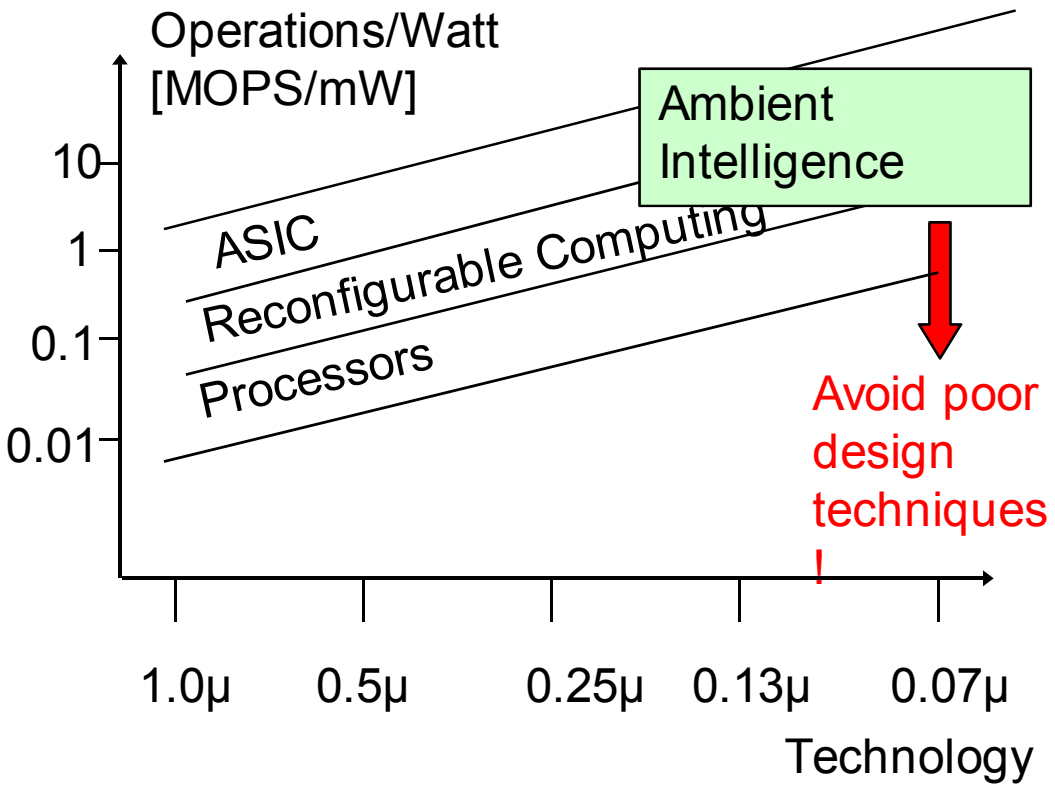
# Key compiler algorithms (for embedded systems)

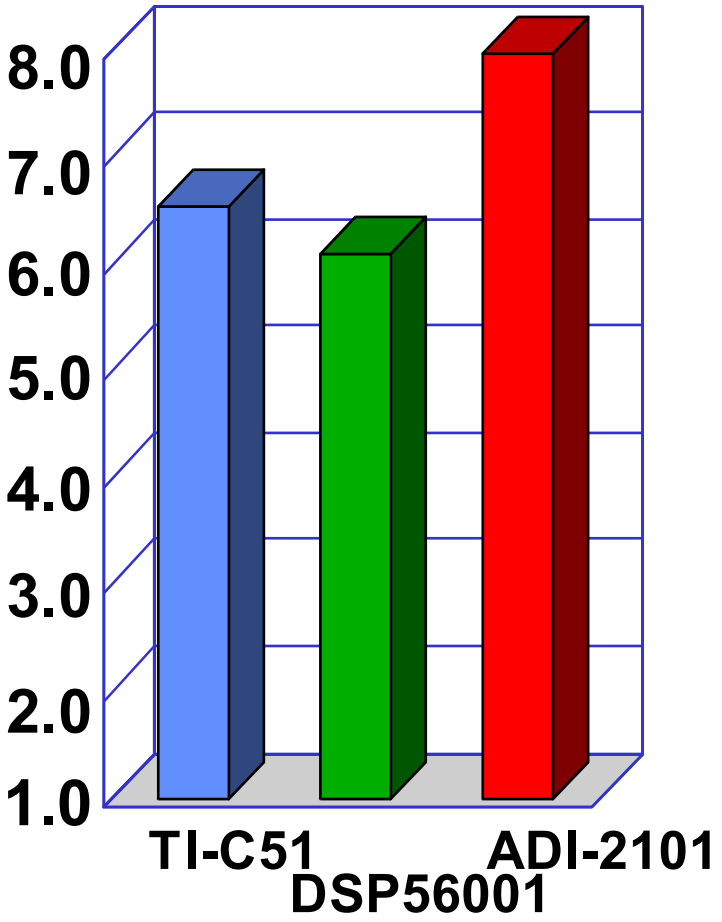
Peter Marwedel
University of Dortmund, Germany

# Motivation: Wishes and Reality




Results for DSPStone benchmark

***Cycle overhead* [× n]**

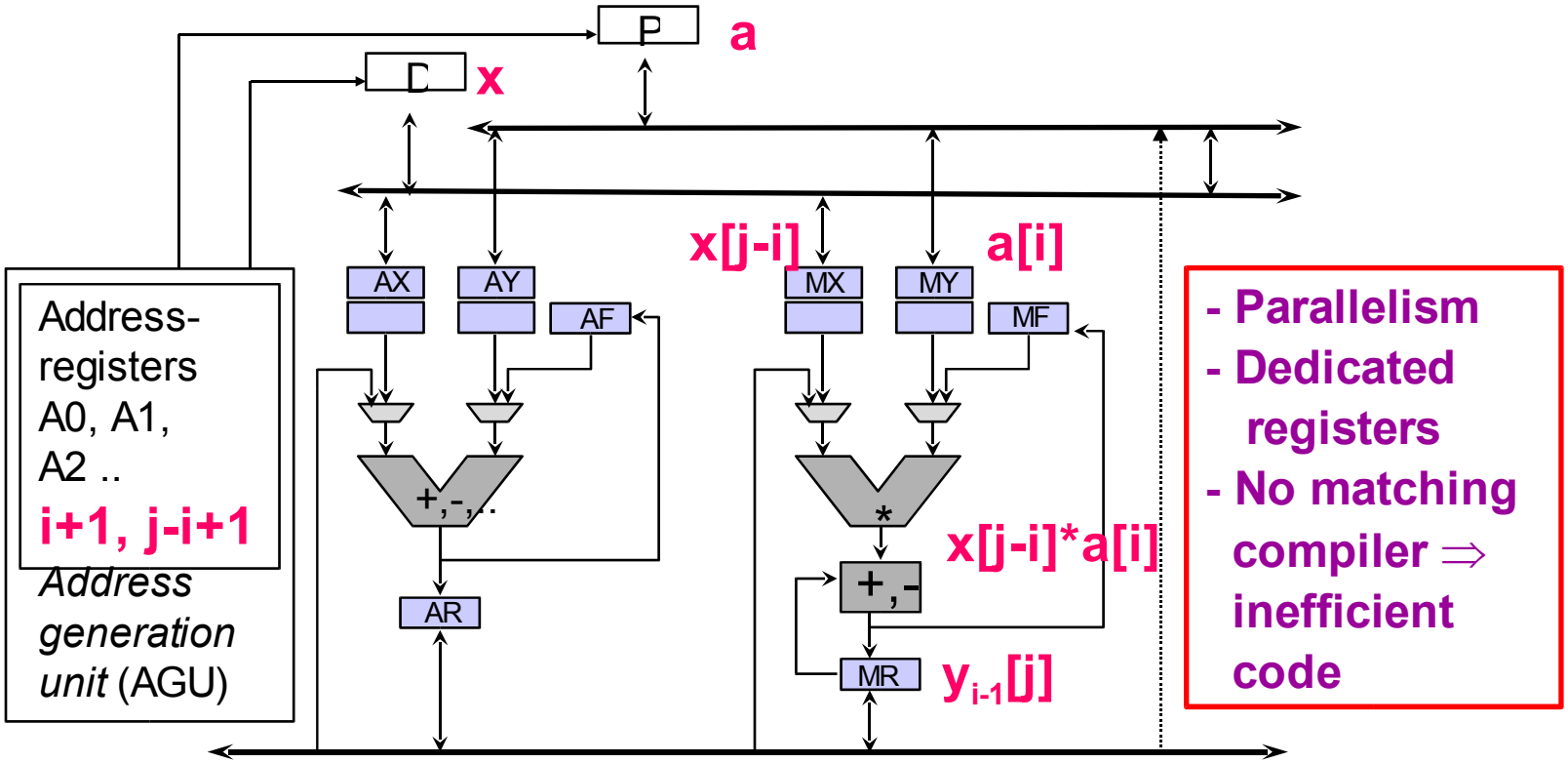


# Reason for compiler-problems: Application-oriented Architectures

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

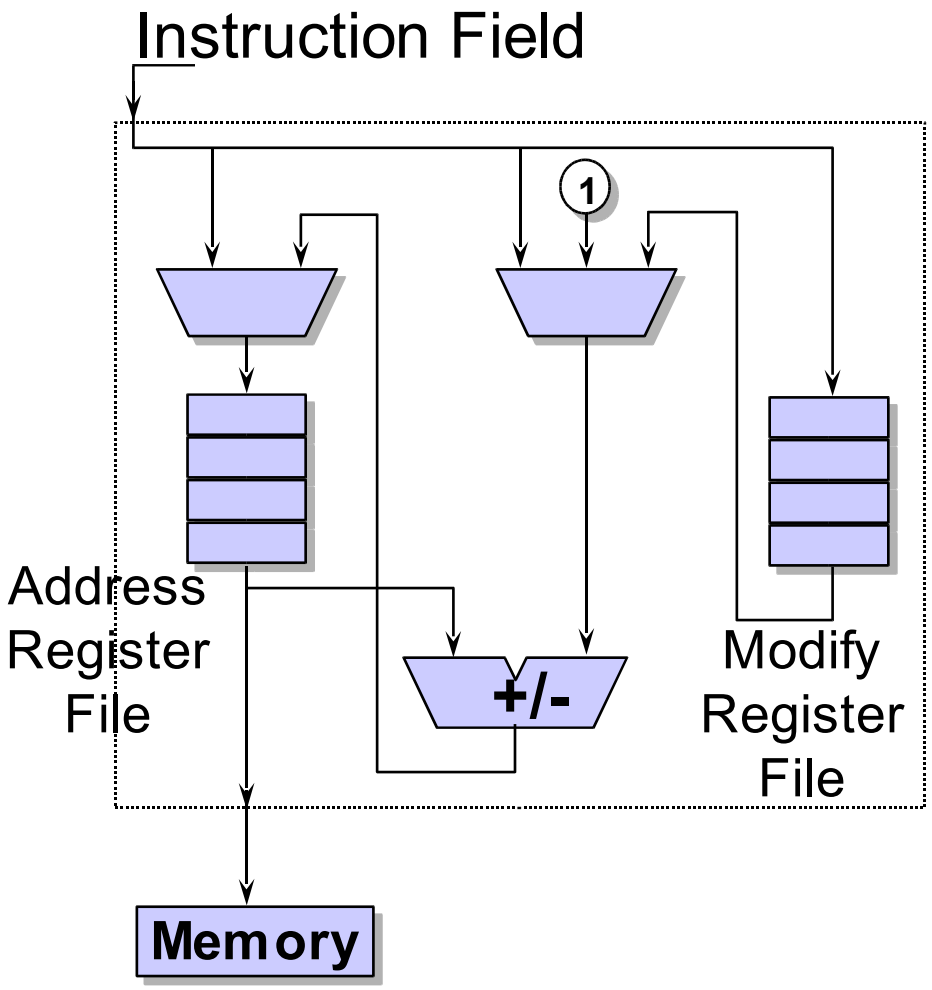
$\forall i: 0 \le i \le n: y_i[j] = y_{i-1}[j] + x[j-i]*a[i]$

**Architecture:** Example: Data path ADSP210x



# Exploitation of parallel address computations

Generic address generation unit (AGU) model




**Parameters:**

*k* = # address registers

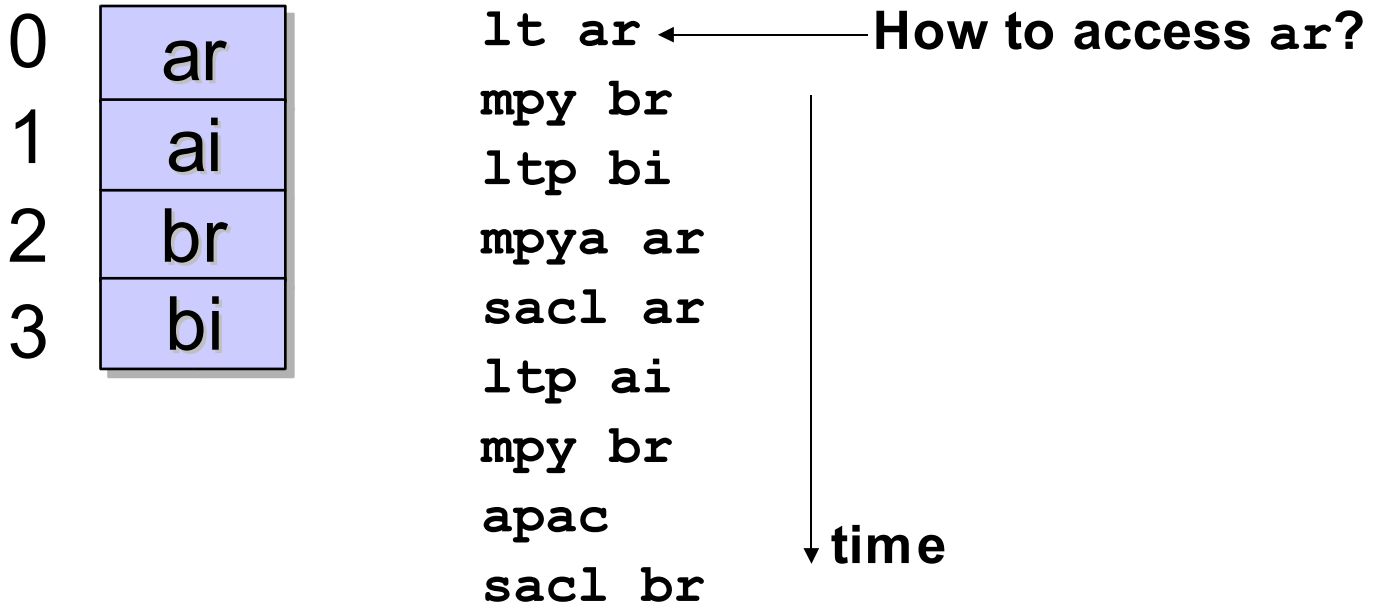
*m* = # modify registers

**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)

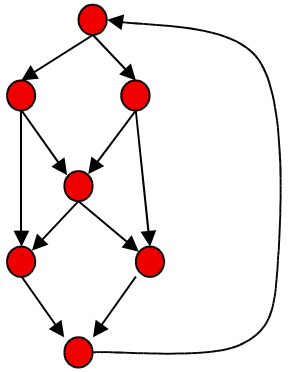
**Given: Memory layout + assembly code (without address code)**




**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



$u$: $E \to \mathbb{R}_{\geq 0}$ is a capacity function,

$c$: $E \to \mathbb{R}$ is a cost function; $n = |V|$, $m = |E|$.

**Definition:**

$g$: $E \to \mathbb{R}_{\geq 0}$ 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)

$g$ is **feasible** if $\forall (v,w) \in E$: $g(v,w) \leq u(v,w)$ (capacity constraints)

The cost of a circulation $g$ is $c(g) = \sum_{(v,w)\in E} c(v,w)\, g(v,w)$.

There may be a lower bound on the flow through an edge.

The **minimum cost circulation problem** is to find a feasible circulation of minimum cost.

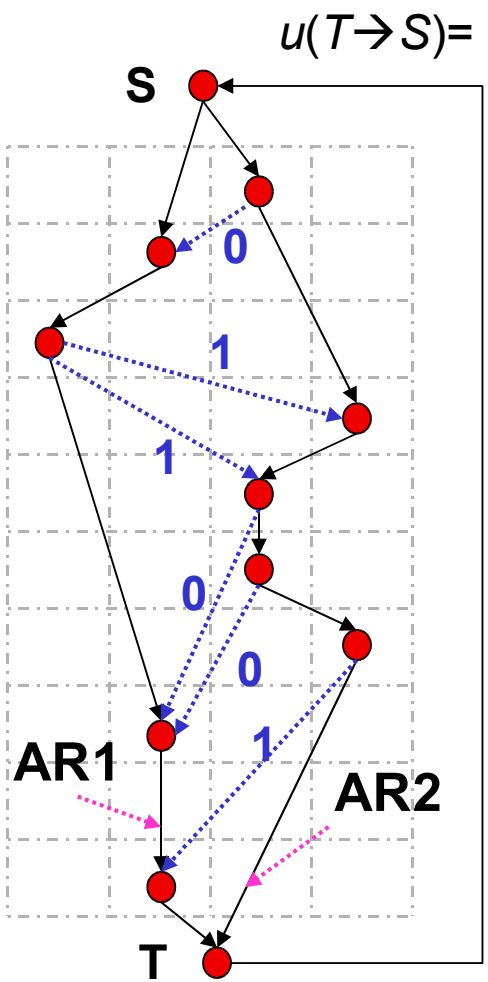
**[K.D. Wayne: A Polynomial Combinatorial Algorithm for Generalized Minimum Cost Flow, http://www.cs.princeton.edu/ ~wayne/ papers/ ratio.pdf]**

# 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
```

time




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**

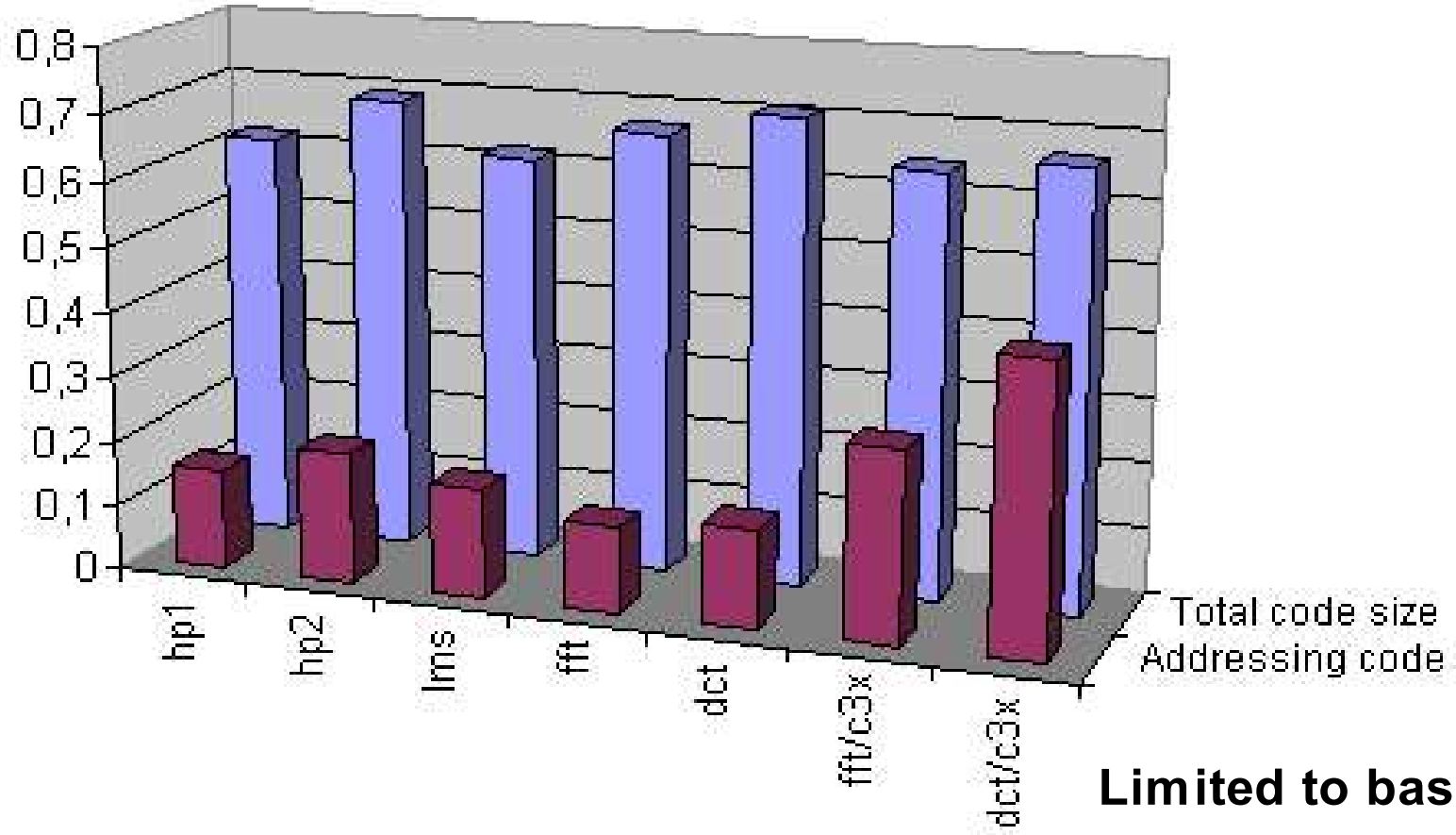
**additional edges of original graph (only samples shown)**

* C2x processor from ti

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

# Results according to Gebotys

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

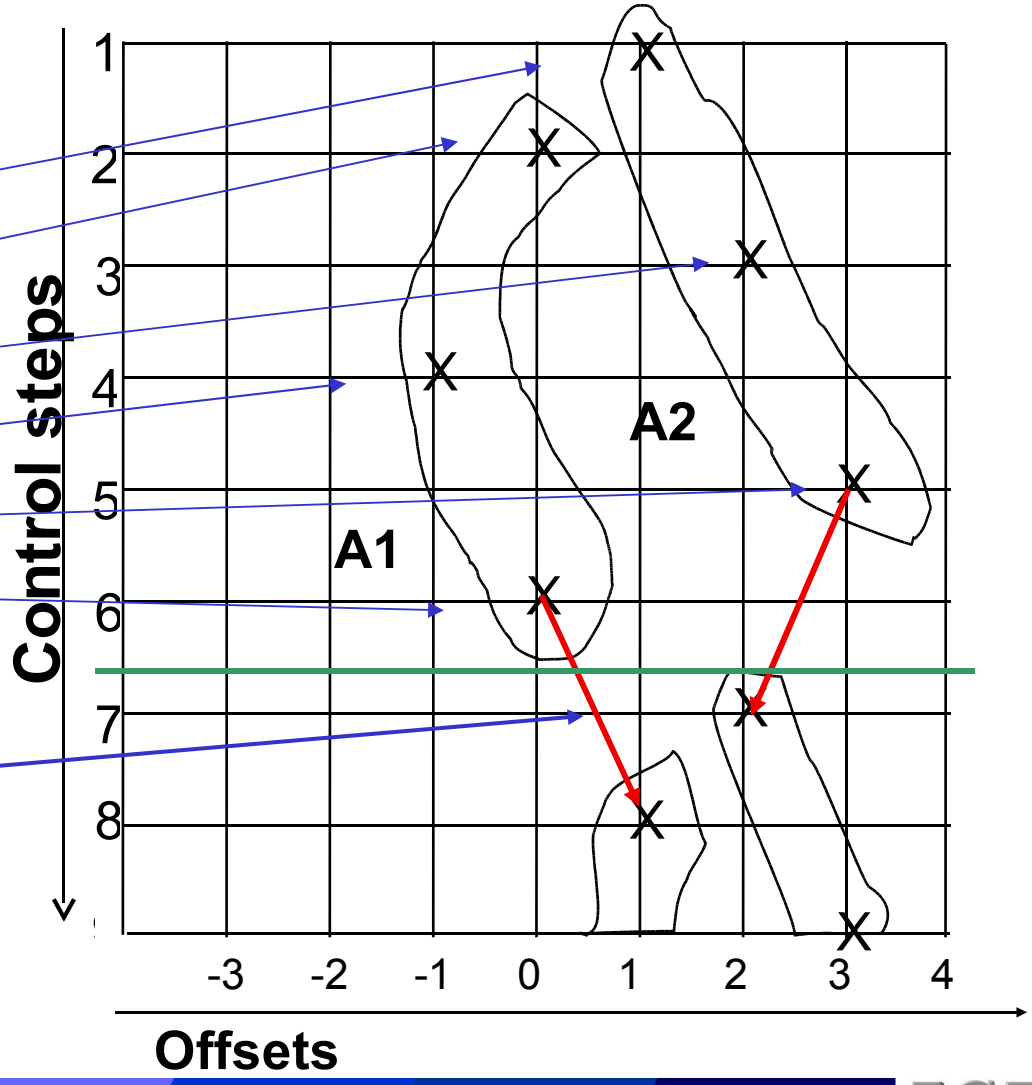



**Limited to basic blocks**

# 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

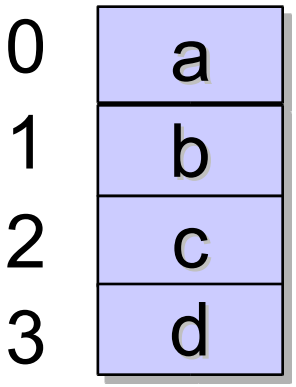
The case (1,0,1) is called simple offset assignment (SOA),

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

# ☞ SOA example
## - Effect of optimised memory layout -

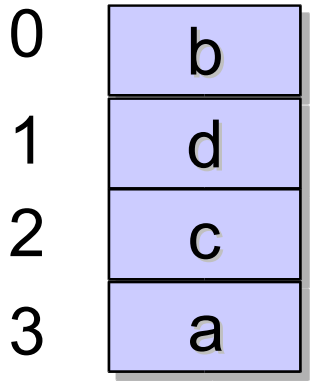
Variables in a basic block: Access sequence:

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




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

cost: 4

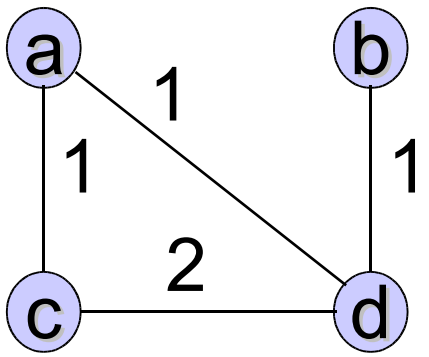



```
Load AR,0  ;b
AR ++      ;d
AR +=2     ;a
AR --      ;c
AR --      ;d
AR ++      ;c
```

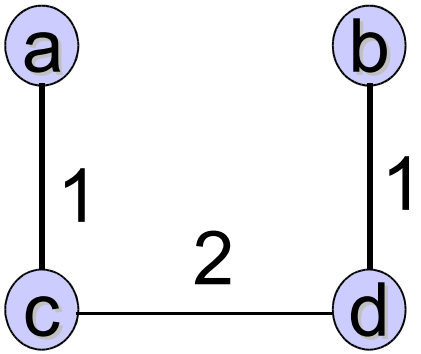
cost: 2

# SOA example: Access sequence, access graph and Hamiltonian paths

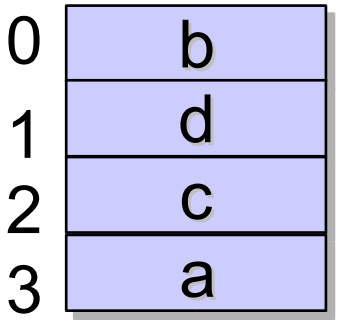
access sequence: b d a c d c




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

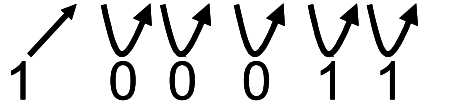
[Bartley, 1992; Liao, 1995]

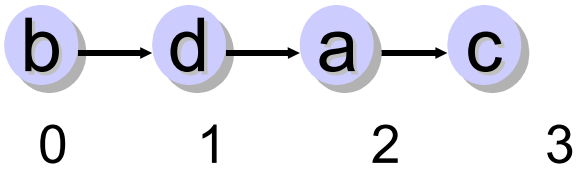
# Naïve SOA

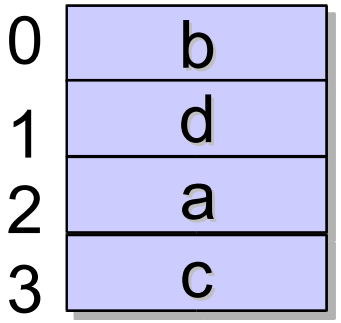
Nodes are added in the order
in which they are used in the program.

Example:

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



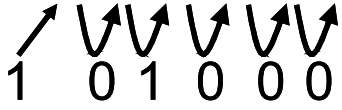

memory layout

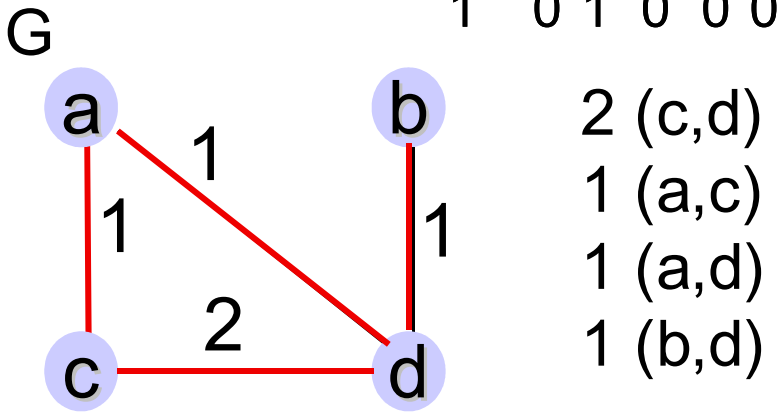
# Liao's algorithm

Similar to Kruskal's spanning tree algorithms:

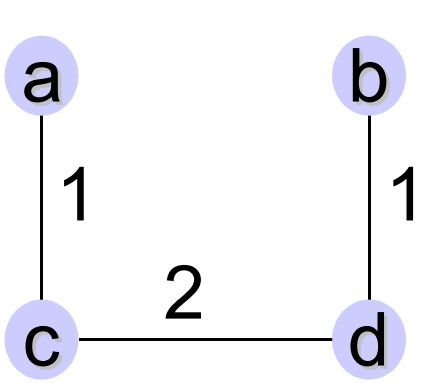
1. Sort edges of access graph G=(V,E) according to their weight
2. Construct a new graph G'=(V',E'), starting with E'=0
3. Select an edge *e* of G of highest weight; If this edge does not cause a cycle in G' and does not cause any node in G' to have a degree > 2 then add this node to E' otherwise discard *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)*




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

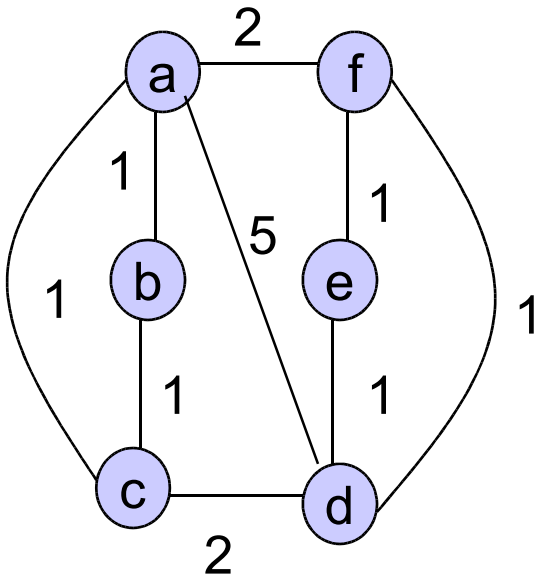



Implicit edges of weight 0 for all unconnected nodes.

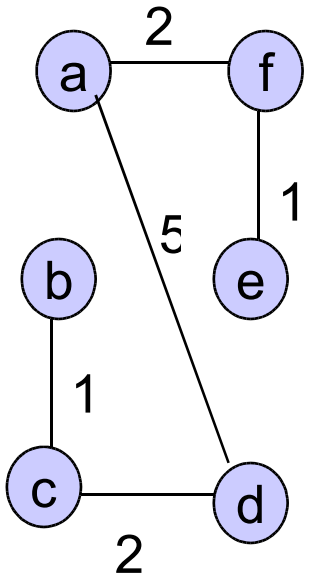
# Liao's algorithm on a more complex graph

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

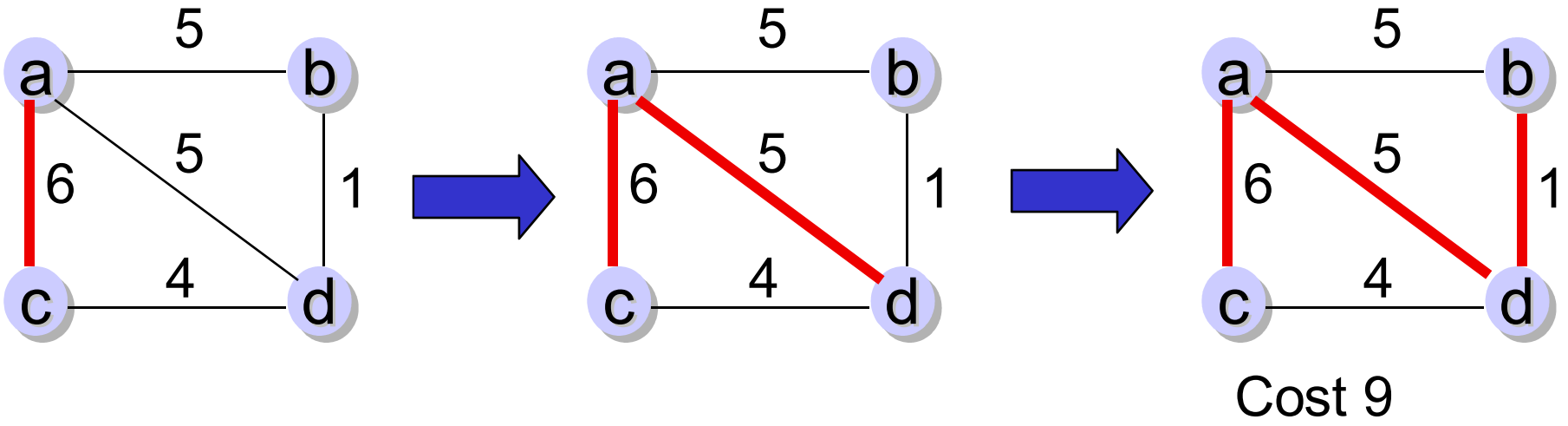
G

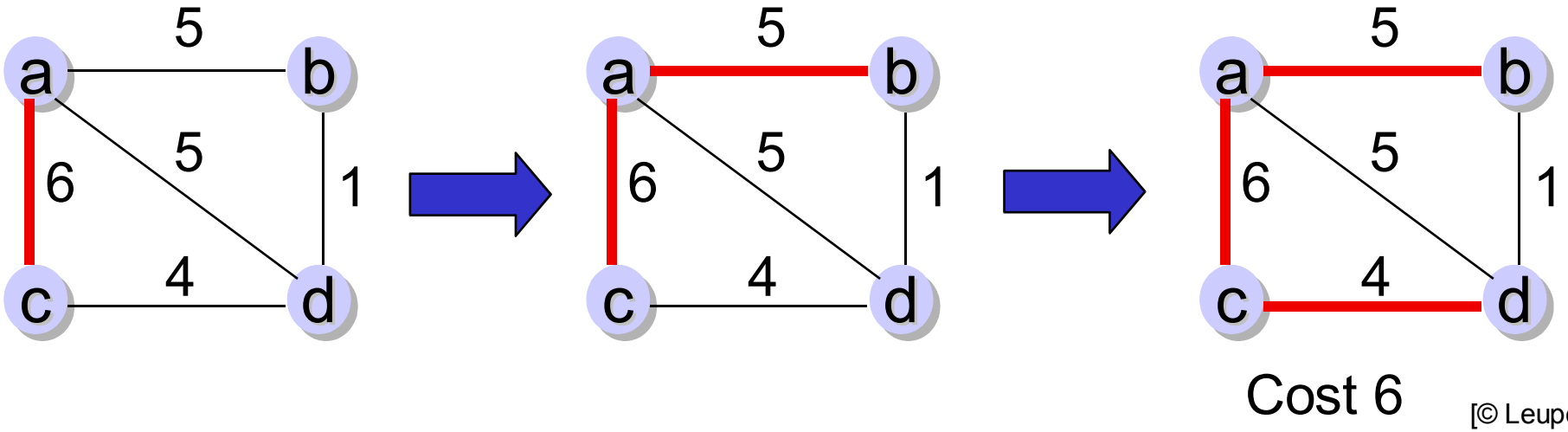



G'



# Liao's algorithm with tie break heuristic
## - Motivation -



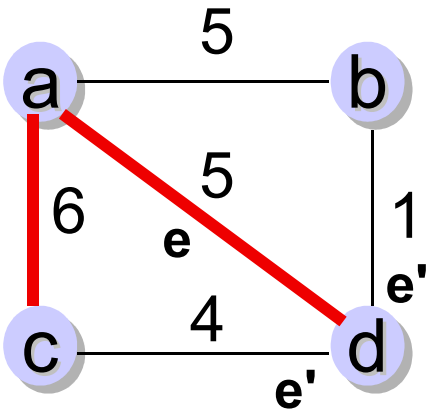




[© Leupers]

# Tiebreak function

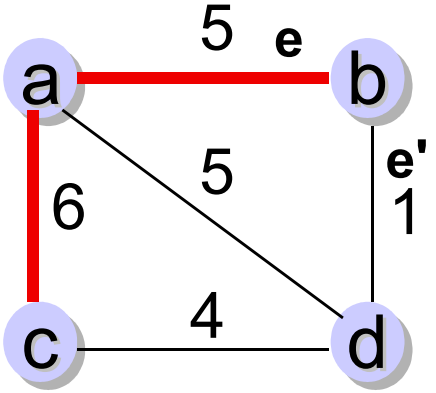
Access graph ($V$,$E$,$w$), for each edge $e$ with weight $w(e)$:

$$T(e) = \sum_{e' \in E, e' \cap e \neq \emptyset} w(e')$$




**$T(e)=5$**

In case of alternative equal-weight edges: give priority to the edge with lower weight $T(e)$, because this will (generally) leave more freedom for selecting further high-weight edges




**$T(e)=1$**

# Comparison of different SOA algorithms

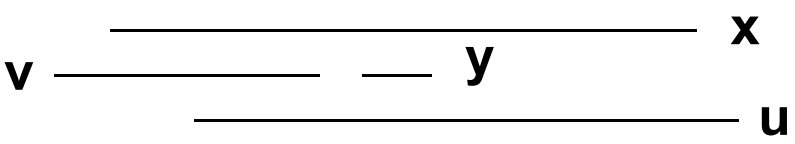
| | Overhead [%] | Run-time [msecs] |
|---|---|---|
| Declaration Order | 67.09 | 0 |
| Liao, as presented | 4.34 | 0.67 |
| Liao, with tie break heuristics for edges of same weight | 0.16 | 0.68 |
| Atri: iterative improvement of initial SOA solution | 2.28 | 4.6 |
| Same as Atri, with tie-break heuristic added | 0.11 | 23 |
| Genetic algorithm [Leupers] | 0 | 8296.26 |
| Branch & Bound (for simple cases) | 0 | >> |

# Variable coalescing SOA (CSOA)

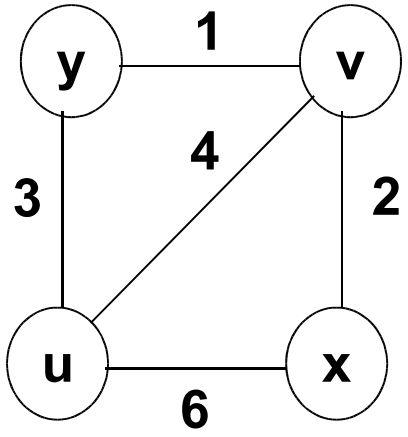
Example:

## Access sequence

(v,x,v,u,v,u,v,y,u,y,u,x,u,x,u,x,u)

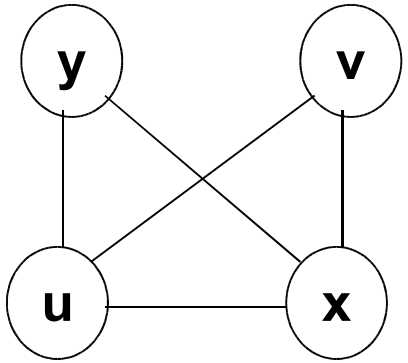



## Access graph

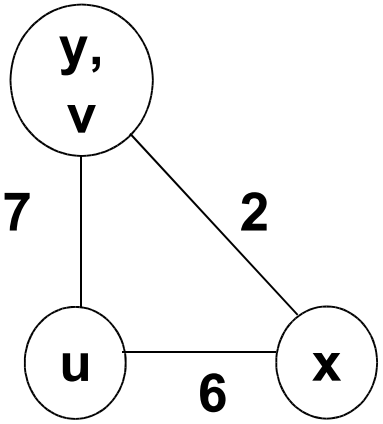



**Iterative extension of Hamilton path and coalescing**

## Interference graph



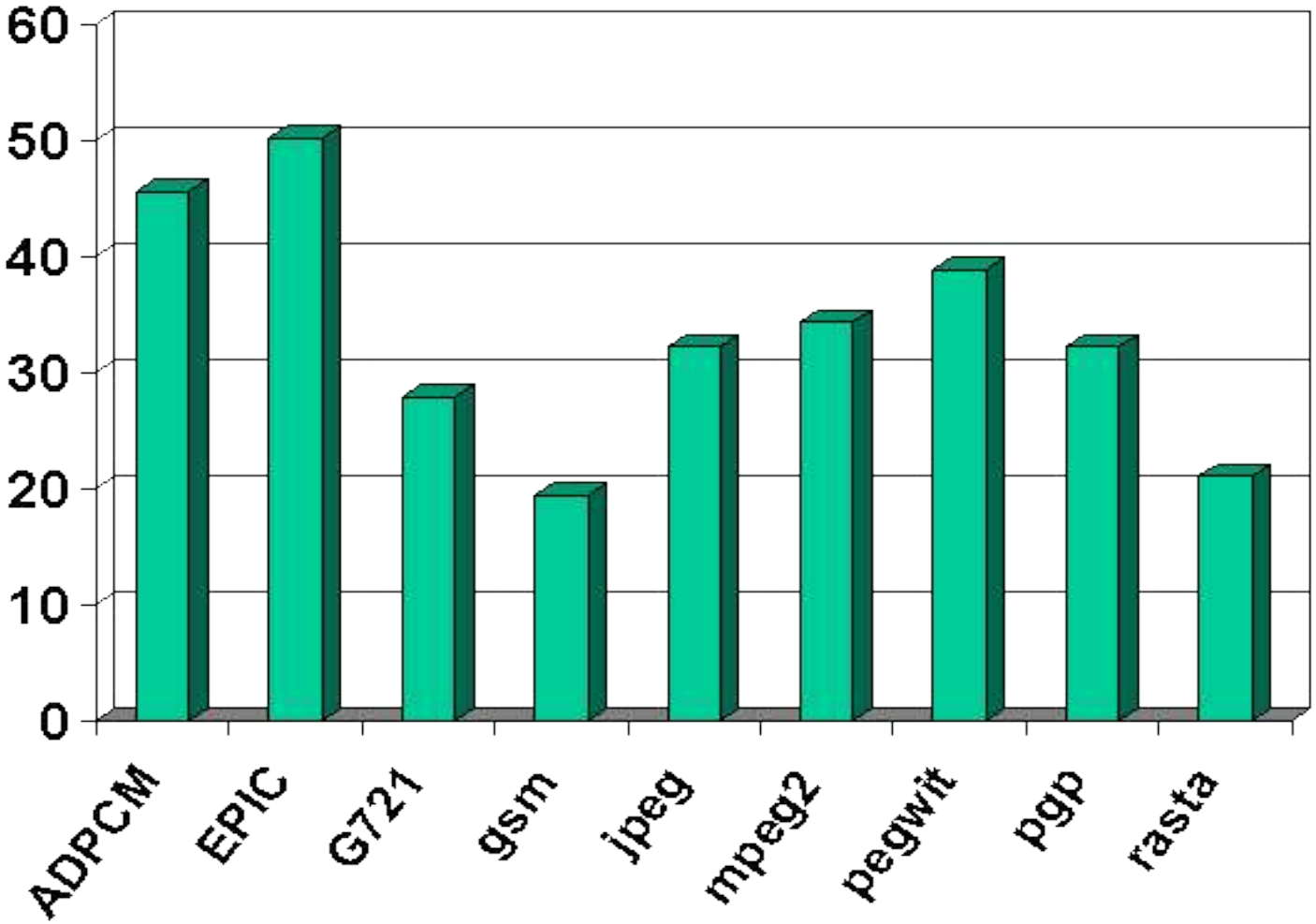

## Access graph




## Memory layout

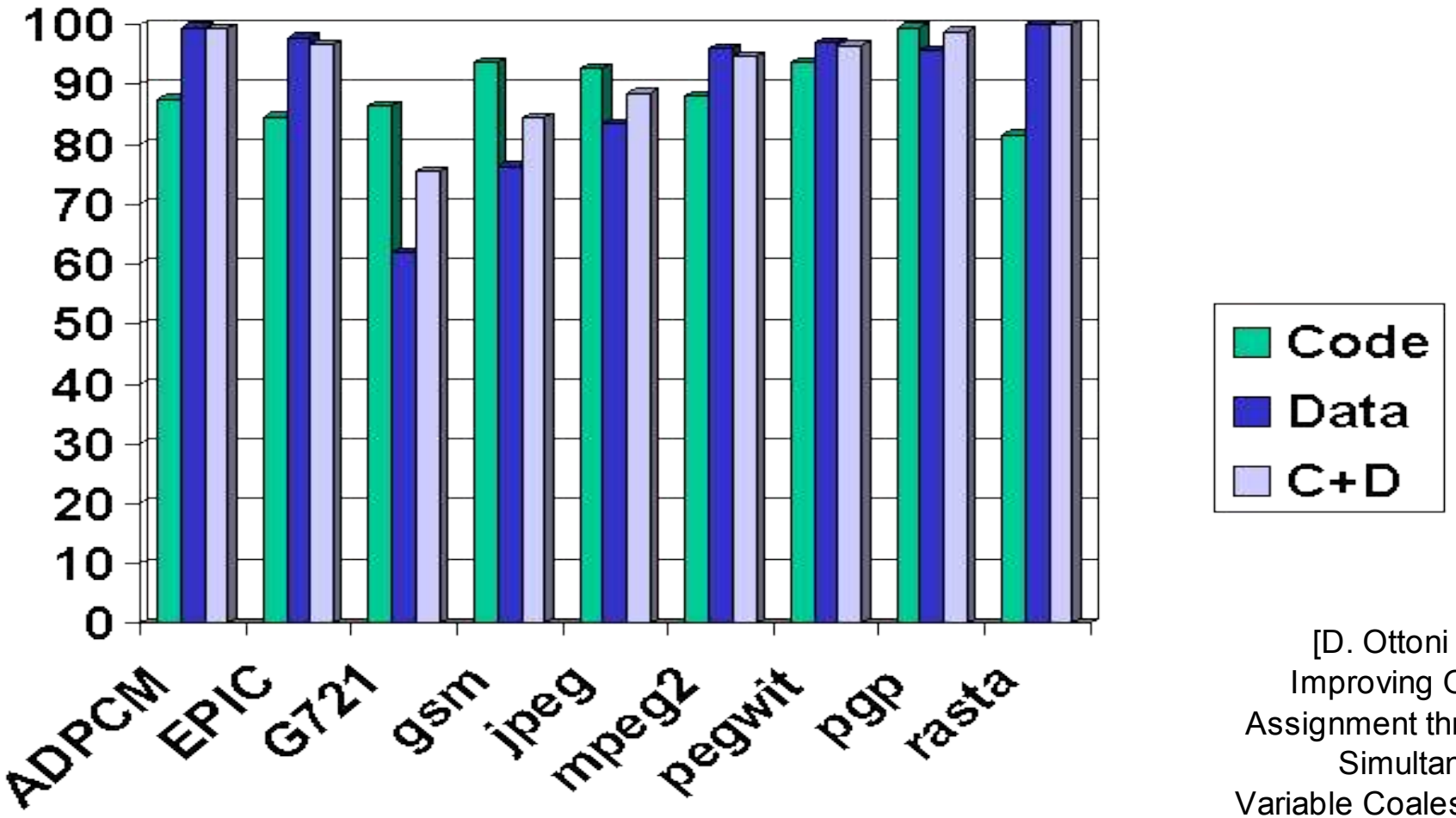
| y,v |
|---|
| u |
| x |

# Offset costs relative to Liao [%]




[Desiree Ottoni et al.: Improving Offset Assignment through Simultaneous Variable Coalescing, SCOPES, 2003]

# Memory size reductions [%]




[D. Ottoni et al.: Improving Offset Assignment through Simultaneous Variable Coalescing, SCOPES, 2003]

# Integrated Scheduling and SOA: Sch-SOA
## - Result for SOA -

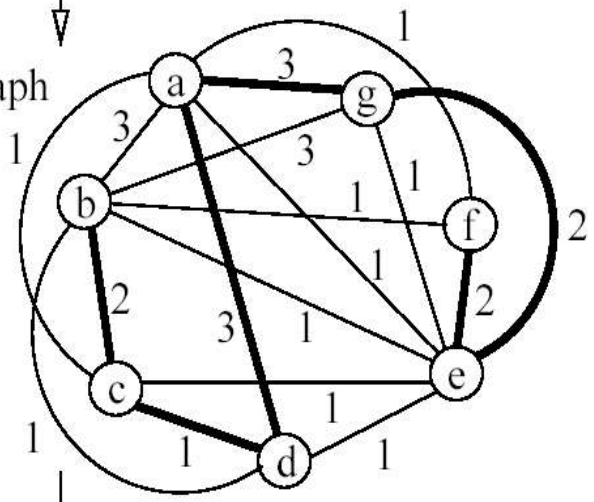
```
    b = f + a;
2)  a = f + e;
3)  g = c − b;
    g = a + b;
    b = g + e;
6)  d = c + b;
    d = d + a;
8)  g = c − e;
    a = a + a;
    f = d − e;
```

(a) Input C code

(i) access sequence:

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

(ii) access graph




(iii) memory layout:

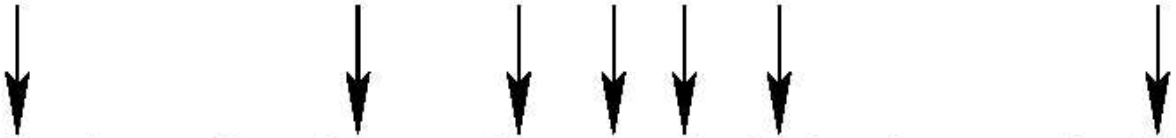
| b | c | d | a | g | e | f |
|---|---|---|---|---|---|---|

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

# of nonzero−cost accesses : 12

# Integrated Scheduling and SOA: Sch-SOA
## - Result for Sch-SOA -

```
   b = f + a;
3) g = c − b;
2) a = e + f;
   g = a + b;
   b = g + e;
8) g = c − e;
6) d = c + b;
   d = d + a;
   a = a + a;
   f = d − e;
```




memory layout:


# of nonzero−cost accesses : 7

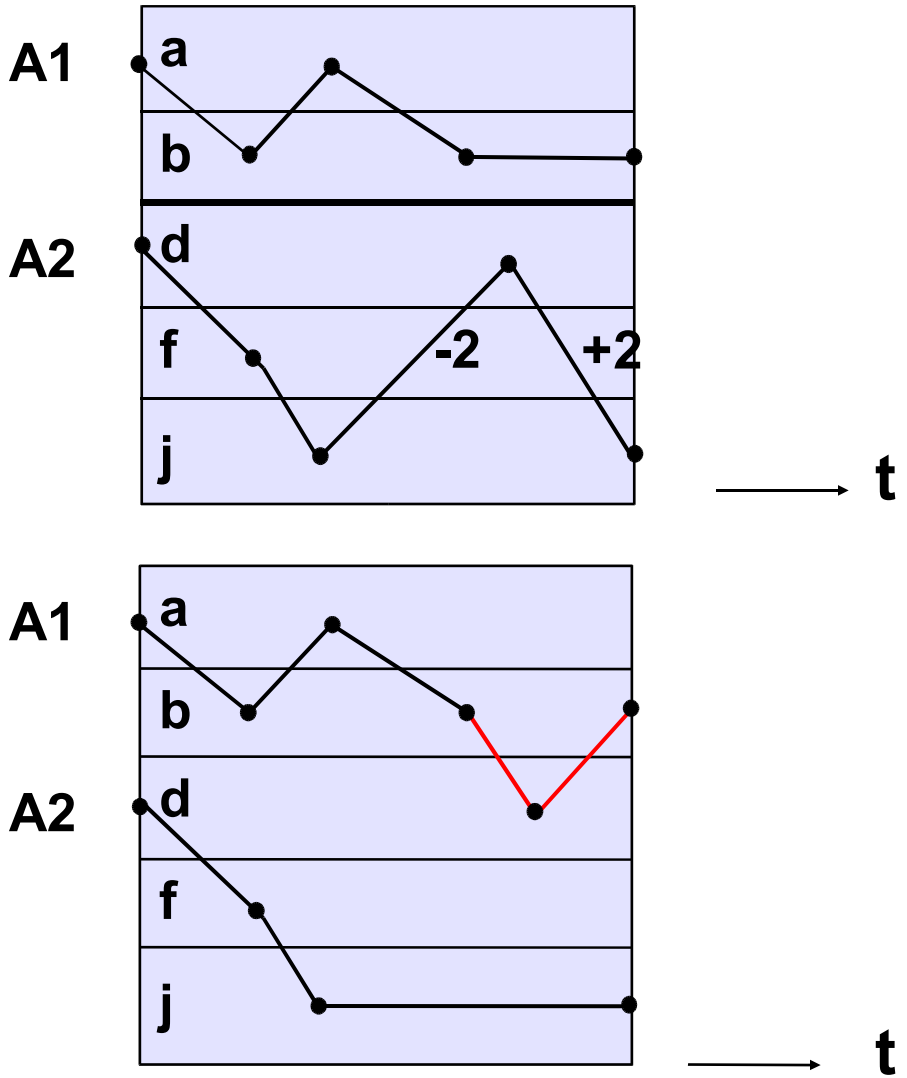
(d) Rescheduled C code with input commutativity and resulting SOA solution produced by our approach

[Y. Choi, T. Kim, Address Assignment Combined with Scheduling in DSP Code Generation, DAC, 2002, © ACM]

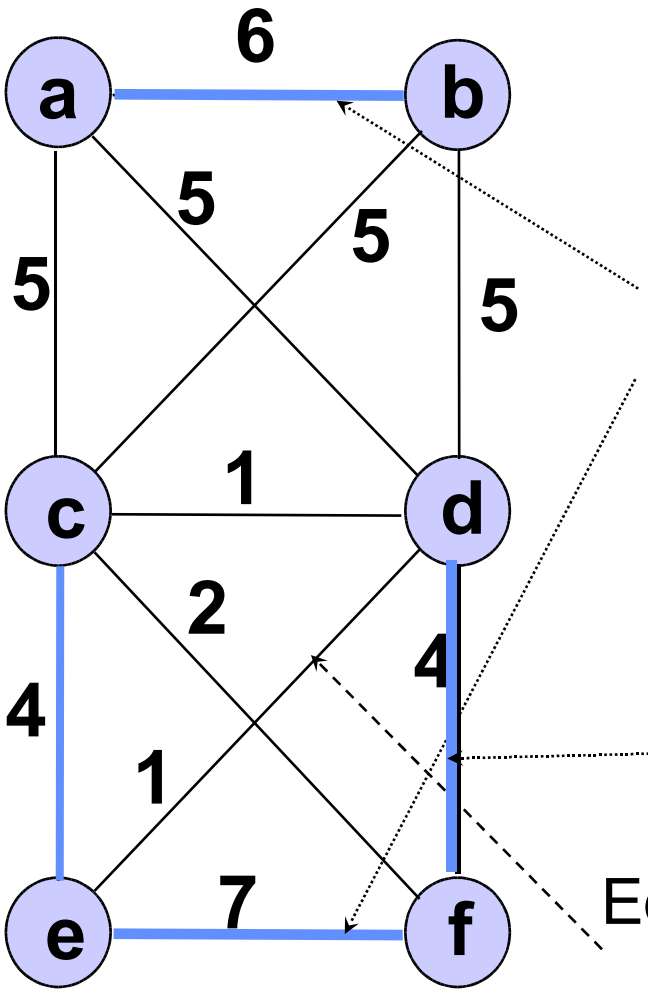
# GOA:
# Address assignment for *k* address registers

Two cases

1. All accesses to a variable done with the same address register ☞ Partitioning of variables into sets
2. Allocation of address registers for each access to a variable. Better results, but more complex.



# GOA with variable partitioning: Address assignment for *k* address registers




Seed: *k* edges of highest weight.

Iteration: Add edge(s) with smallest incremental cost for edges not covered.

Edge not covered

Disjoint sets of vars
☞ GOA → *k* × SOA

[Leupers & Marwedel, ICCAD, 1996]

# Optimised use of *m* modify-registers

**Given: Sequence of Modify-Values:**

Load AR,0
AR +=2
AR +=4
AR -=3
AR +=2
AR -=3
AR +=2
AR -=4
AR +=2
AR -=4

No separate address calculation cycle if constants are contained in modify registers

Modify registers are containers, constants represent contents.

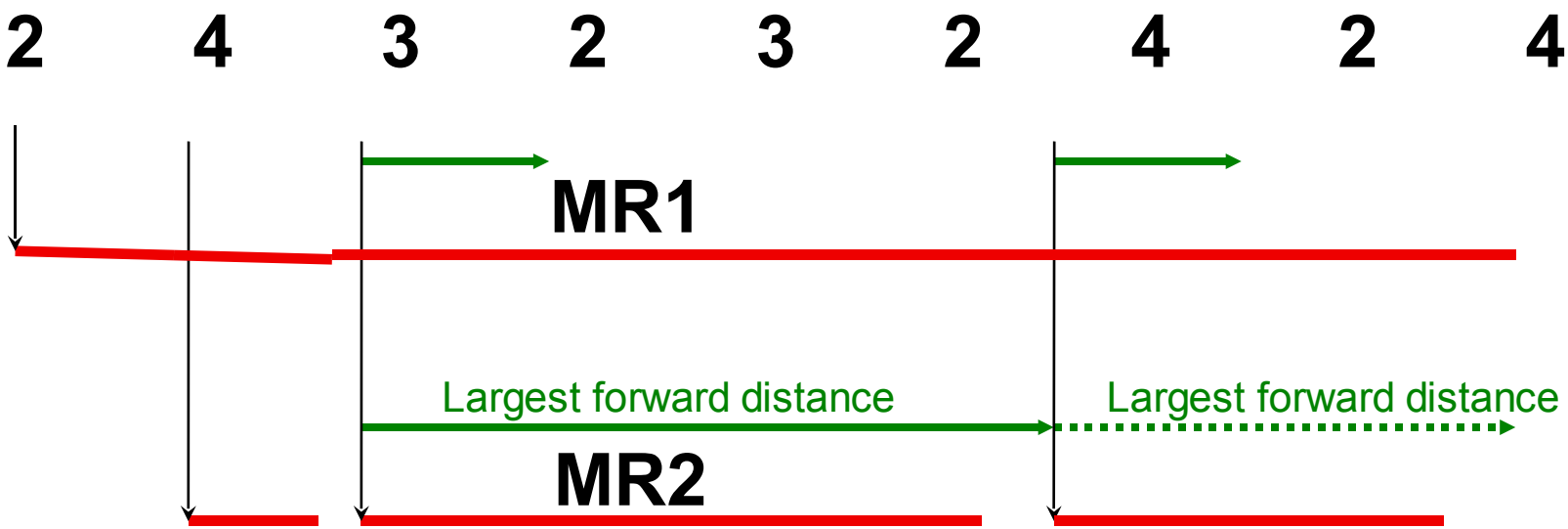
☞ Try to assign contents to containers such that content-misses occur as less frequent as possible.

☞ Equivalent to pages and page frames.

☞ Optimal page replacement algorithm of Belady can be applied
(replace page with largest forward distance) [Leupers]

# Optimising the use of *m* modify registers

## Given: sequence of modify values




[Leupers&Marwedel, ICCAD, 1996]

# Integrated memory allocation/ Modify register assignment

**Standard approach**

**Memory allocation**

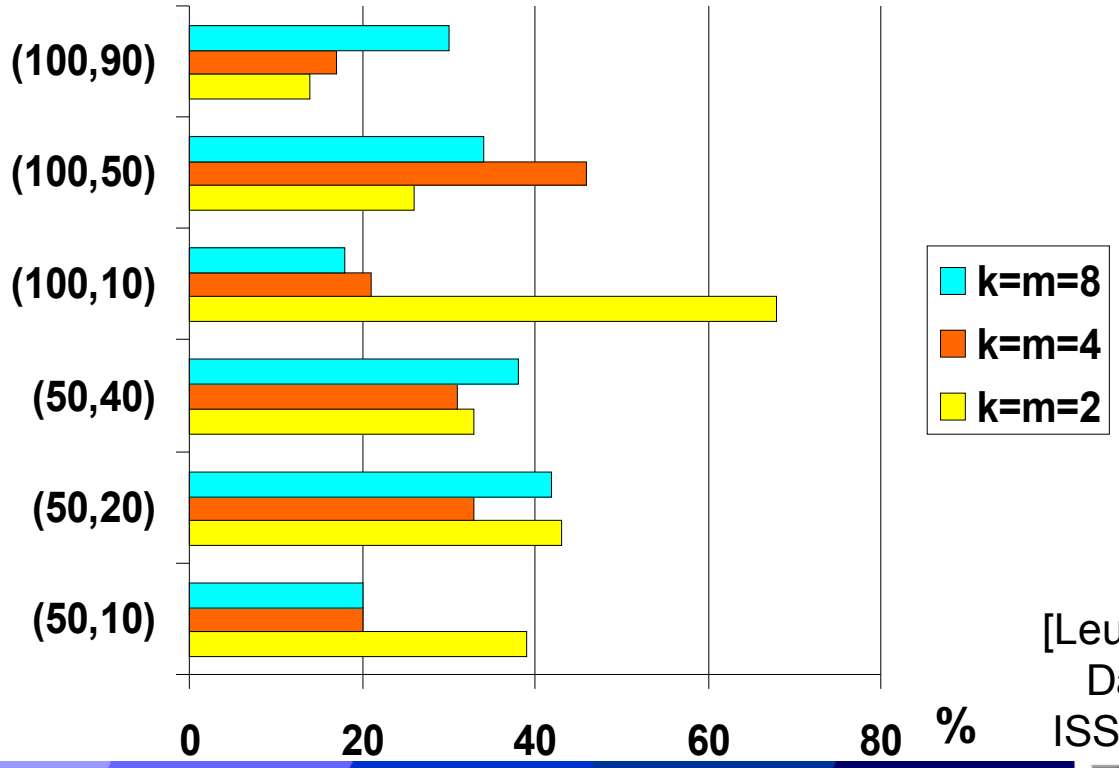
↓

**M register assignment**

**Simultaneous Optimisation using genetic algorithm:**

Percentage of saved address computation cycles

(Sequence length ,# Variables)




[Leupers, David; ISSS'98]

# Work on larger autoincrement/decrement ranges

Codesize [kB] as a function of the number of address registers ($k$) and the increment ranges ($r$), bold numbers indicate minimum

| Small program | $k$ | | | | |
|---|---|---|---|---|---|
| $r$ | 1 | 2 | 3 | 4 | 5 |
| 1 | 29.71 | **25.73** | 26.70 | **26.47** | 28.35 |
| 2 | 27.86 | 26.59 | 28.26 | 28.26 | 30.10 |
| 3 | **25.92** | **26.33** | 28.22 | 28.22 | 30.10 |
| 4 | 30.54 | 28.22 | 30.10 | 30.10 | 31.98 |
| 5 | 26.70 | 28.22 | 30.10 | 30.10 | 31.98 |

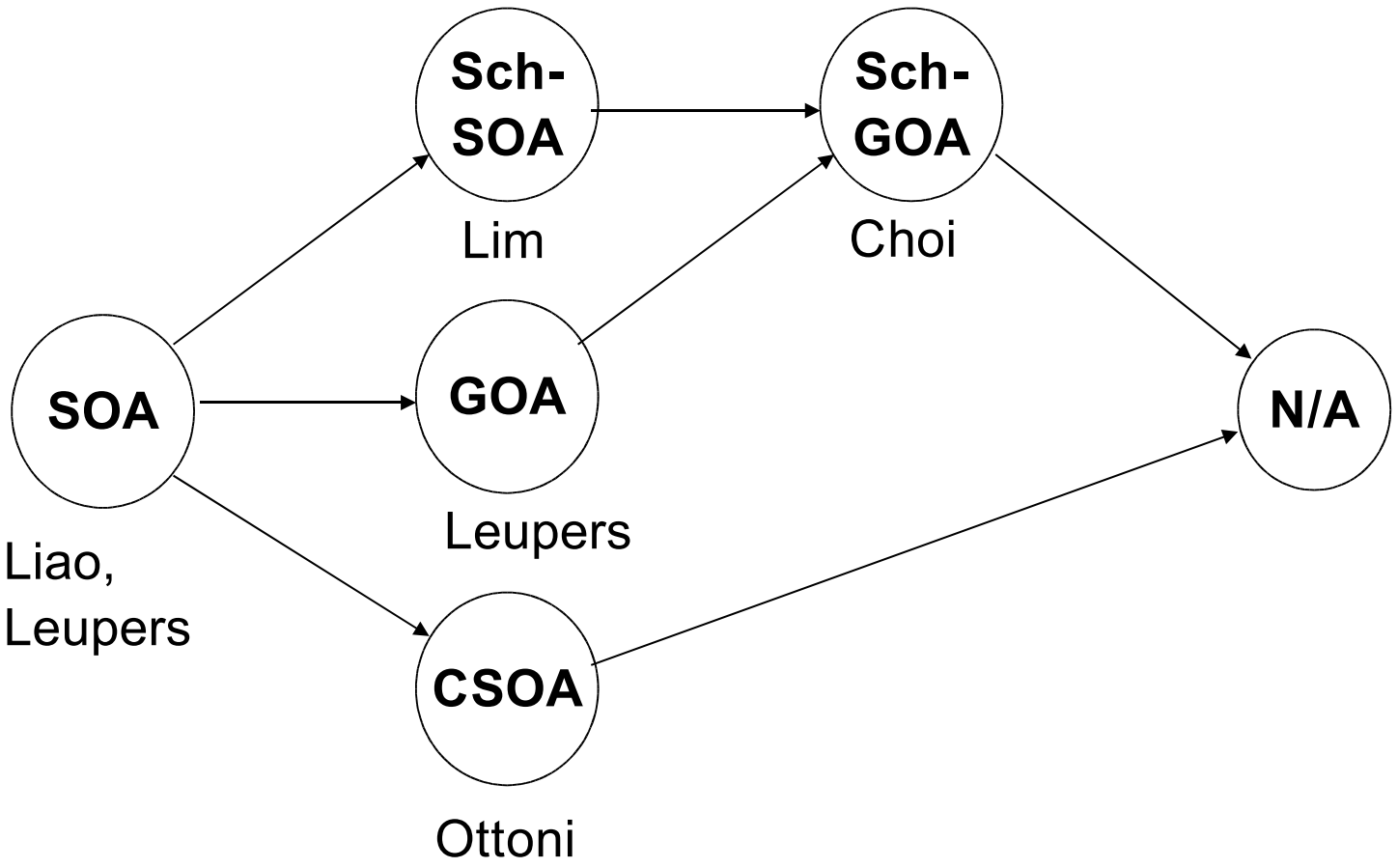
Medium-sized program

| | $k$ | | | | |
|---|---|---|---|---|---|
| $r$ | 1 | 2 | 3 | 4 | 5 |
| 1 | 123.5 | 110.3 | 118.7 | **103.0** | 107.2 |
| 2 | 122.9 | 107.0 | 107.1 | 104.9 | 111.4 |
| 3 | 116.4 | **101.2** | 104.7 | **104.4** | 111.3 |
| 4 | 118.6 | 105.8 | 111.4 | 111.3 | 118.2 |
| 5 | 114.4 | 104.7 | 111.3 | 111.3 | 118.2 |

($k$=3, $r$=1) and ($k$=2, $r$=3) are useful designs. Larger ranges do not reduce code size.

**Refererence:** Sudarsanam, Liao, Devadas: Analysis and Address Arithmetic Capabilities in Custom DSP Architectures, DAC, 1997, p. 287-292

# (Some) Generalizations of SOA

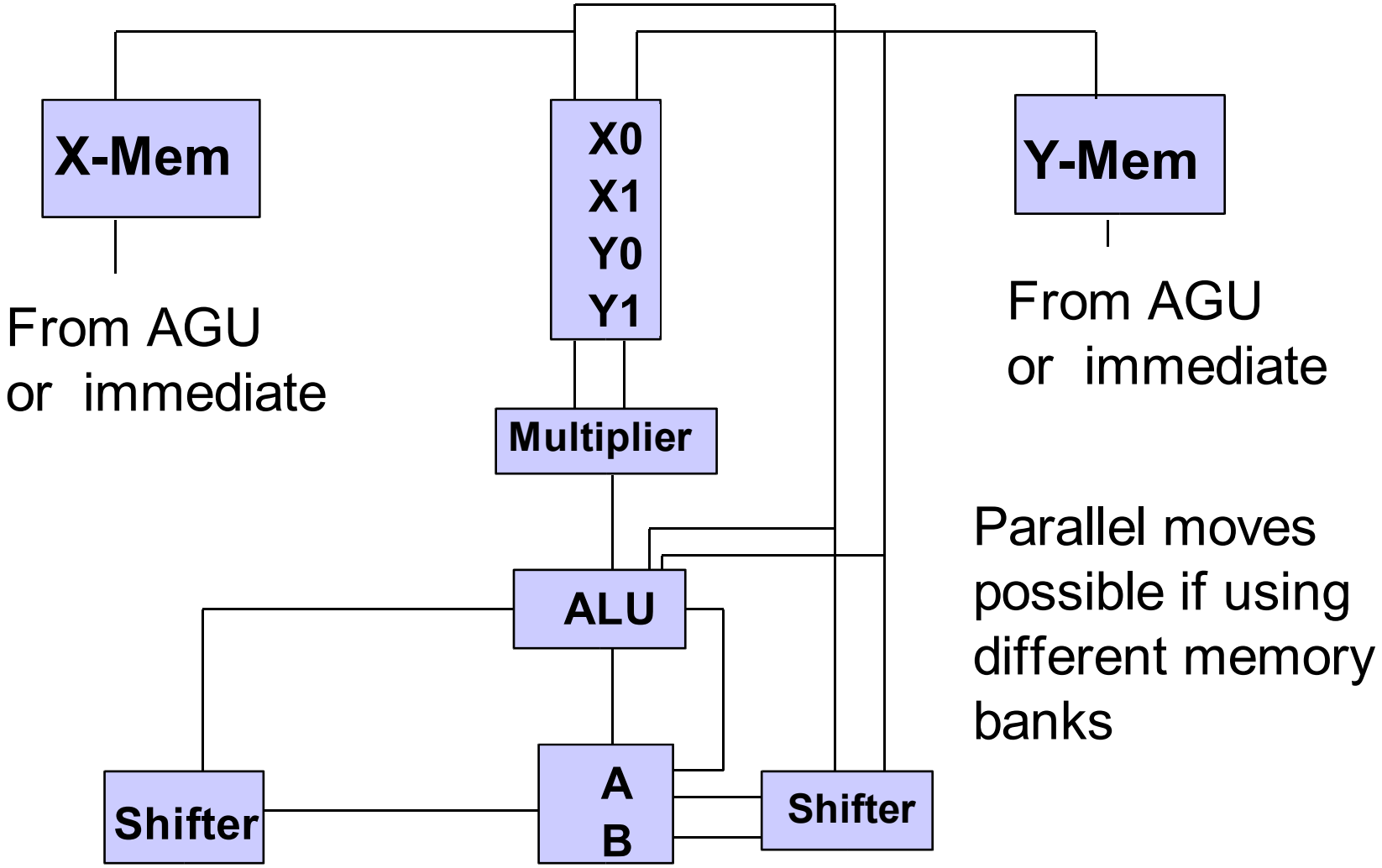


# Comparison of different approaches for (*k*,*m*,*r*)-OA

| Author(s) | Source | Problem | | | | | | Approach |
|---|---|---|---|---|---|---|---|---|
| | | (*k*,*m*,*r*) | **APA** | **OA** | A | S | M | |
| Bartley | | | + | + | - | - | - | |
| Liao | | (1,0,1) | + | + | - | - | - | |
| Leupers @ | ICCAD 96 | (*k*,0,1) | + | + | - | - | - | Hamilton path + tie break |
| Gebotys | ICCAD 97 | (*k*,0,*r*) | + | - | + | - | - | Network flow |
| Sudarsanam | DAC 97 | (*k*,0,*r*) | + | + | | | | Effect of *r* on code size |
| Basu @ | VLSI 97 | (*k*,0,1) | + | - | + | - | - | Focus on arrays in loops |
| Leupers @ | ISSS 98 | (*k*,*m*,1) | + | + | - | - | - | Genetic algorithm |
| Lim | CODES 2001 | (1,0,1) | + | + | - | + | - | VAG edges minimized during scheduling |
| Choi, Kim | DAC 2002 | (*k*,0,1) | + | + | - | + | - | |
| Ottoni ++ | SCOPES 2003 | (1,0,1) | + | + | - | - | - | Variable coalescing |
| Wess | SCOPES 2004 | (*k*,0,*r*) | + | + | + | - | + | Iterated APA+GOA cycles |

*k*= address regs.; *m*=modif. regs.; *r*: offset range; APA: address pointer ass.; OA: offset ass.; A: access based (+), partitioned variables (-); S: integrated scheduling (+); M+: modulo addressing; @: at U. Dortm

# Multiple memory banks
# - Sample hardware -

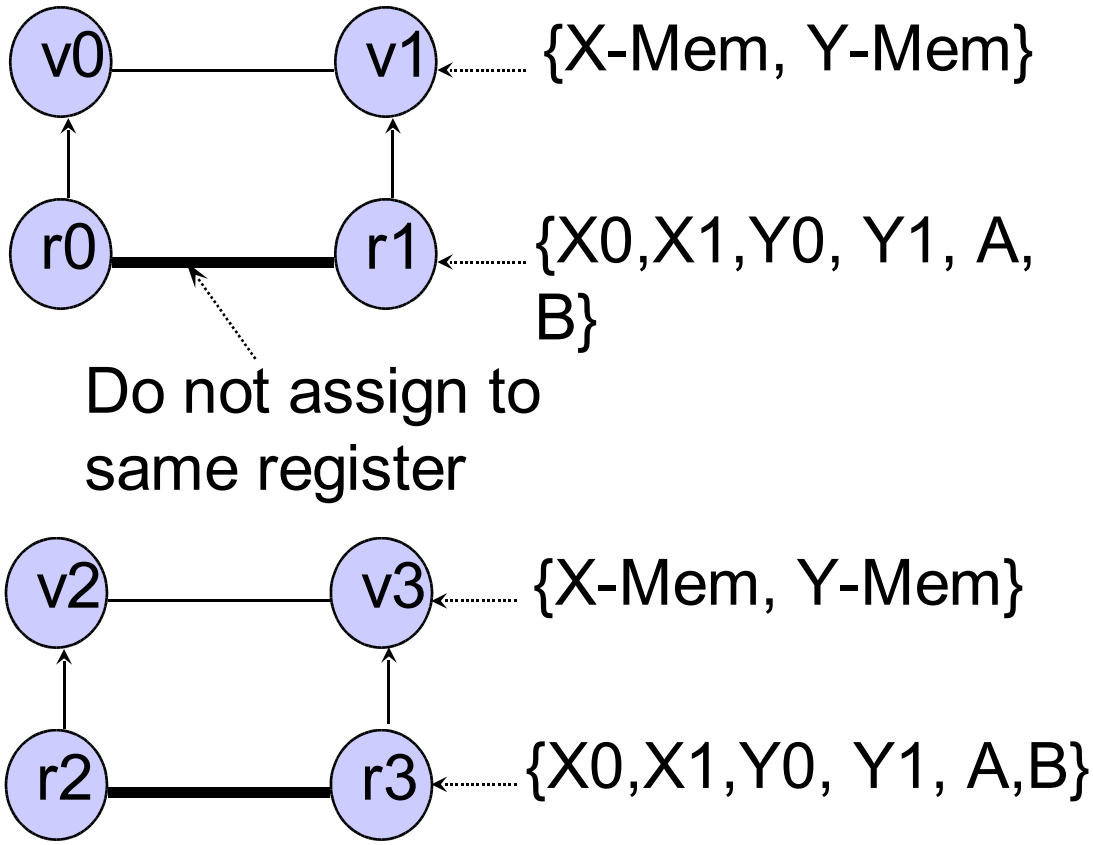


# Multiple memory banks
# - Constraint graph generation -

Precompacted code (symbolic variables and registers)

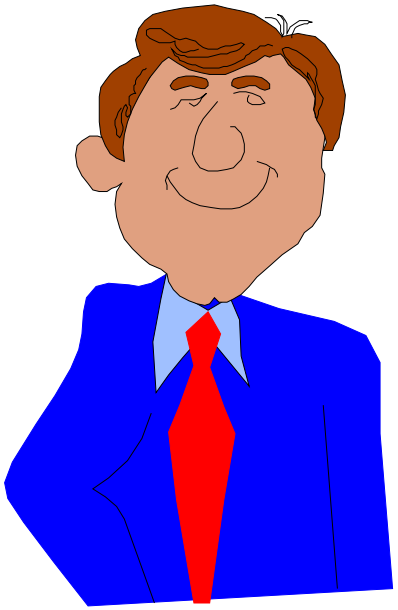
**Move v0,r0  v1,r1**
**Move v2,r2  v3,r3**

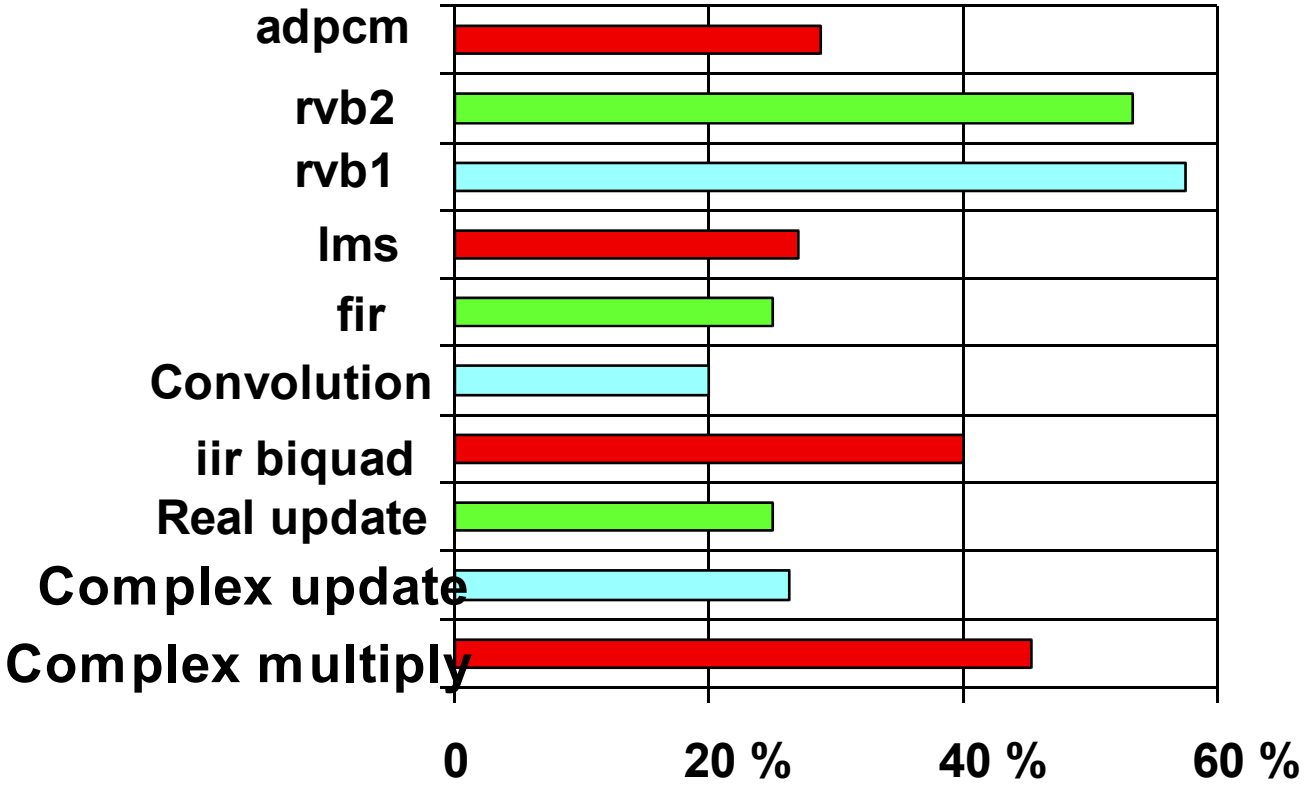
**Constraint graph**




Links maintained, more constraints ...

# Multiple memory banks
# Code size reduction through simulated annealing






[Sudarsanam, Malik, 1995]

# Summary

Exploiting some of the special features of embedded processors in compilers

- autoincrement- and decrement modes
  - address pointer assignment problem (APA)
  - simple offset assignment problem (SOA)
  - SOA with variable coalescing
  - integrated scheduling
  - general offset assignment problem
  - use of modify registers
- multiple memory banks