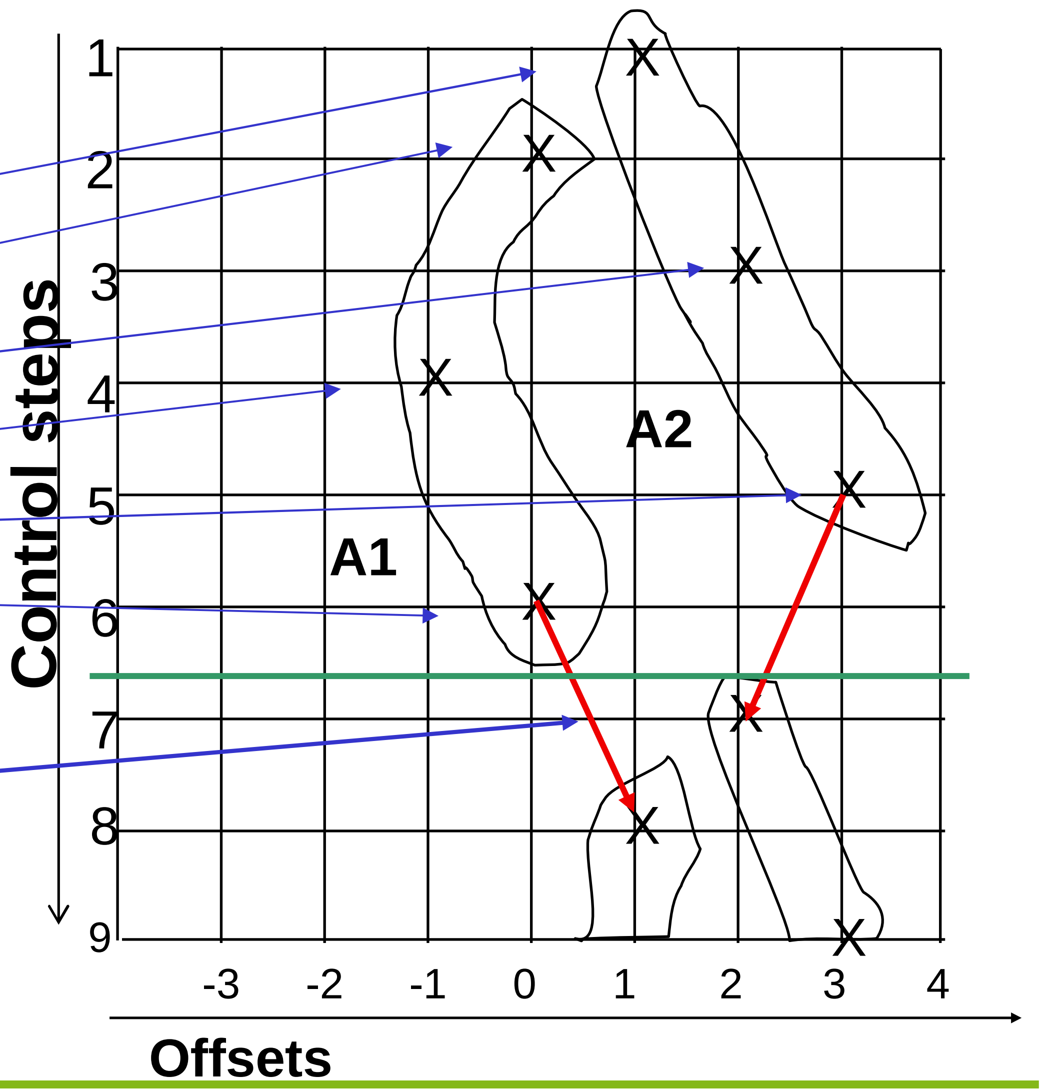
Beyond basic blocks: - handling array references in loops -

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

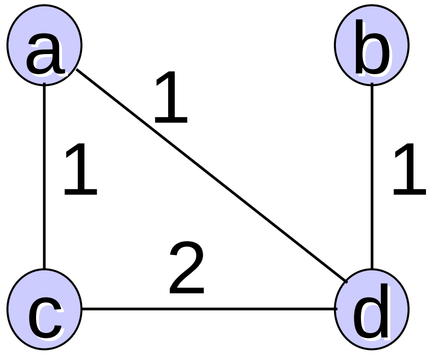
0	b
1	d
2	c
3	a

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

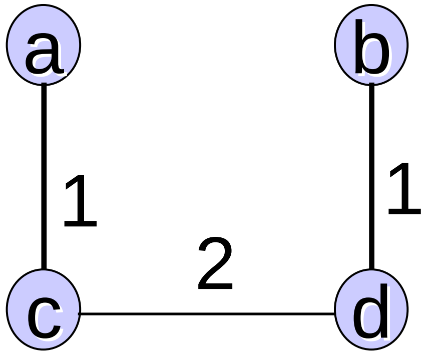
cost: 2

SOA example: Access sequence, access graph and Hamiltonian paths

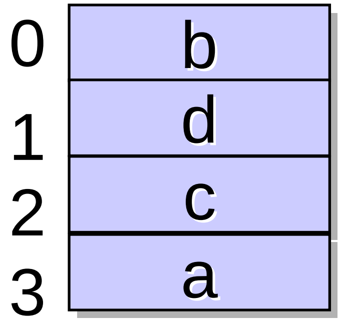
access sequence: b d a c d c



access graph



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



memory layout

SOA used as a building block for more complex situations

➡ significant interest in good SOA algorithms

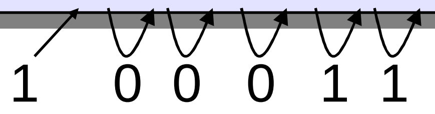
[Bartley, 1992; Liao, 1995]

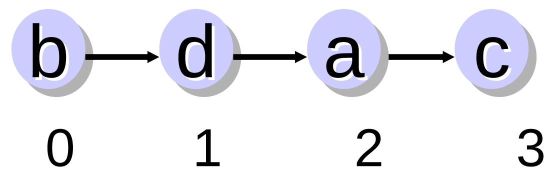
Naïve SOA

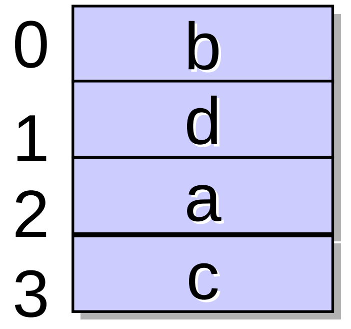
Nodes are added in the order
in which they are used in the program.

Example:

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







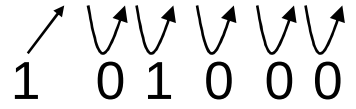
memory layout

Liao's algorithm

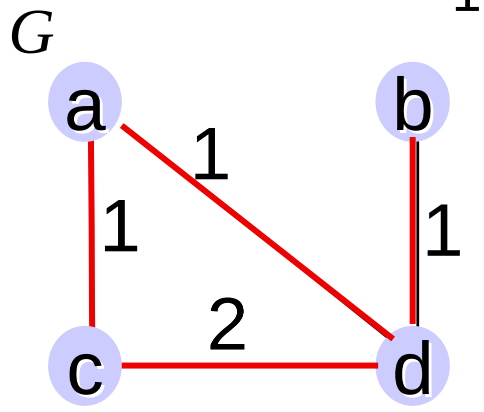
Similar to Kruskal's spanning tree algorithms:

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).

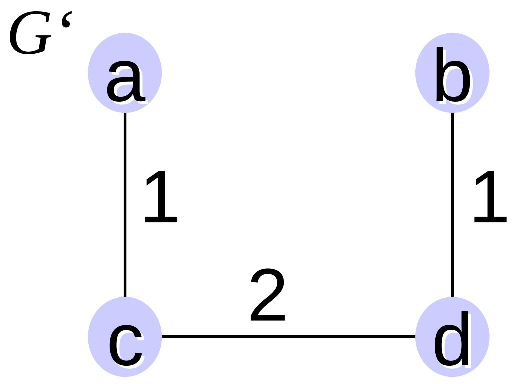
Example: Access sequence: $S=(b, d, a, c, d, c)$



Implicit edges of weight 0 for all unconnected nodes.

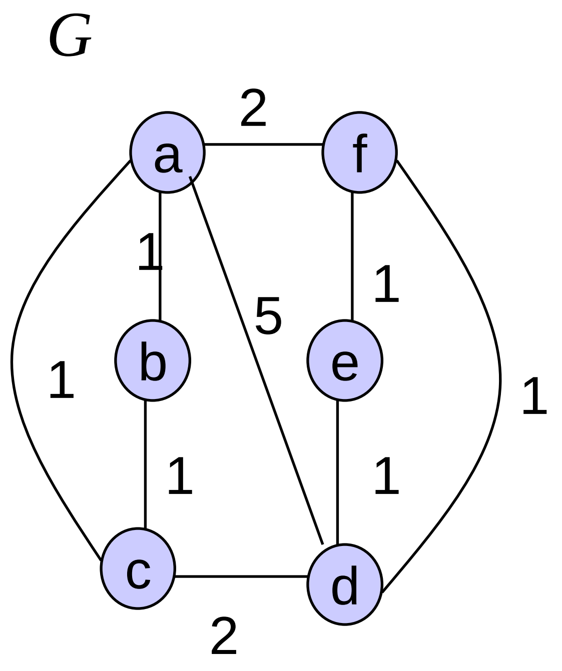


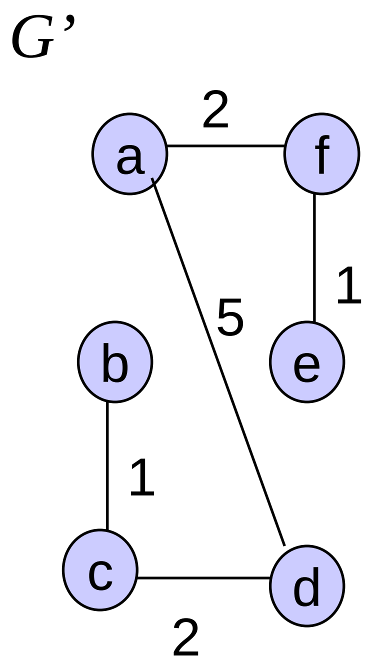
2 (c,d)
1 (a,c)
1 (a,d)
1 (b,d)



Liao's algorithm on a more complex graph

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





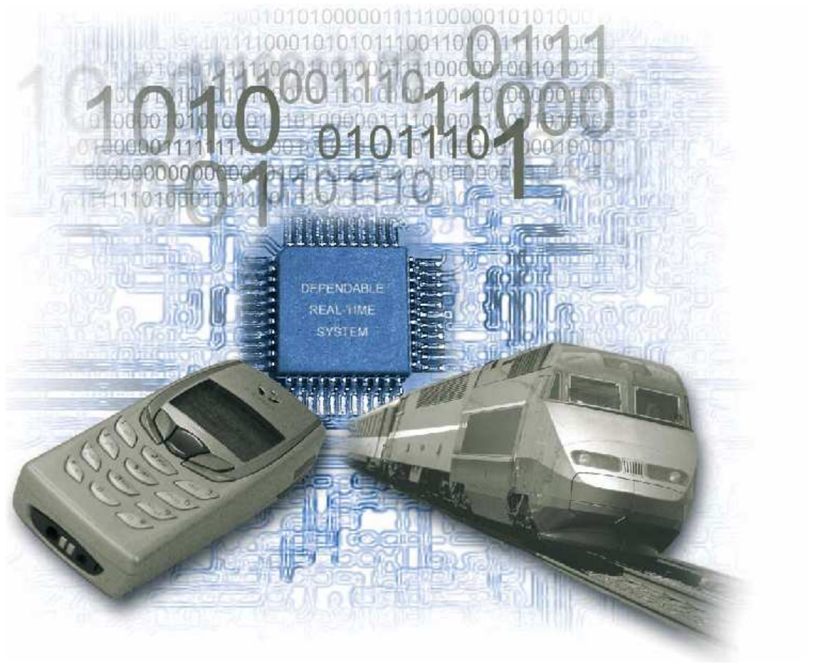
technische universität
dortmund

fakultät für informatik
informatik 12

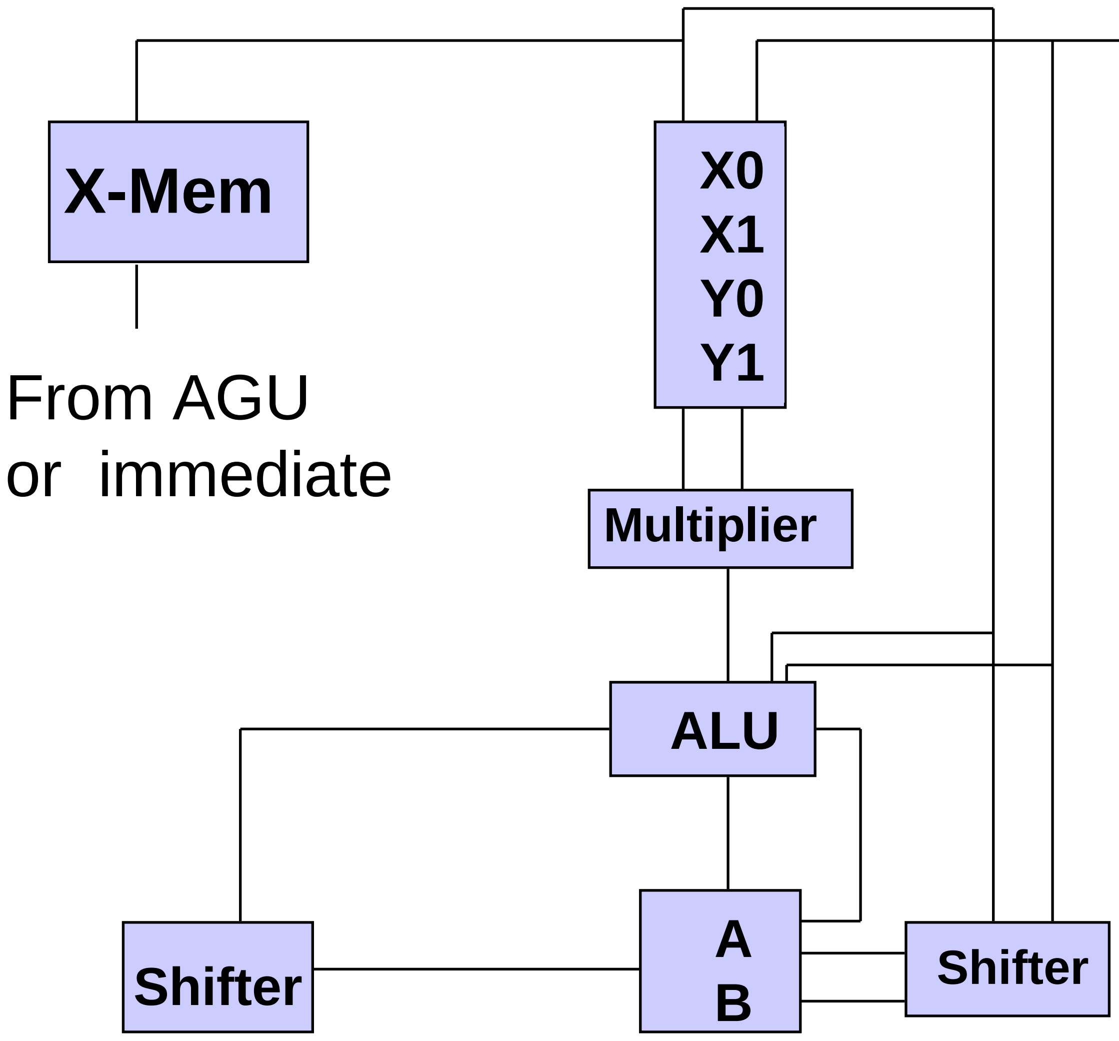
Further optimizations

Peter Marwedel
TU Dortmund
Informatik 12
Germany

2010/01/13



Multiple memory banks - Sample hardware -



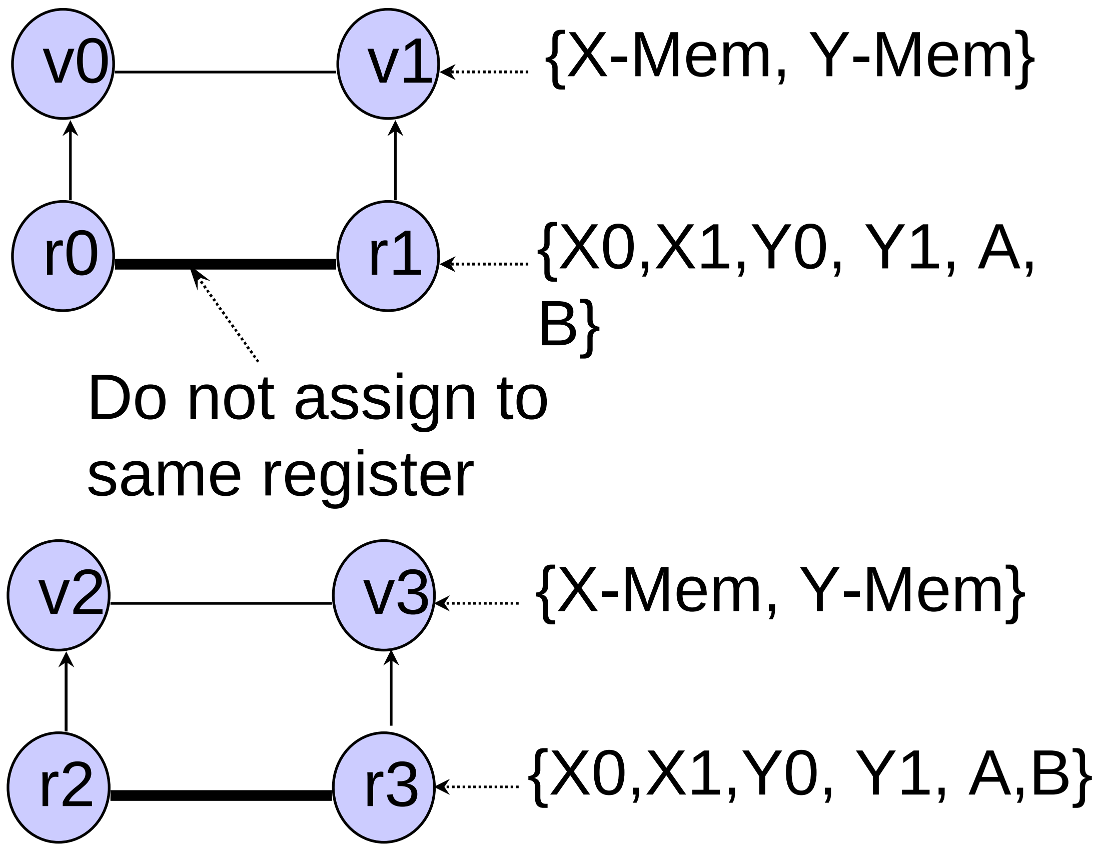
Parallel moves possible if using different memory banks

Multiple memory banks
- Constraint graph generation -

Precompacted code (symbolic variables and registers)

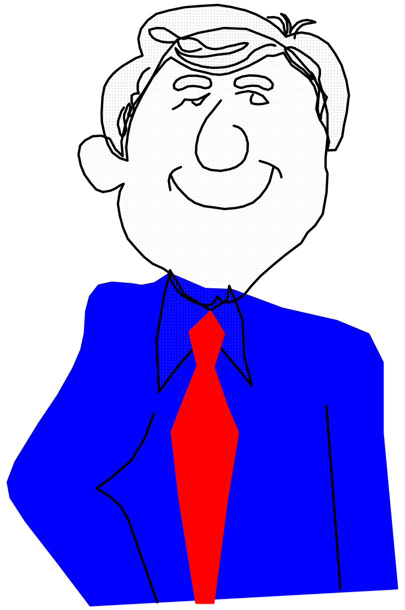
Move v0,r0 v1,r1
Move v2,r2 v3,r3

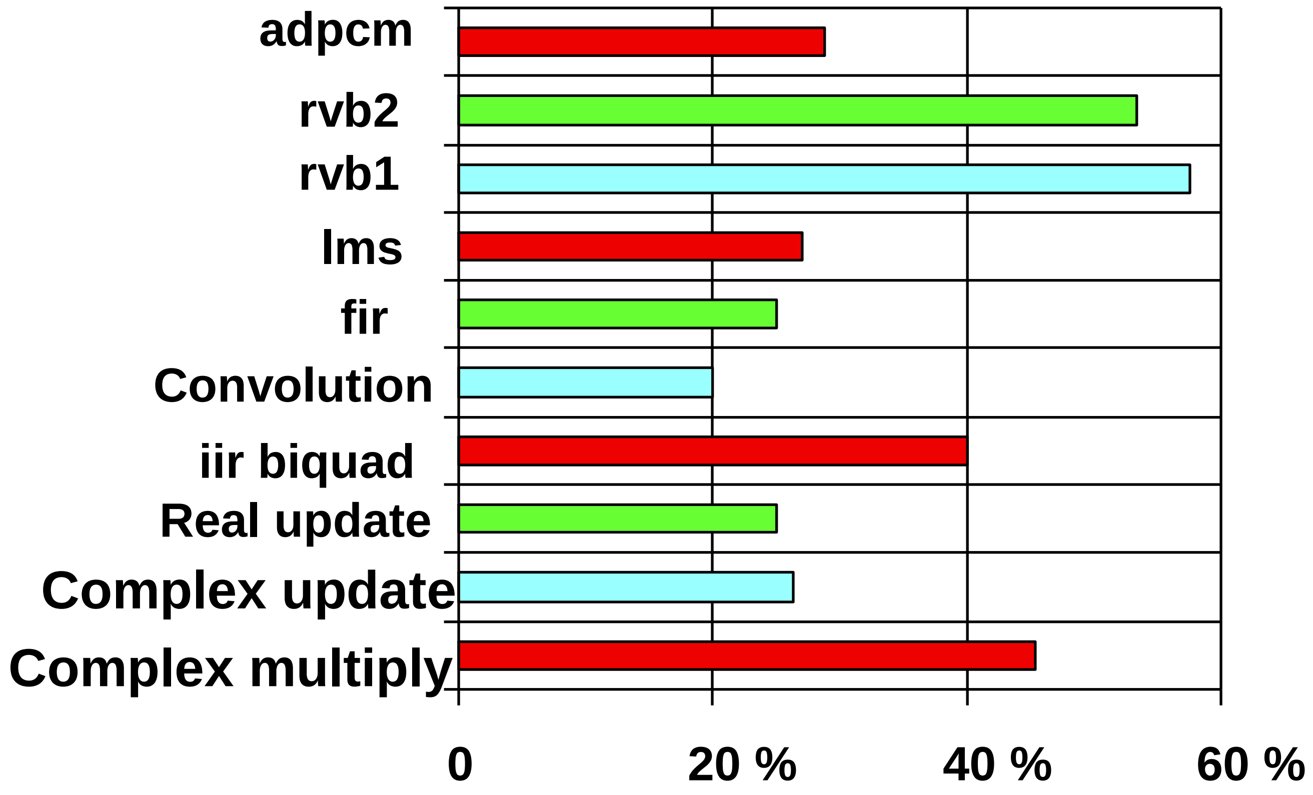
Constraint graph



Links maintained, more constraints ...

Multiple memory banks
Code size reduction through simulated annealing

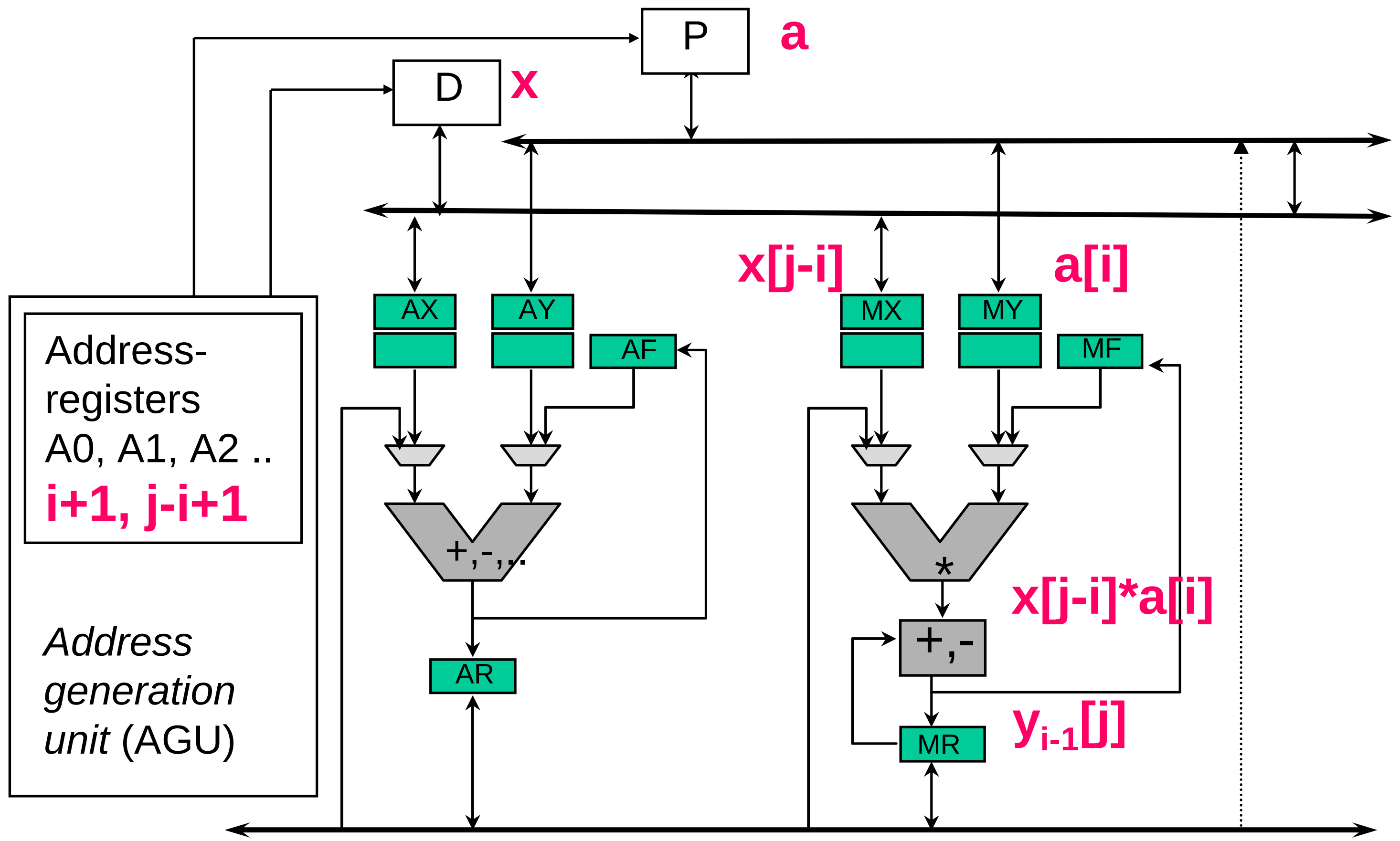




[Sudarsanam, Malik, 1995]

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)

$u(n) = u(n - 1) + K0 \times e(n) + K1 \times e(n - 1);$
$e(n - 1) = e(n)$

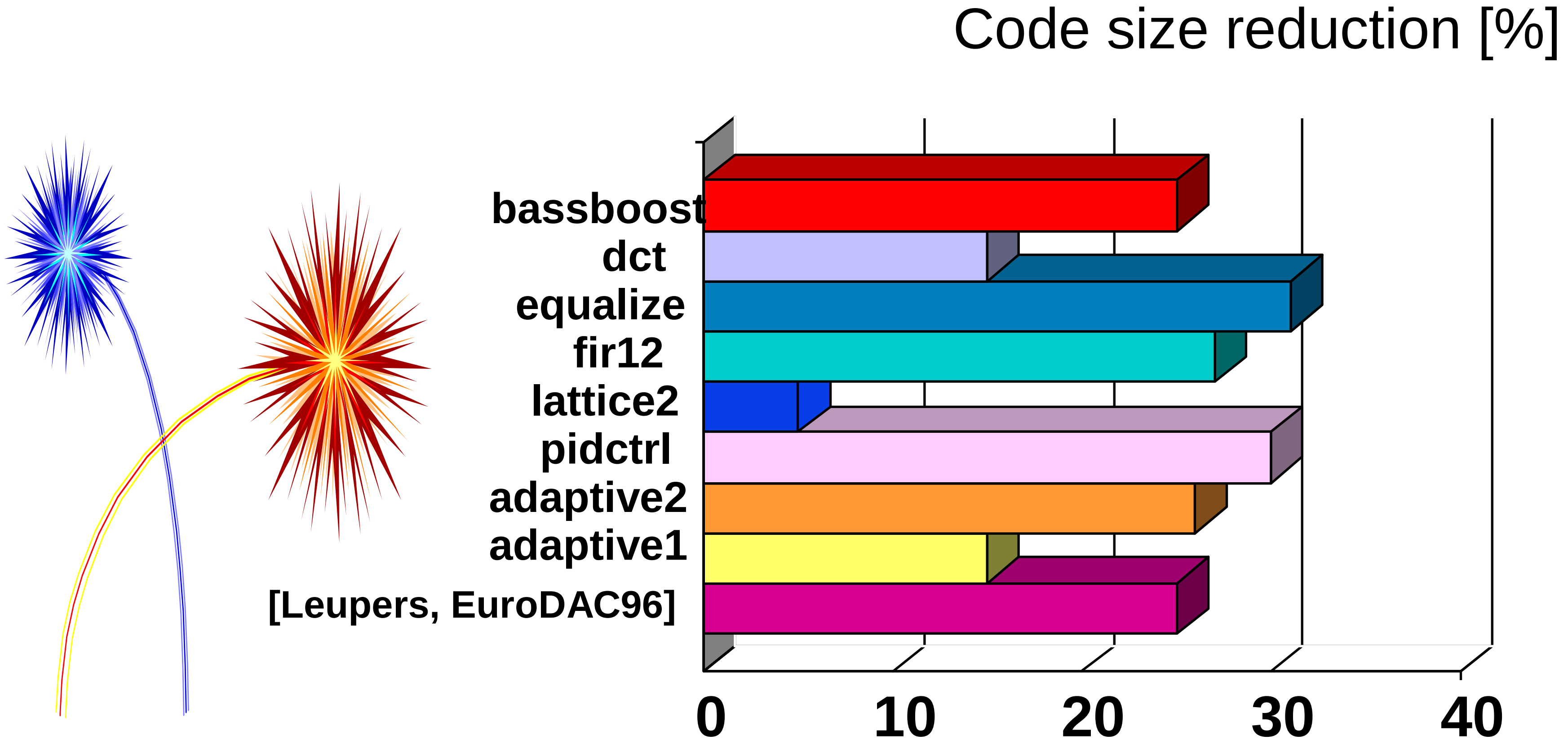
```
ACCU   := u(n - 1)
TR     := e(n - 1)
PR     := TR × K1
TR     := e(n)
e(n - 1) := e(n)
ACCU   := ACCU + PR
PR     := TR × K0
ACCU   := ACCU + PR
u(n)   := ACCU
```

```
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
```

- From 9 to 7 cycles through compaction
-

Exploitation of instruction level parallelism (ILP)

Results obtained through integer programming:



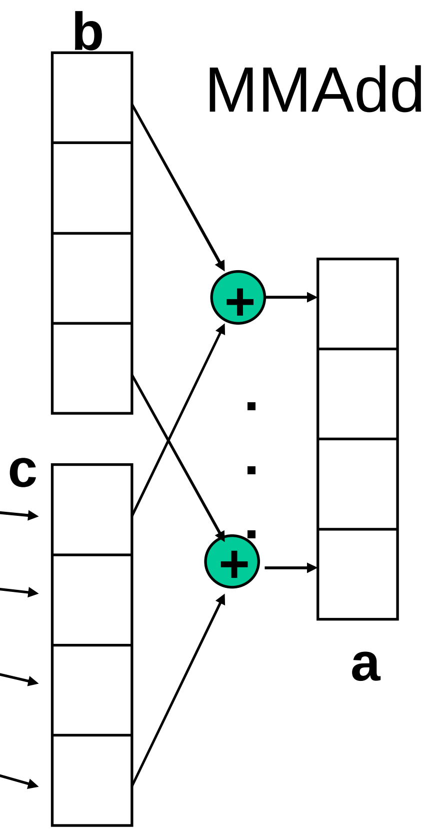
[Leupers, EuroDAC96]

Compaction times: 2 .. 35 sec

Exploitation of Multimedia Instructions

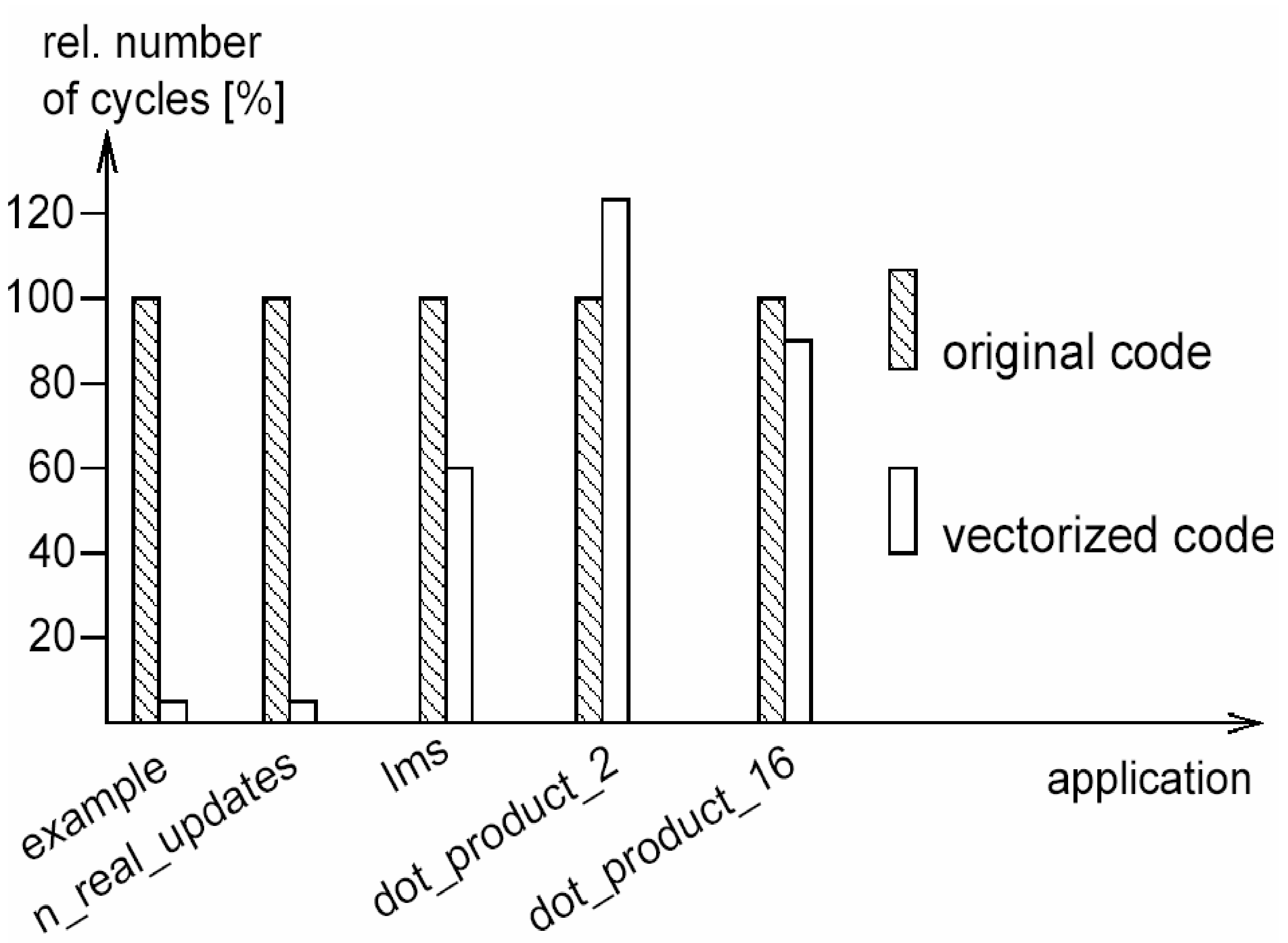
```
FOR i:=0 TO n DO
 a[i] = b[i] + c[i]
```

```
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)

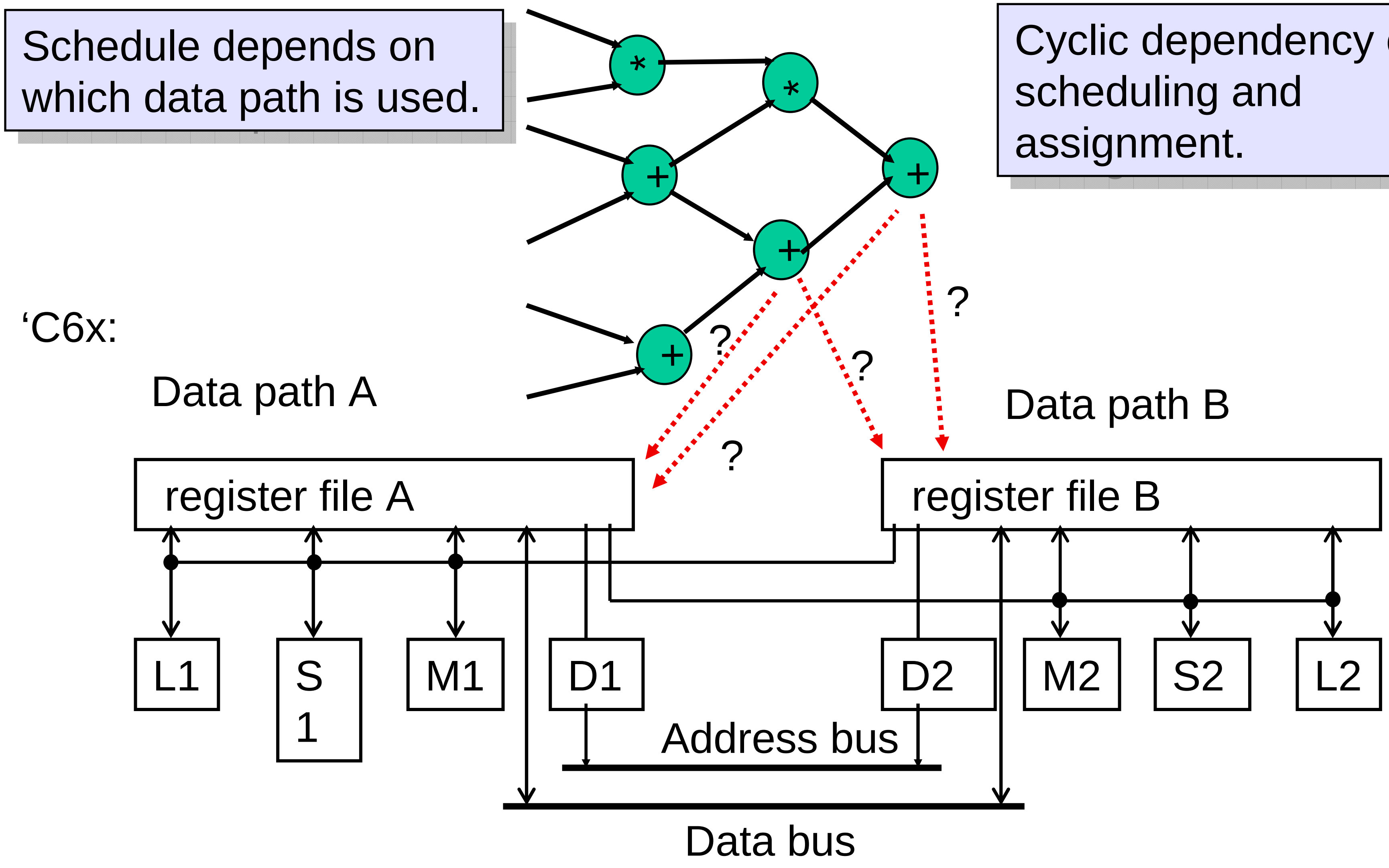
Improvements for M3 DSP due to vectorization



Scheduling for partitioned data paths

Schedule depends on which data path is used.

Cyclic dependency of scheduling and assignment.

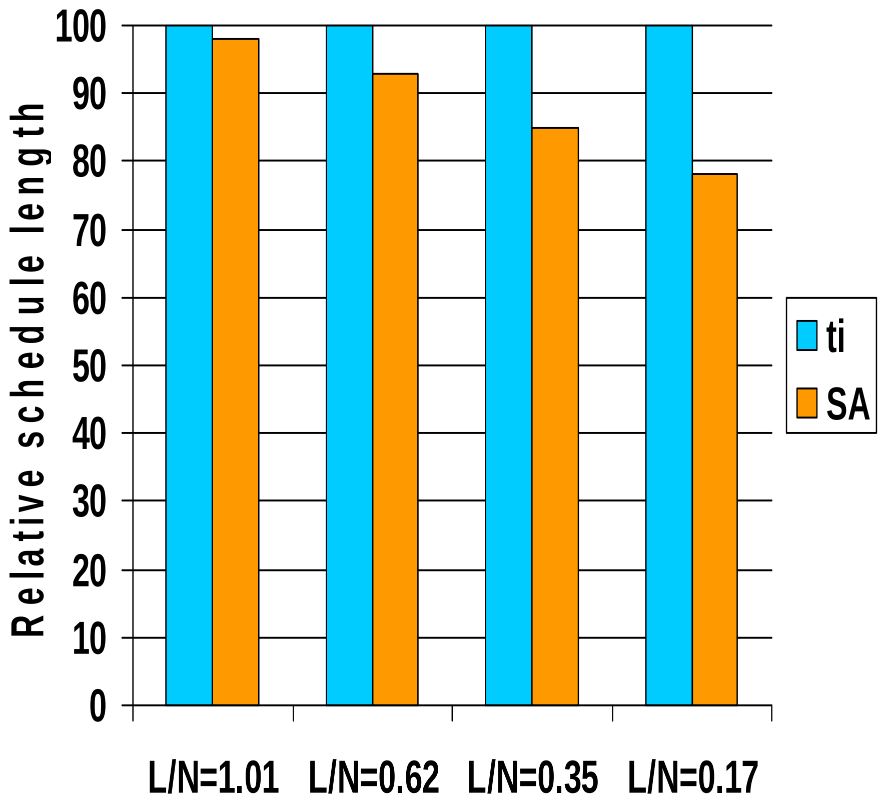


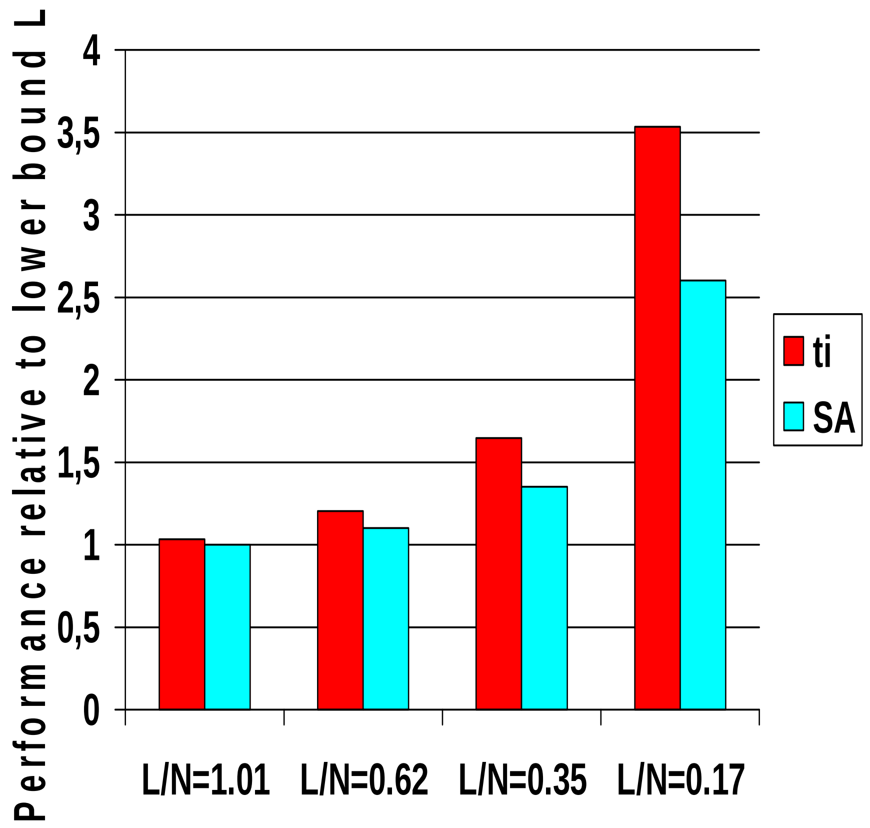
Integrated scheduling and assignment using Simulated Annealing (SA)

```
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



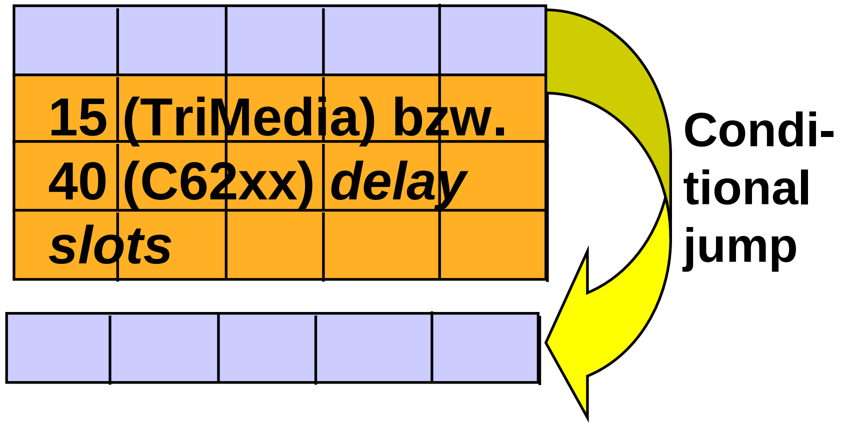


SA approach outperforms the ti approach for "wide" DFGs (containing a lot of parallelism)

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)

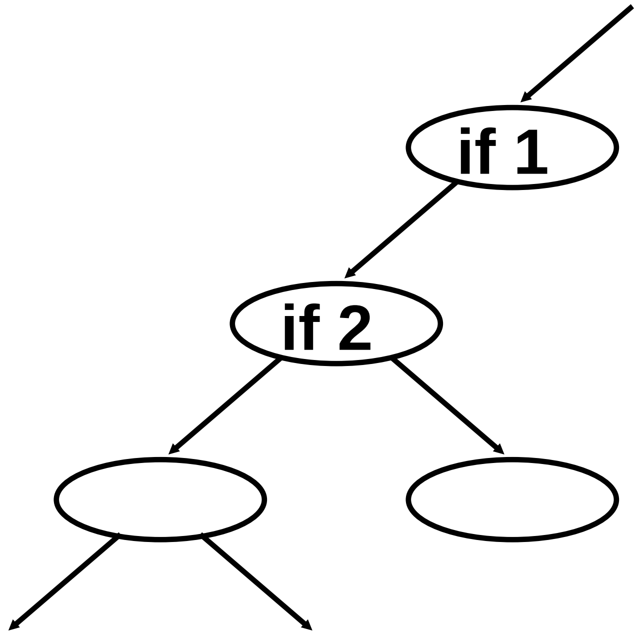
3. Conditional jump :
```
[c1] c:=c2
[~c1] c:=0
BNE c, L;
t2;
L: ...
```

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

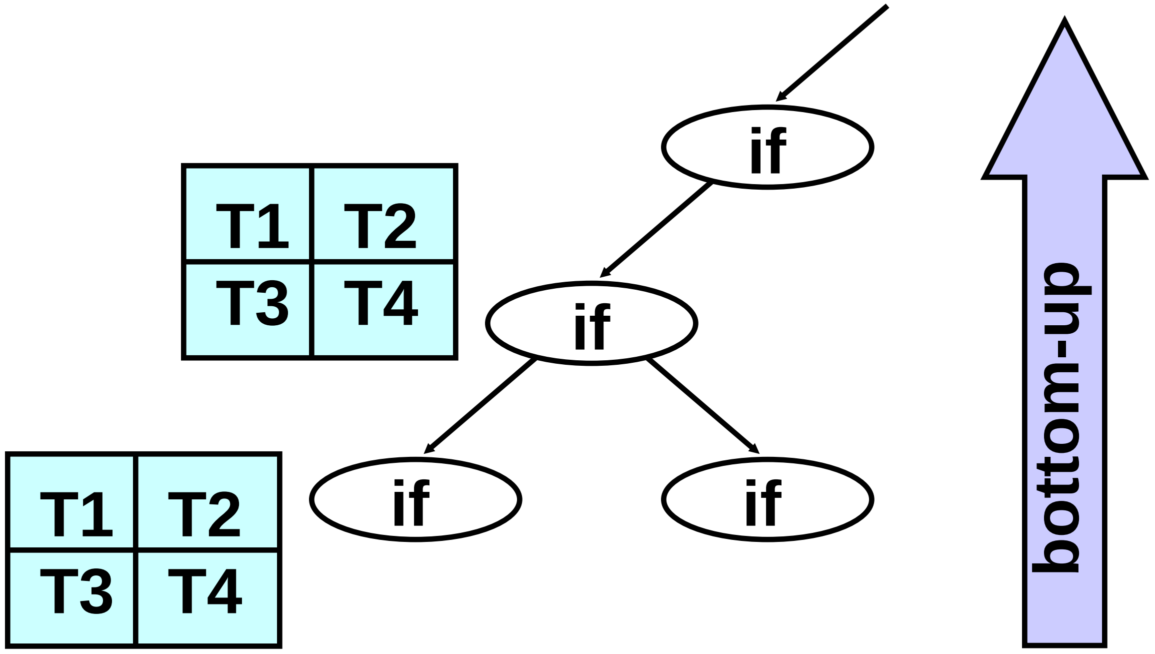


- Selection of fastest implementation for if-1 requires knowledge of how fast if-2 can be implemented.
- Execution time of if-2 depends on setup code, and, hence, also on how if 1 is implemented
- cyclic dependency!

Dynamic programming algorithm (phase 1)

For each if-statement compute 4 cost values:

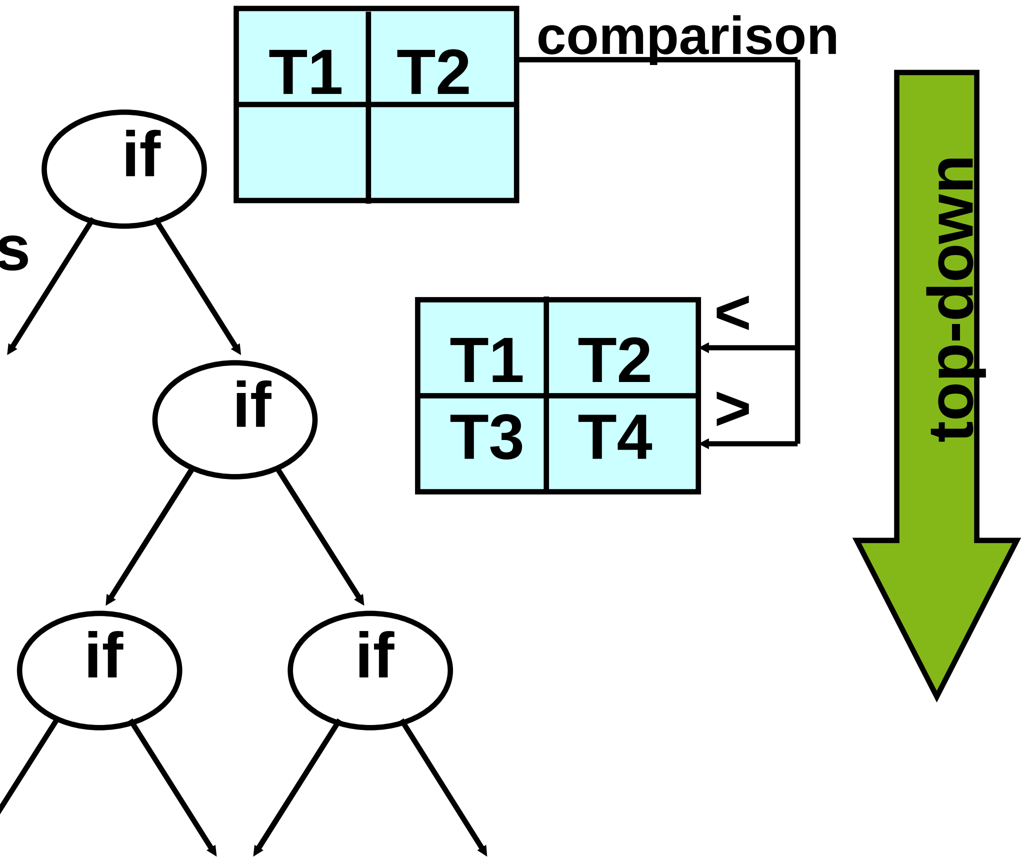
T1 : cond. jump, no precondition
T2 : cond. instructions, no precondition
T3 : cond. jump, with precondition
T4: cond. instructions, with precondition



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

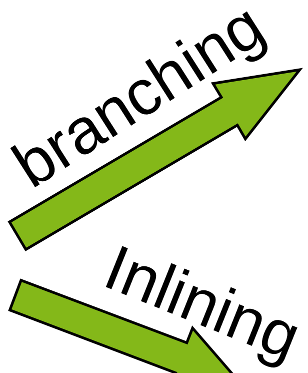
Example	Conditional jumps	Conditional instructions	Dynamic program.	Min (col. 2-5)	TI C compiler
1	21	11	11	11	15
2	12	13	13	12	13
3	26	21	22	21	27
4	9	12	12	9	10
5	26	30	24	24	21
6	32	23	23	23	30
7	57	173	49	49	51
8	39	244	30	30	41
9	28	27	27	27	29
10	27	30	30	27	28

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
```

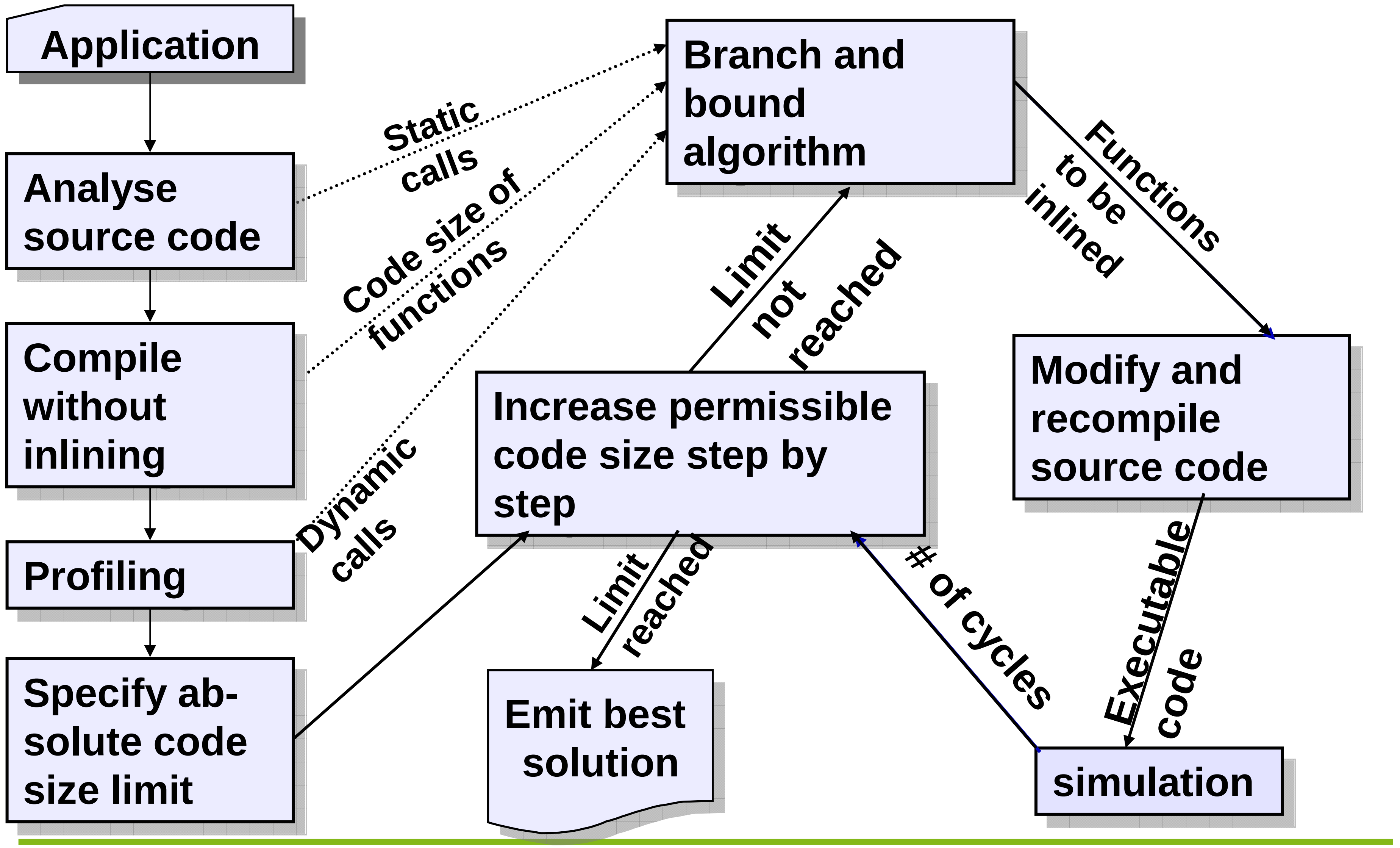
Limitations:

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

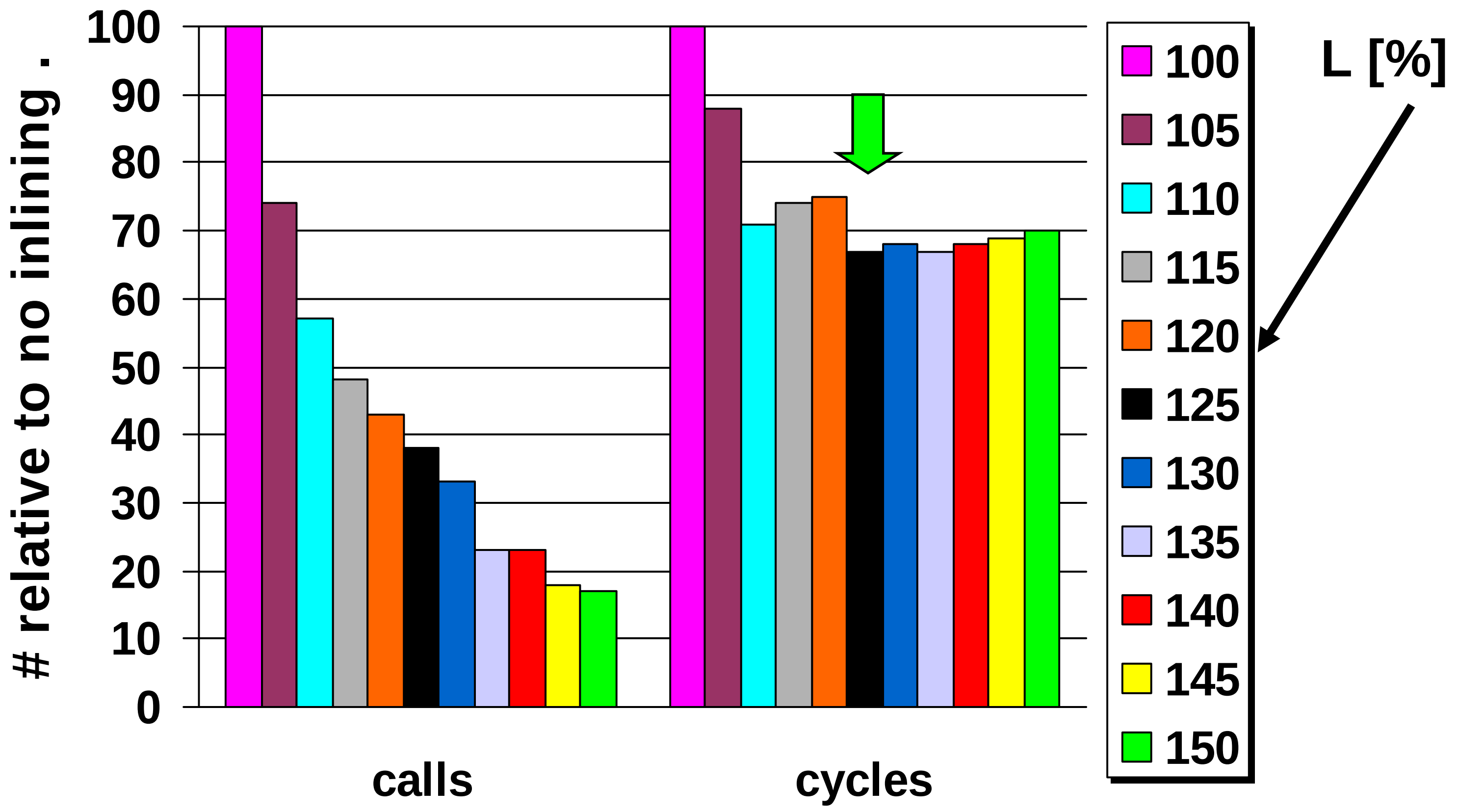
Goal:

- 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
Executable 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.

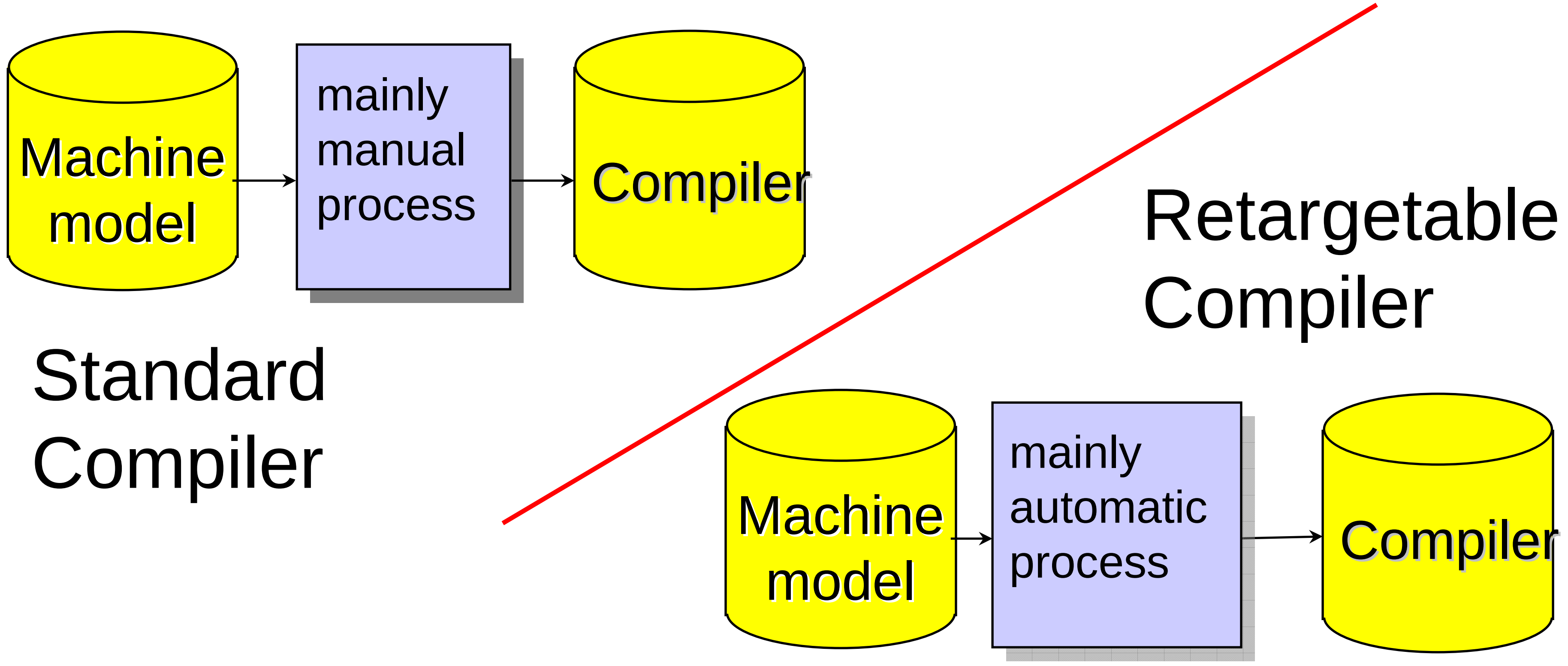
References:

- 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

Summary

- Optimizations for Caches
  - Code Layout transformations
  - Way prediction
- The Offset assignment problem
  - Address pointer assignment problem
  - Simple offset assignment problem
  - General offset assignment problem
- Further optimizations
  - Compaction
  - Multimedia- and VLIW architecture support
  - Predicated execution support
  - Space-aware inlining

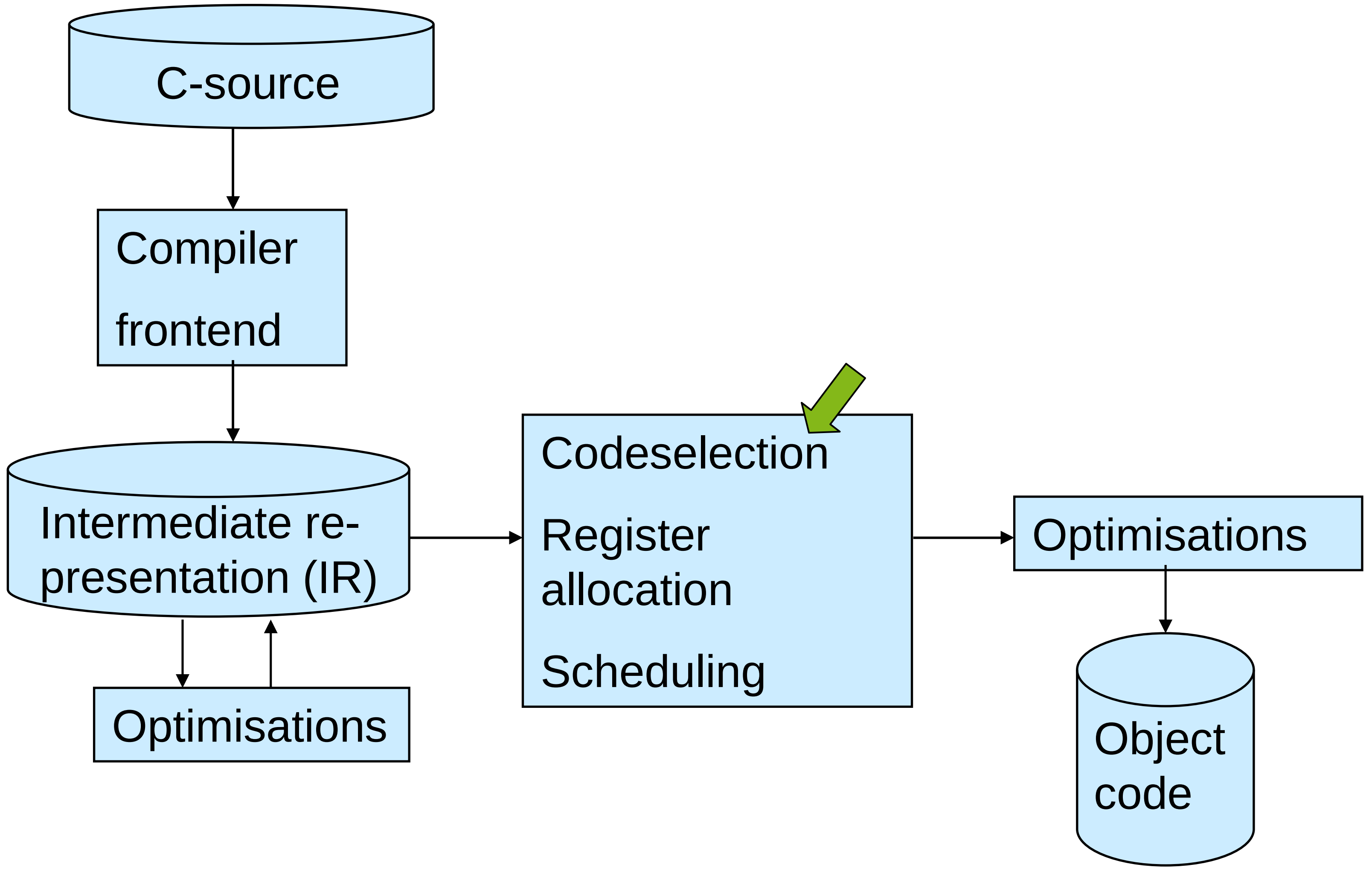
Retargetable Compilers vs. Standard Compilers



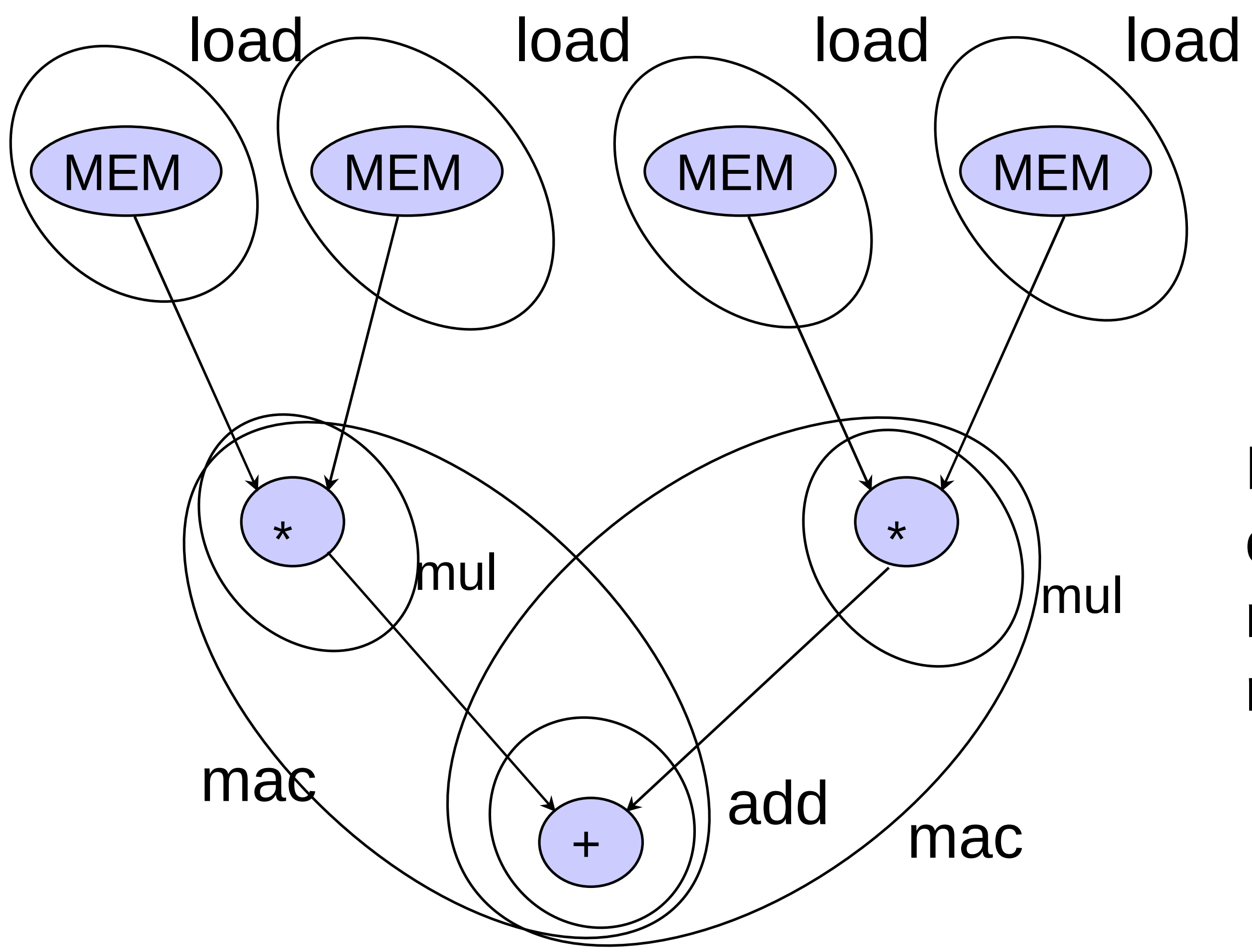
Developer retargetability: compiler specialists responsible for retargeting compilers.

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

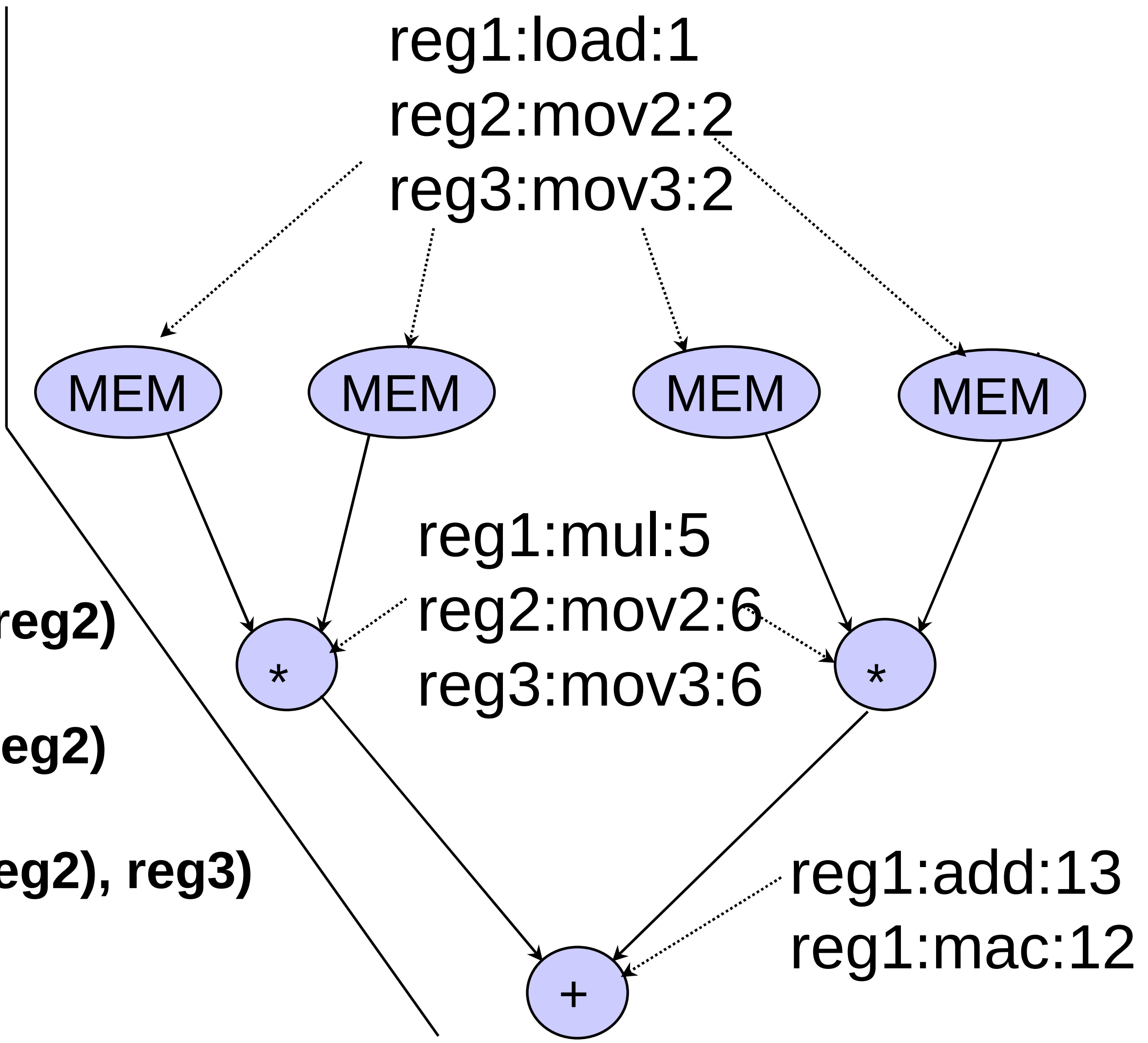
“mov2”(cost=1): reg2 -> reg1

“mov3”(cost=1): reg3 -> reg1

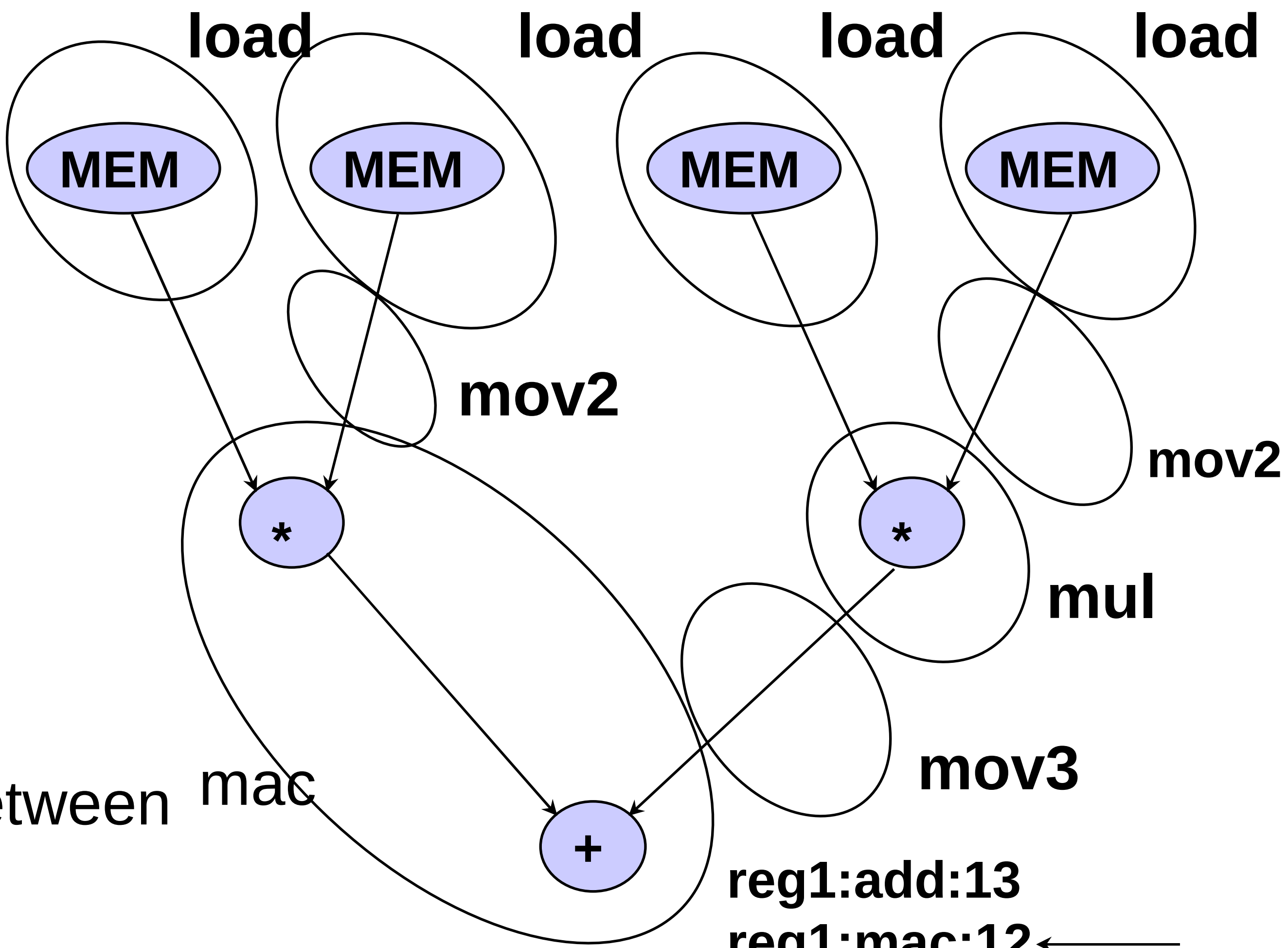
Code selection by tree parsing (2)
- nodes annotated with (register/pattern/cost)-triples -

"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)



Code selection by tree parsing (3)
- final selection of cheapest set of instructions -



Includes routing of values between various registers!

technische universität
dortmund

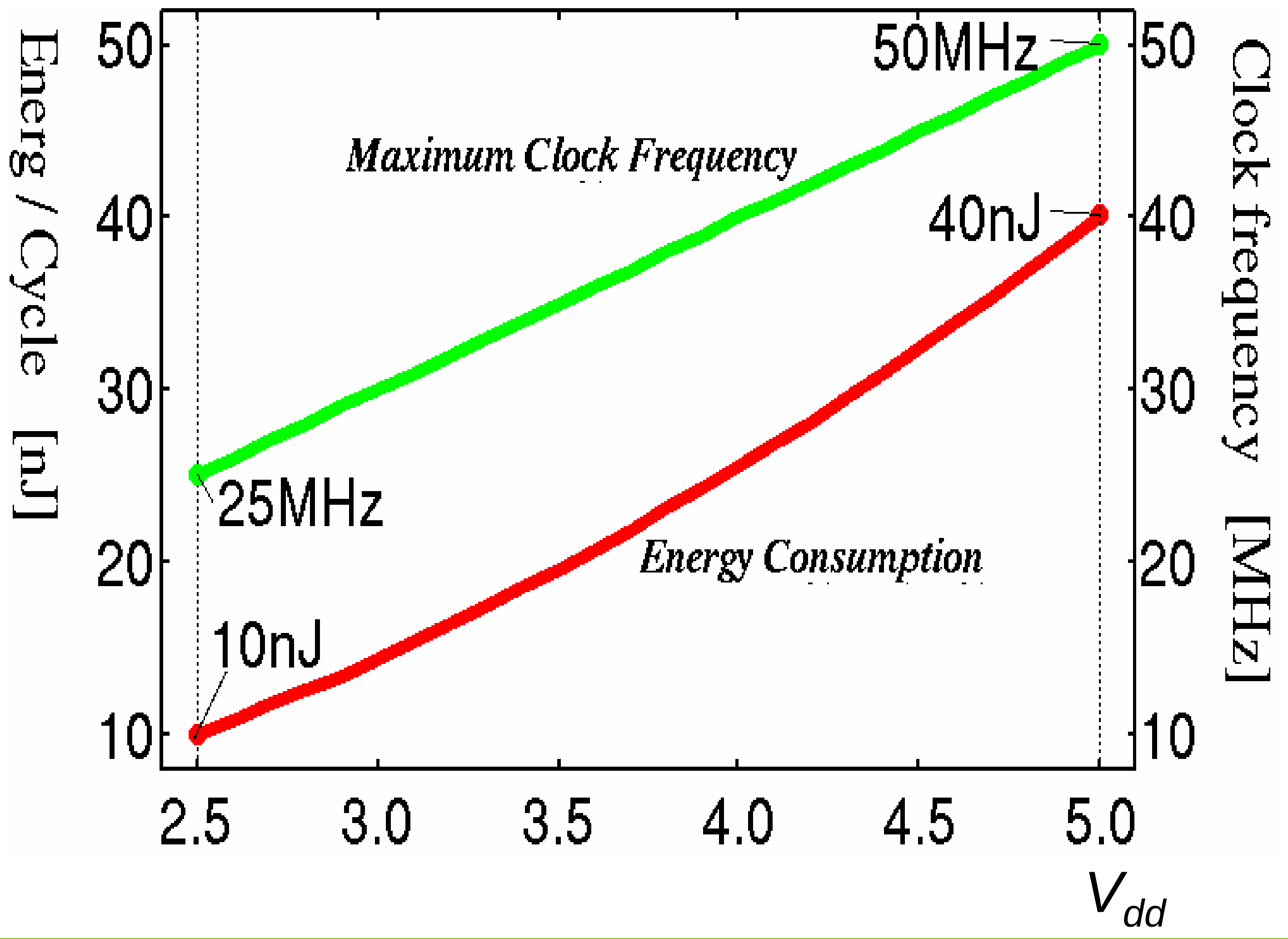
fakultät für informatik
informatik 12

Dynamic Voltage Scaling

Peter Marwedel
TU Dortmund
Informatik 12
Germany

2009/01/17

Voltage Scaling and Power Management Dynamic Voltage Scaling



Recap from chapter 3: Fundamentals of dynamic voltage scaling (DVS)

Power consumption of CMOS circuits (ignoring leakage):

$P = \alpha \, C_L V_{dd}^2 \; f$ with

α: switching activity

C_L : load capacitance

V_{dd} : supply voltage

f : clock frequency

Delay for CMOS circuits:

$\tau = k \, C_L \frac{V_{dd}}{(V_{dd} - V_t)^2}$ with

V_t : threshhold voltage

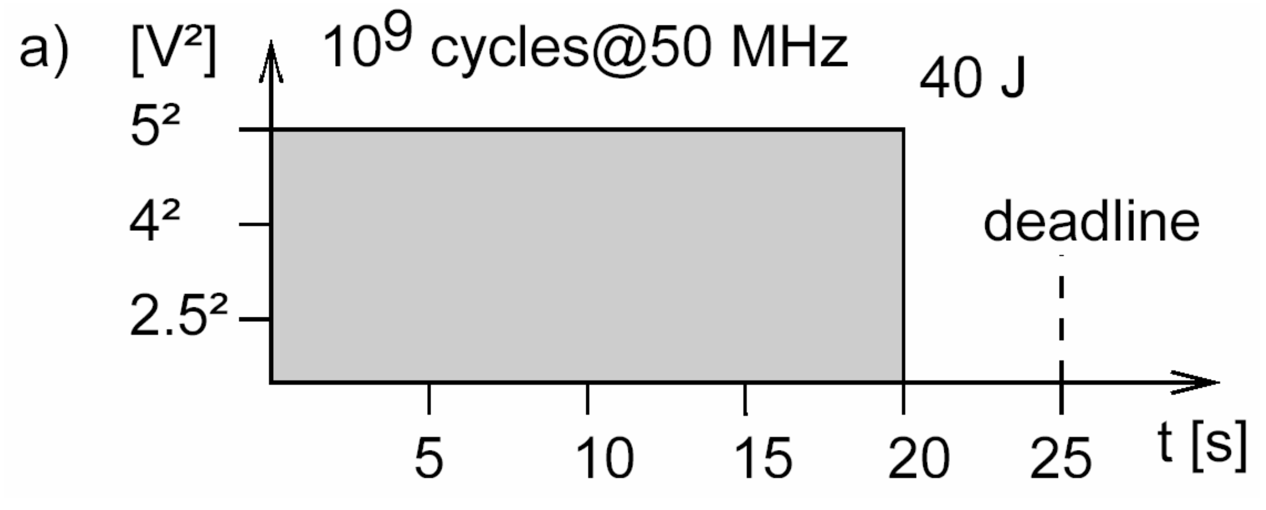
(V_t substancially $<$ than V_{dd})

☞ Decreasing V_{dd} reduces P quadratically,
while the run-time of algorithms is only linearly increased (ignoring the effects of the memory system).

Example: Processor with 3 voltages Case a): Complete task ASAP

Task that needs to execute 10^9 cycles within 25 seconds.

V_{dd} [V]	5.0	4.0	2.5
Energy per cycle [nJ]	40	25	10
f_{max} [MHz]	50	40	25
cycle time [ns]	20	25	40

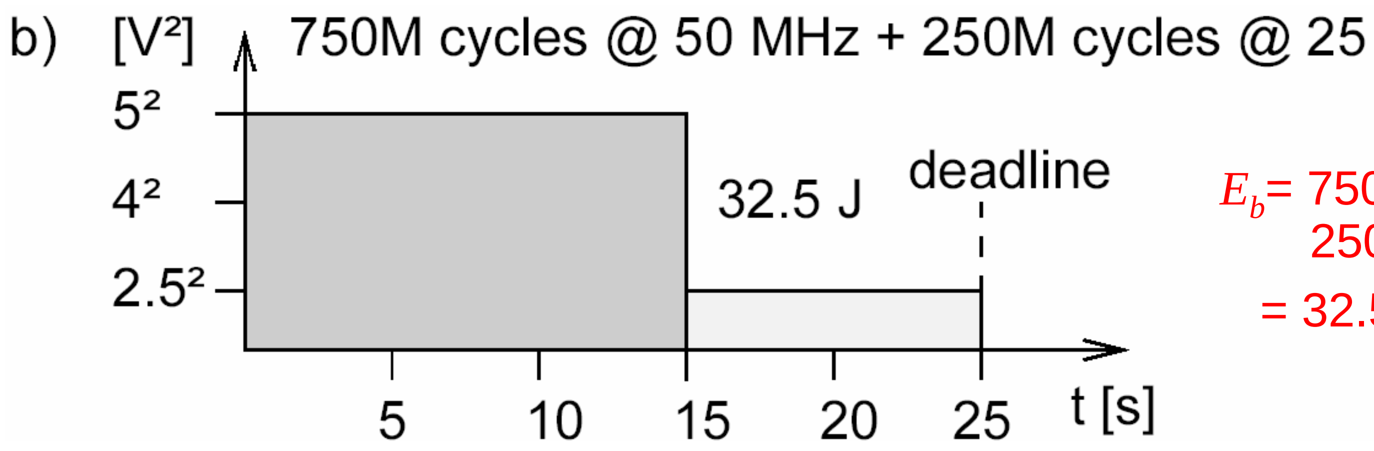


$E_a = 10^9 \times 40 \times 10^{-9}$ [J]
$= 40$ [J]

Case b): Two voltages

V_{dd} [V]	5.0	4.0	2.5
Energy per cycle [nJ]	40	25	10
f_{max} [MHz]	50	40	25
cycle time [ns]	20	25	40

b)

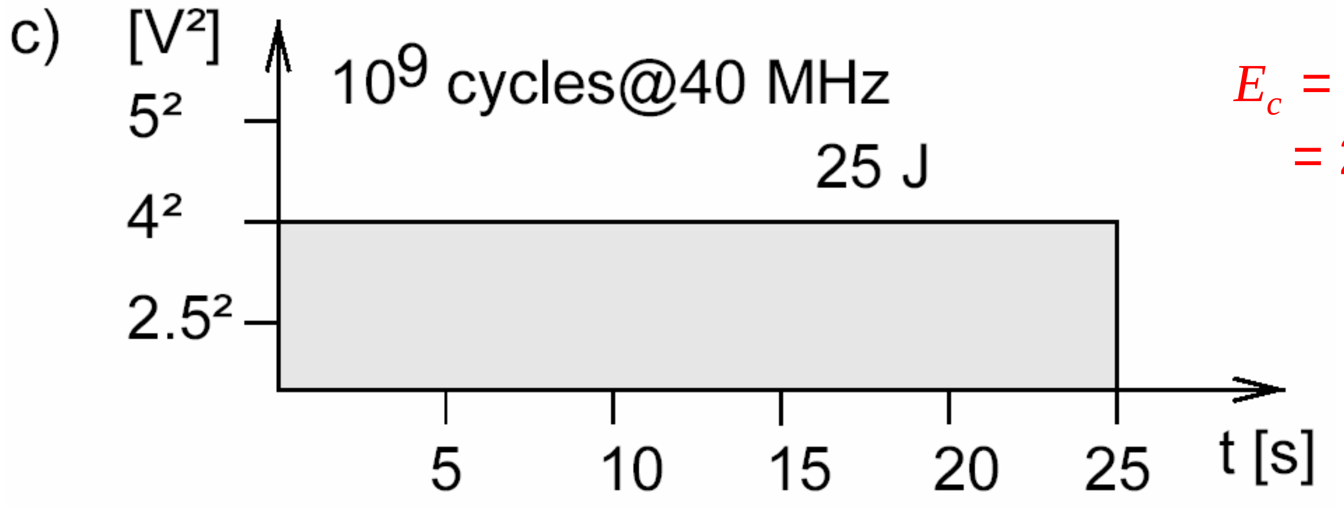


E_b= 750 10^6 x 40 10^{-9} +
250 10^6 x 10 10^{-9} [J]
= 32.5 [J]

Case c): Optimal voltage

V_{dd} [V]	5.0	4.0	2.5
Energy per cycle [nJ]	40	25	10
f_{max} [MHz]	50	40	25
cycle time [ns]	20	25	40

c)



$E_c = 10^9 \times 25 \times 10^{-9}$ [J]

$= 25$ [J]

Observations

- A minimum energy consumption is achieved for the ideal supply voltage of 4 Volts.
- In the following: **variable voltage processor =** processor that allows **any** supply voltage up to a certain maximum.
- It is expensive to support truly variable voltages, and therefore, actual processors support only a few fixed voltages.

Generalisation

Lemma [Ishihara, Yasuura]:

- If a variable voltage processor completes a task before the deadline, the energy consumption can be reduced.
- If a processor uses a single supply voltage V and completes a task T just at its deadline, then V is the unique supply voltage which minimizes the energy consumption of T.
- If a processor can only use a number of discrete voltage levels, then the two voltages which (almost*) minimize the energy consumption are the two immediate neighbors of the ideal voltage V_{ideal} possible for a variable voltage processor.

----

* Except for small amounts of energy resulting from the fact that cycle counts must be integers.

The case of multiple tasks: Assignment of optimum voltages to a set of tasks

N : the number of tasks

EC_j : the number of executed cycles of task j

L : the number of voltages of the target processor

V_i : the i^{th} voltage, with $1 \leq i \leq L$

F_i : the clock frequency for supply voltage V_i

T : the global deadline at which all tasks must have been completed

$X_{i,j}$: the number of clock cycles task j is executed at voltage V_i

SC_j : the average switching capacitance during the execution of task j (SC_j comprises the actual capacitance CL and the switching activity α)

Designing an ILP model

Simplifying assumptions of the ILP-model include the following:

- There is one target processor that can be operated at a limited number of discrete voltages.
- The time for voltage and frequency switches is negligible.
- The worst case number of cycles for each task are known.

Energy Minimization using an Integer Linear Programming Model

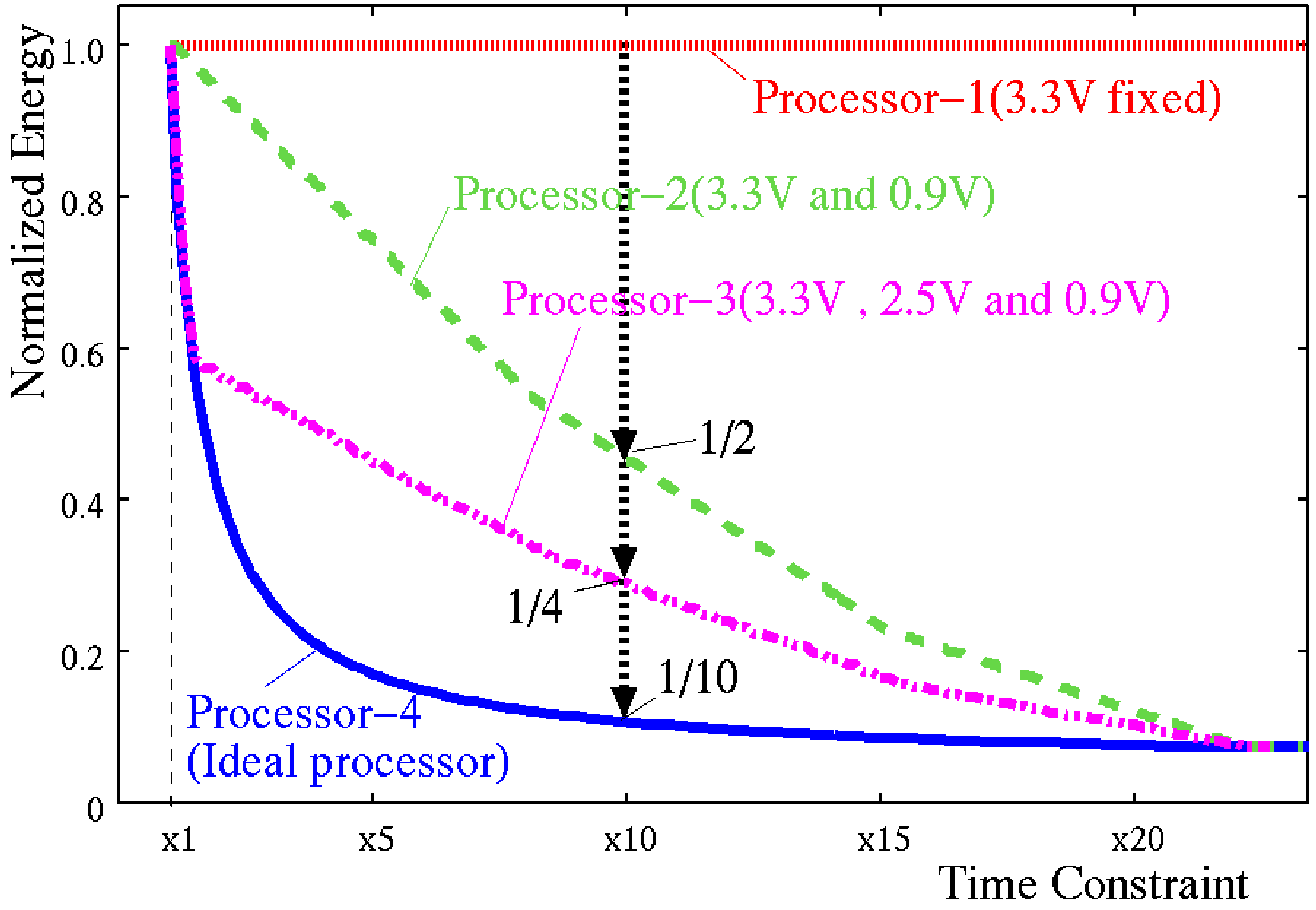
Minimize $$E = \sum_{j=1}^{N} \sum_{i=1}^{L} SC_j \cdot x_{i,j} \cdot V_i^2$$

subject to $$\forall j : \sum_{i=1}^{L} x_{i,j} = EC_j$$

and $$\sum_{i=1}^{L} \sum_{j=1}^{N} \frac{x_{i,j}}{F_i} \leq T$$

Experimental Results

3 tasks; EC_j=50 10^9; SC_1=50 pF; SC_2=100 pF; SC_3=150 pF

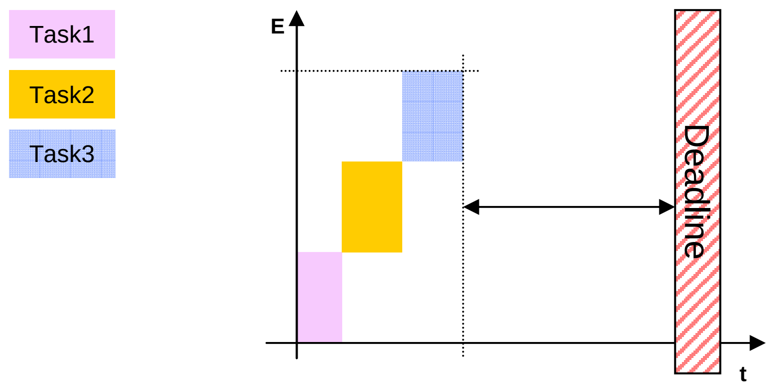


Task-level concurrency management

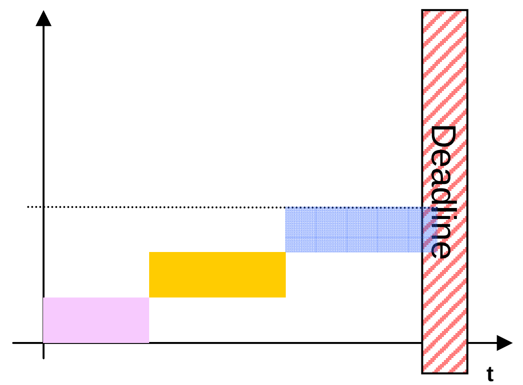
- The dynamic behavior of applications getting more attention.
- Energy consumption reduction is the main target.
- Some classes of applications (i.e. video processing) have a considerable variation in processing power requirements depending on input data.
- Static design-time methods becoming insufficient.
- Runtime-only methods not feasible for embedded systems.

→ How about mixed approaches?

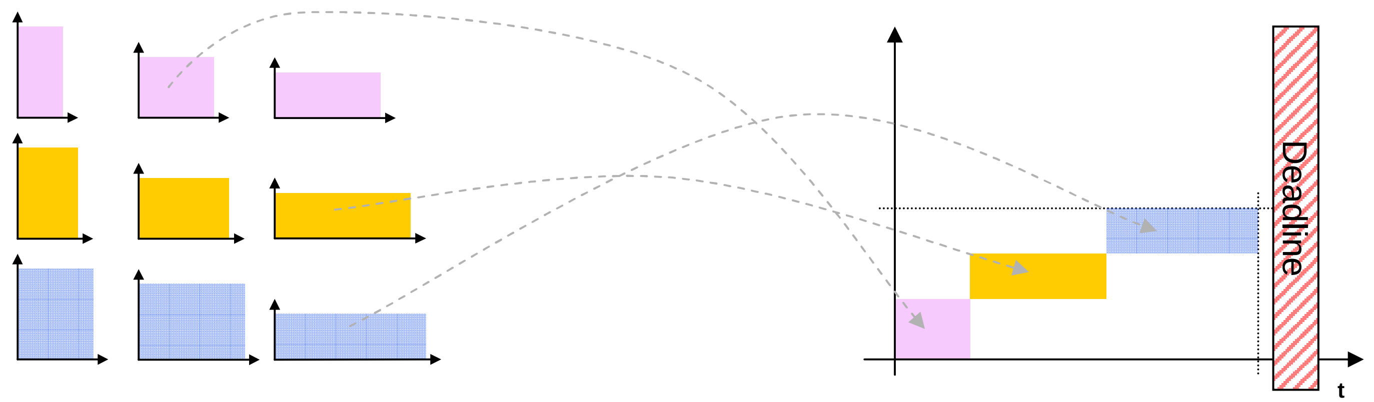
Example of a mixed TCM



Static (compile-time) methods can ensure WCET feasible schedules, but waste energy in the average case.



…or they can define a probability for violating the deadline.



Mixed methods use compile-time analysis to define a set of possible execution parameters for each task.

Runtime scheduler selects the most energy saving, deadline preserving combination.

[IMEC, Belgium, http://www.imec.be/]

Dynamic power management (DPM)

Dynamic Power management tries to assign optimal power saving states.

- Questions: When to go to an power-saving state? Different, but typically complex models:
- Markov chains, renewal theory , ….

Summary

- Worst-case execution time aware compilation
- The offset assignment problem
- Retargetability and code selection
- Dynamic voltage scaling (DVS)
  - An ILP model for voltage assignment in a multi-tasking system
- Dynamic power management (DPM) (briefly)