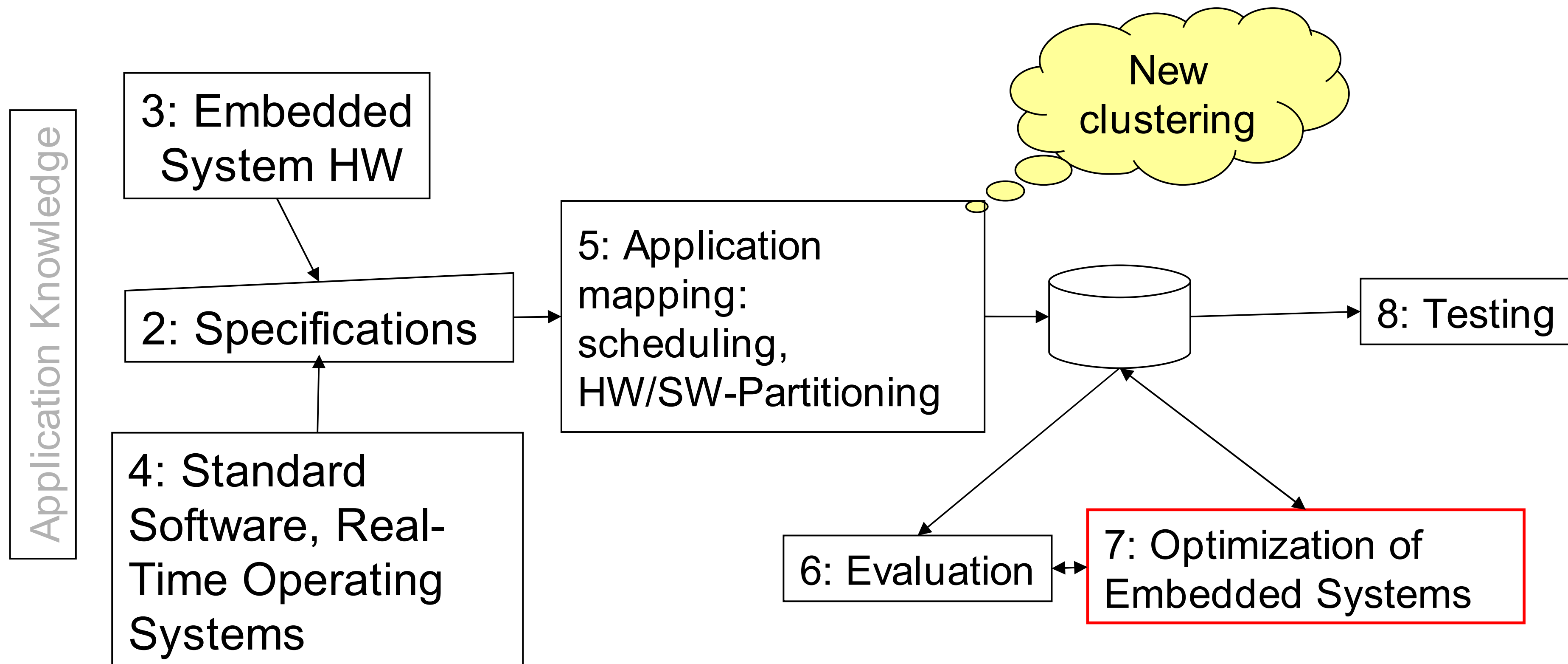
Optimizations

Peter Marwedel
TU Dortmund, Informatik 12
Germany

2008/06/25

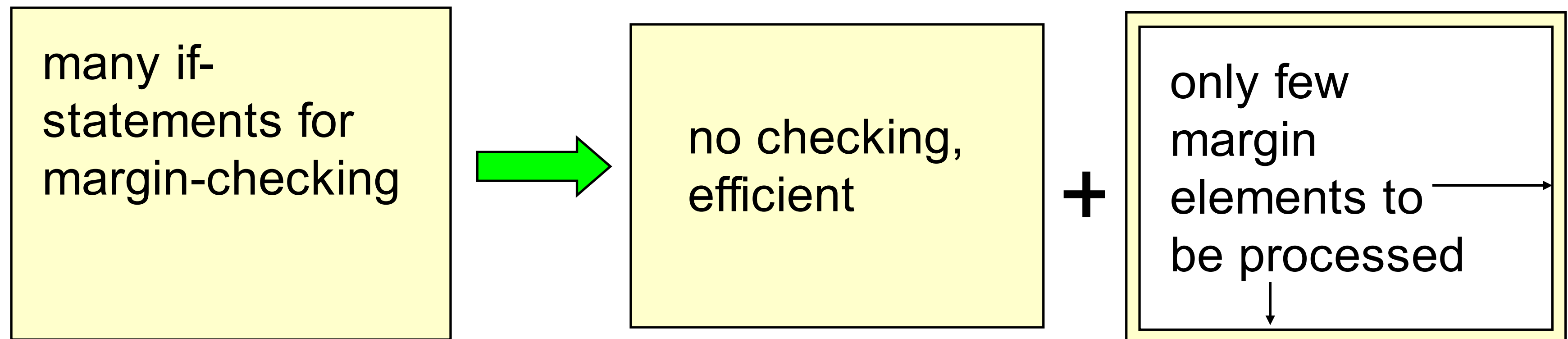
Structure of this course



Chapter 5.2.4

Transformation "Loop nest splitting"

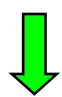
Example: Separation of margin handling



Loop nest splitting at University of Dortmund

Loop nest from MPEG-4 full search motion estimation

```
for (z=0; z<20; z++)
 for (x=0; x<36; x++) {x1=4*x;
  for (y=0; y<49; y++) {y1=4*y;
   for (k=0; k<9; k++) {x2=x1+k-4;
    for (l=0; l<9; ) {y2=y1+l-4;
     for (i=0; i<4; i++) {x3=x1+i; x4=x2+i;
      for (j=0; j<4;j++) {y3=y1+j; y4=y2+j;
       if (x3<0 || 35<x3||y3<0||48<y3)
        then_block_1; else else_block_1;
       if (x4<0|| 35<x4||y4<0||48<y4)
        then_block_2; else else_block_2;
}}}}}}
```



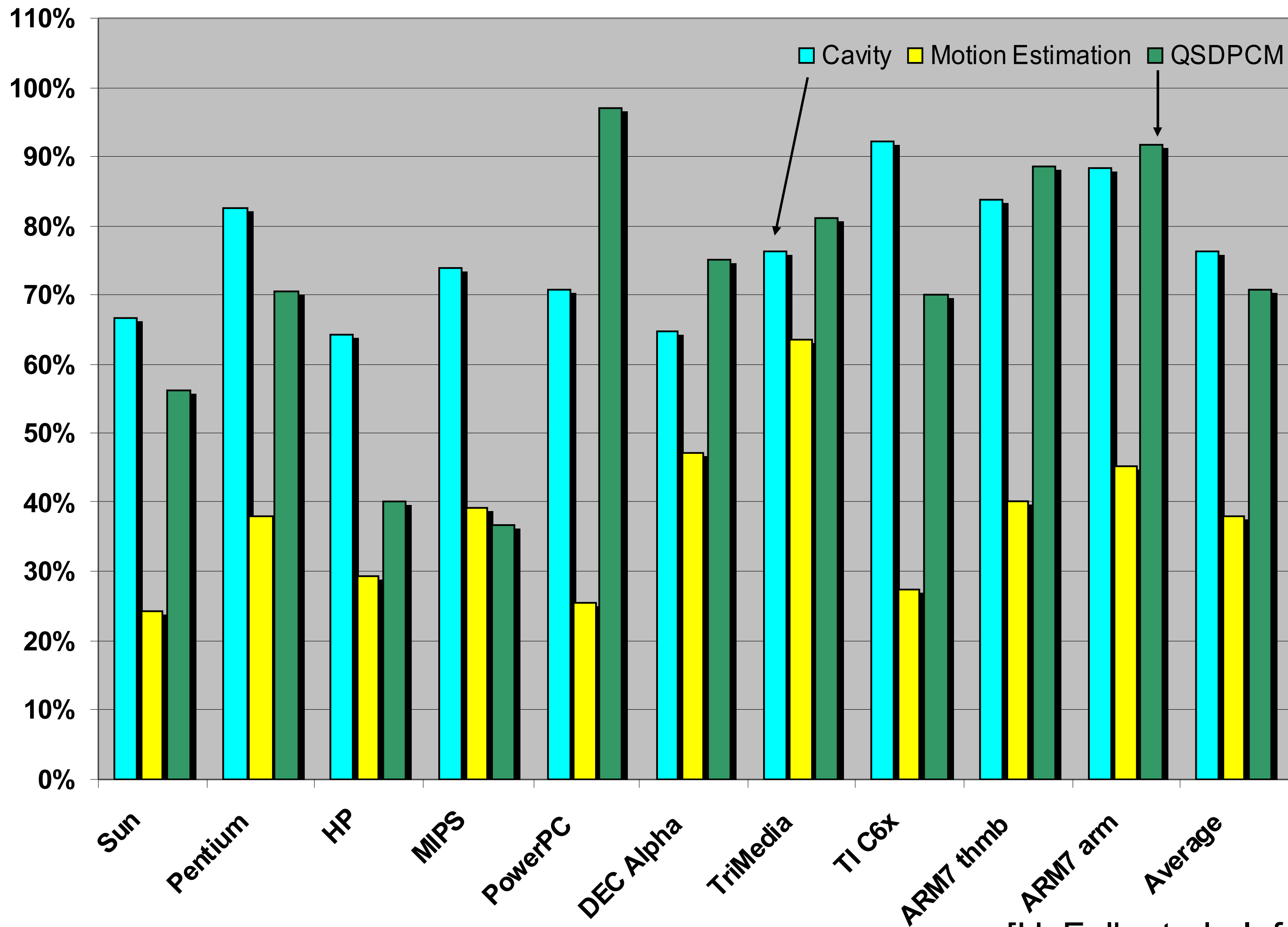
analysis of polyhedral domains, selection with genetic algorithm

```
for (z=0; z<20; z++)
 for (x=0; x<36; x++) {x1=4*x;
  for (y=0; y<49; y++)
```

```
if (x>=10||y>=14)
 for (; y<49; y++)
  for (k=0; k<9; k++)
   for (l=0; l<9;l++ )
    for (i=0; i<4; i++)
     for (j=0; j<4;j++) {
      then_block_1; then_block_2}
else {y1=4*y;
 for (k=0; k<9; k++) {x2=x1+k-4;
  for (l=0; l<9; ) {y2=y1+l-4;
   for (i=0; i<4; i++) {x3=x1+i; x4=x2+i;
    for (j=0; j<4;j++) {y3=y1+j; y4=y2+j;
     if (0 || 35<x3 ||0 || 48<y3)
      then-block-1; else else-block-1;
     if (x4<0|| 35<x4||y4<0||48<y4)
      then_block_2; else else_block_2;
}}}}}}
```

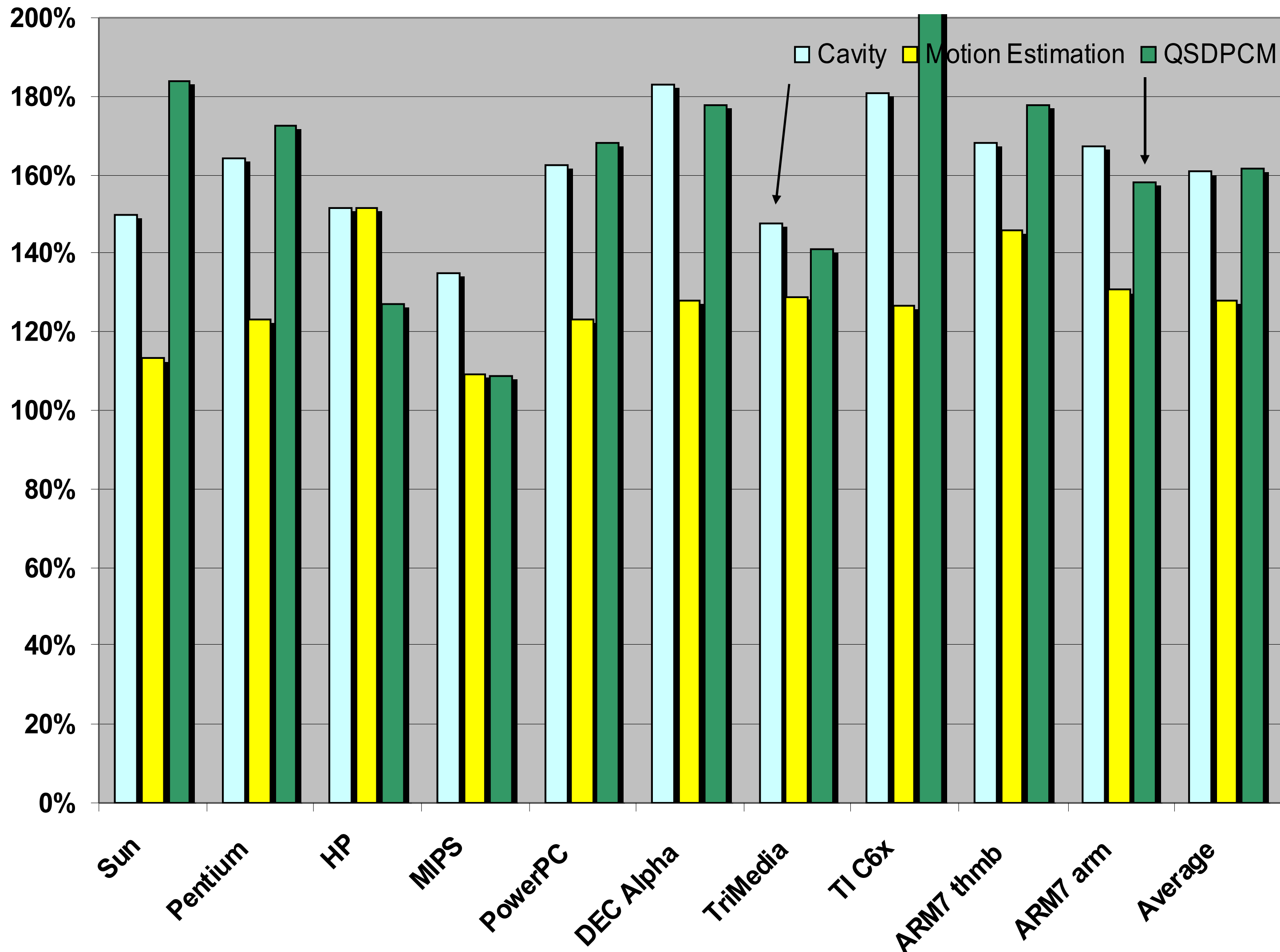
[H. Falk et al., Inf 12, UniDo, 2002]

Results for loop nest splitting - Execution times -



[H. Falk et al., Inf 12, UniDo, 2002]

Results for loop nest splitting - Code sizes -

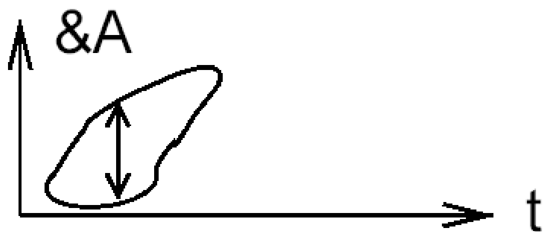


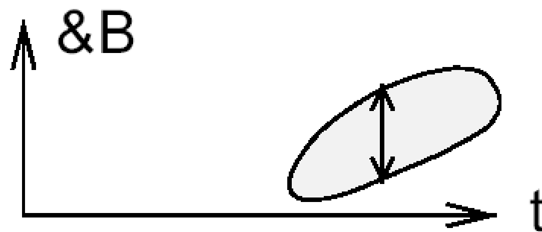
[Falk, 2002]

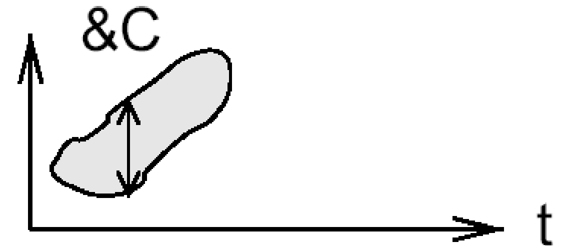
Chapter 5.2.5

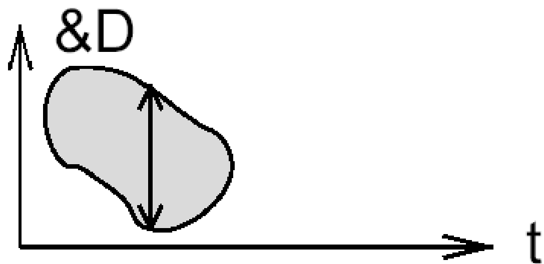
Array folding

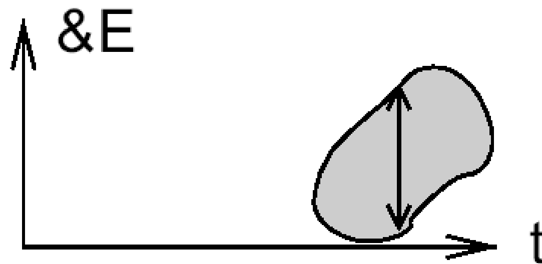
Initial arrays





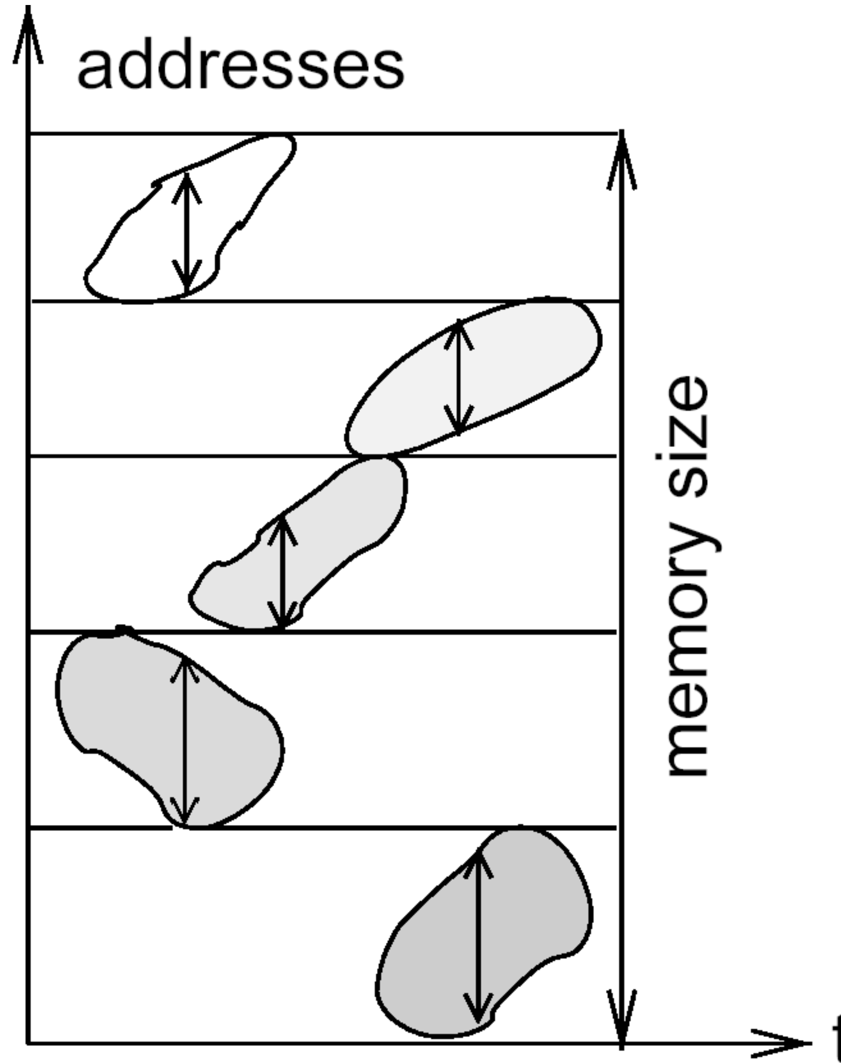






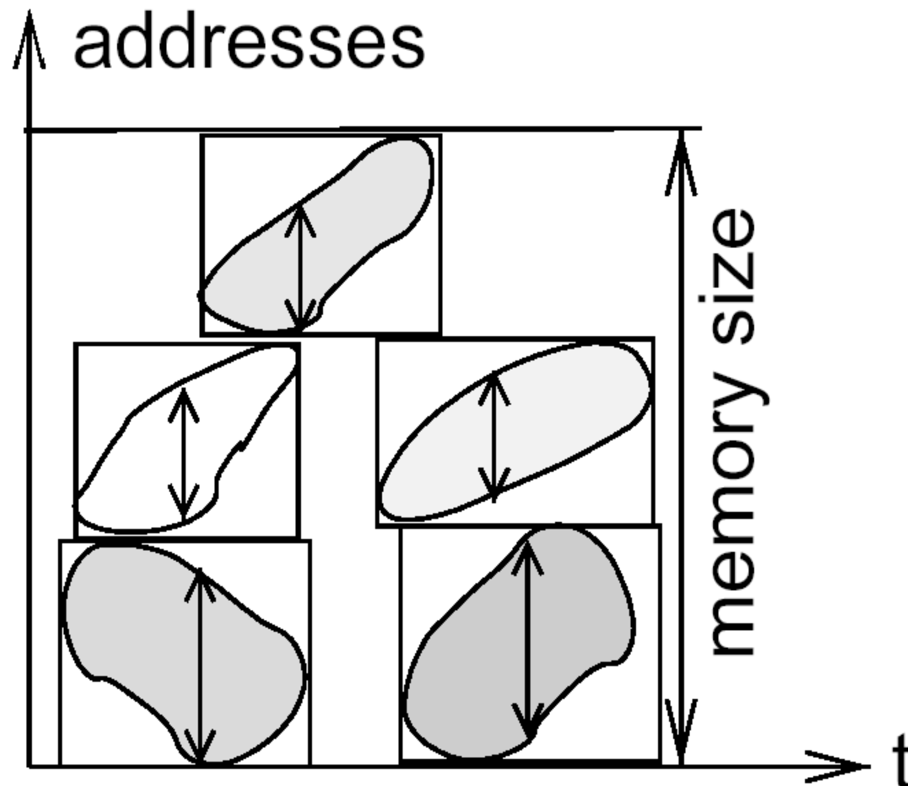
Array folding

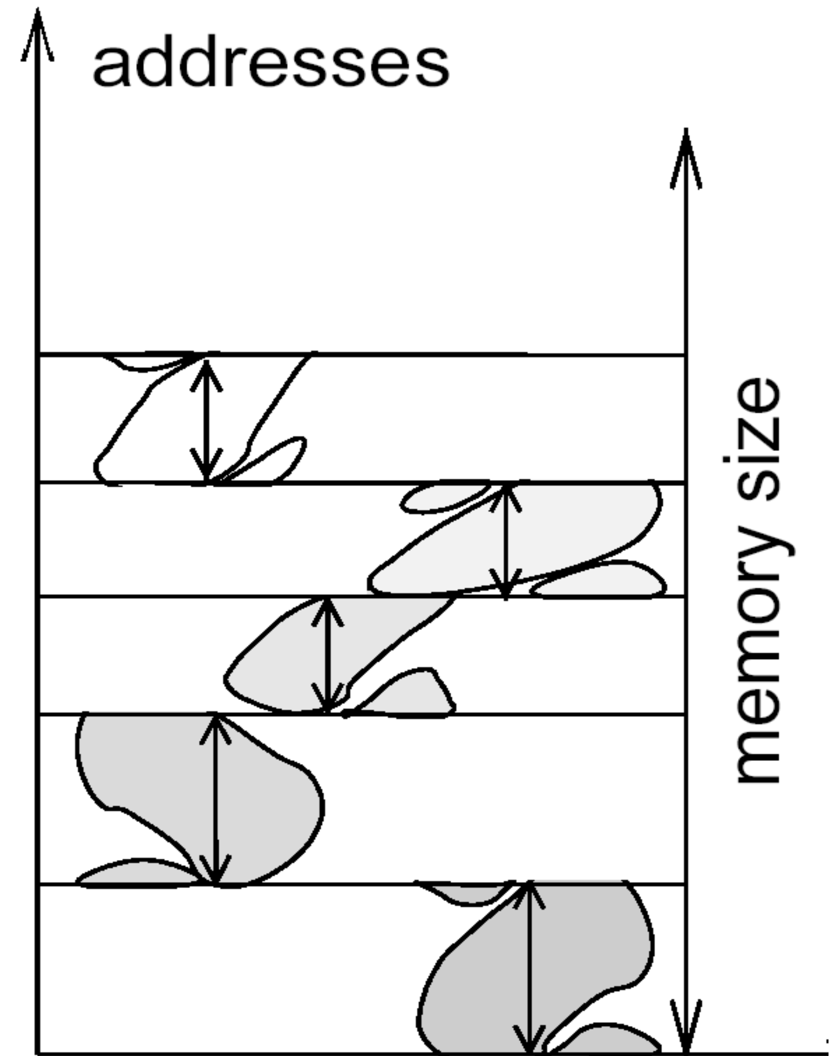
Unfolded arrays



Intra-array folding

Inter-array folding





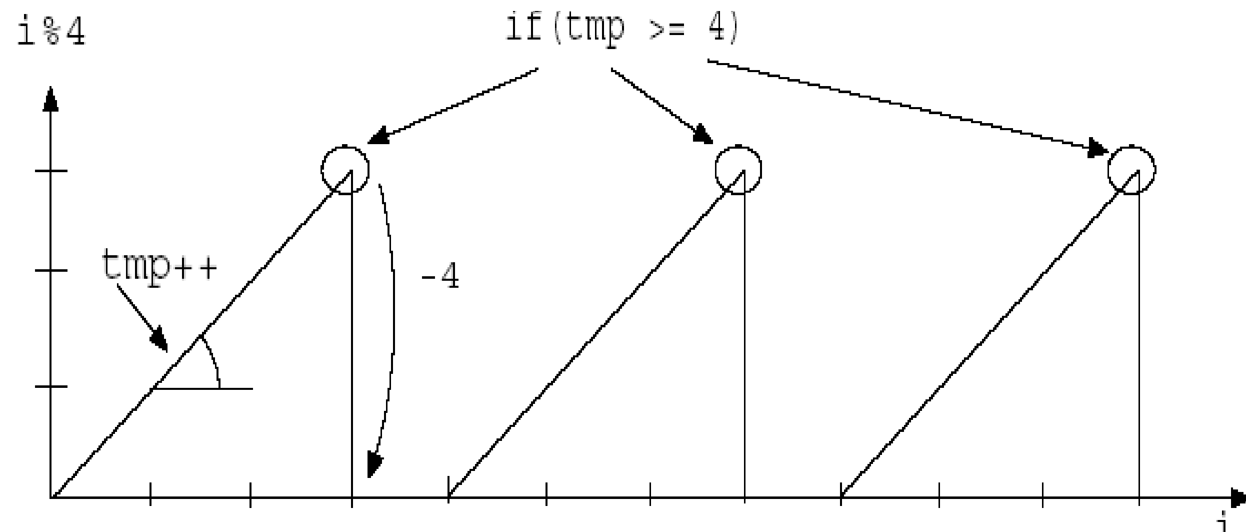
© p. marwedel, informatik 12, 2008

Application

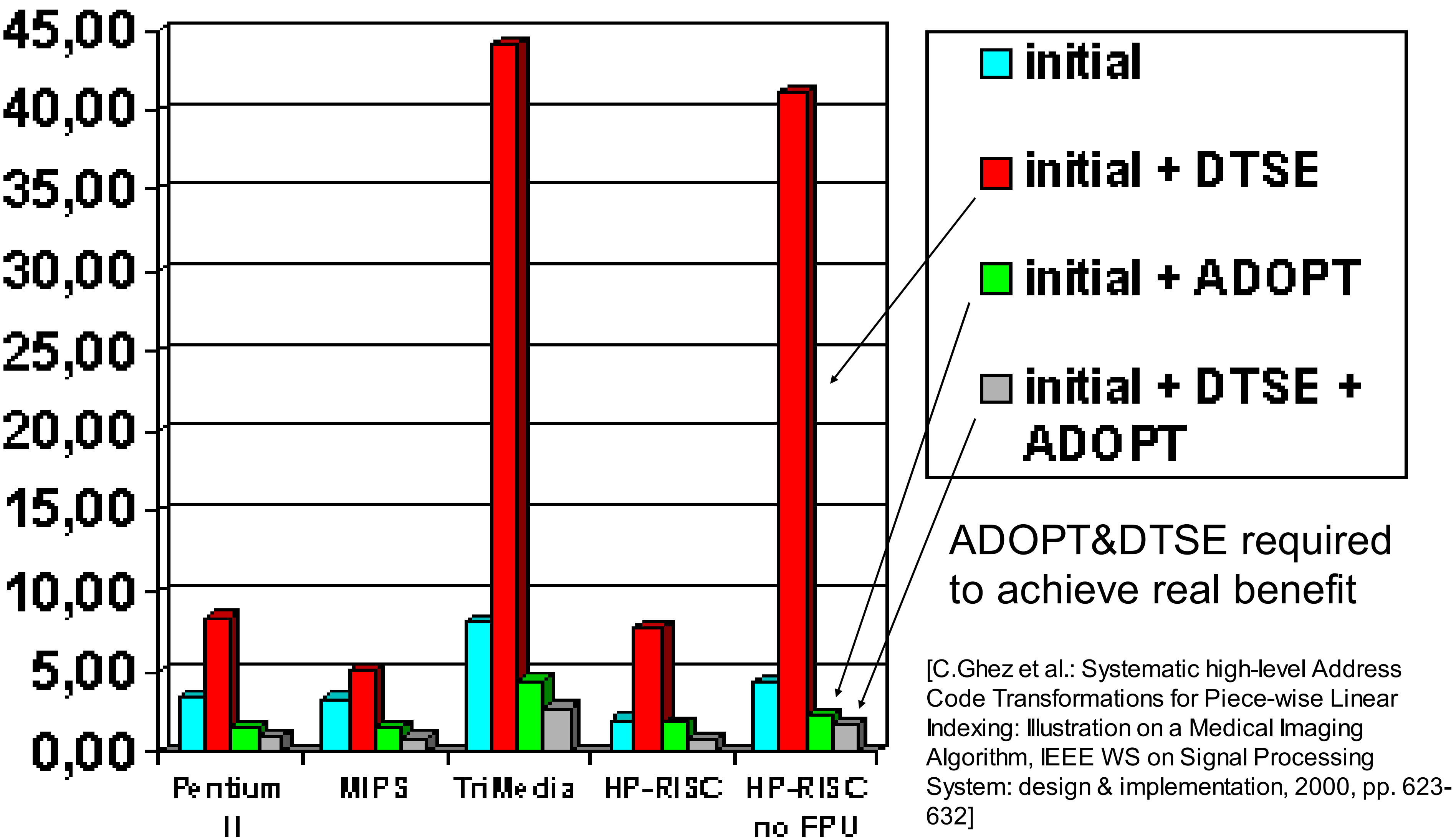
- Array folding is implemented in the DTSE optimization proposed by IMEC. Array folding adds div and mod ops. Optimizations required to remove these costly operations.
- At IMEC, ADOPT address optimizations perform this task. For example, modulo operations are replaced by pointers (indexes) which are incremented and reset.

```
for(i=0; i<20; i++)
      B[i % 4];

tmp=0;
for(i=0; i<20; i++)
      if(tmp >= 4)
         tmp -=4;
      B[tmp];
      tmp ++;
```



Results (Mcycles for cavity benchmark)

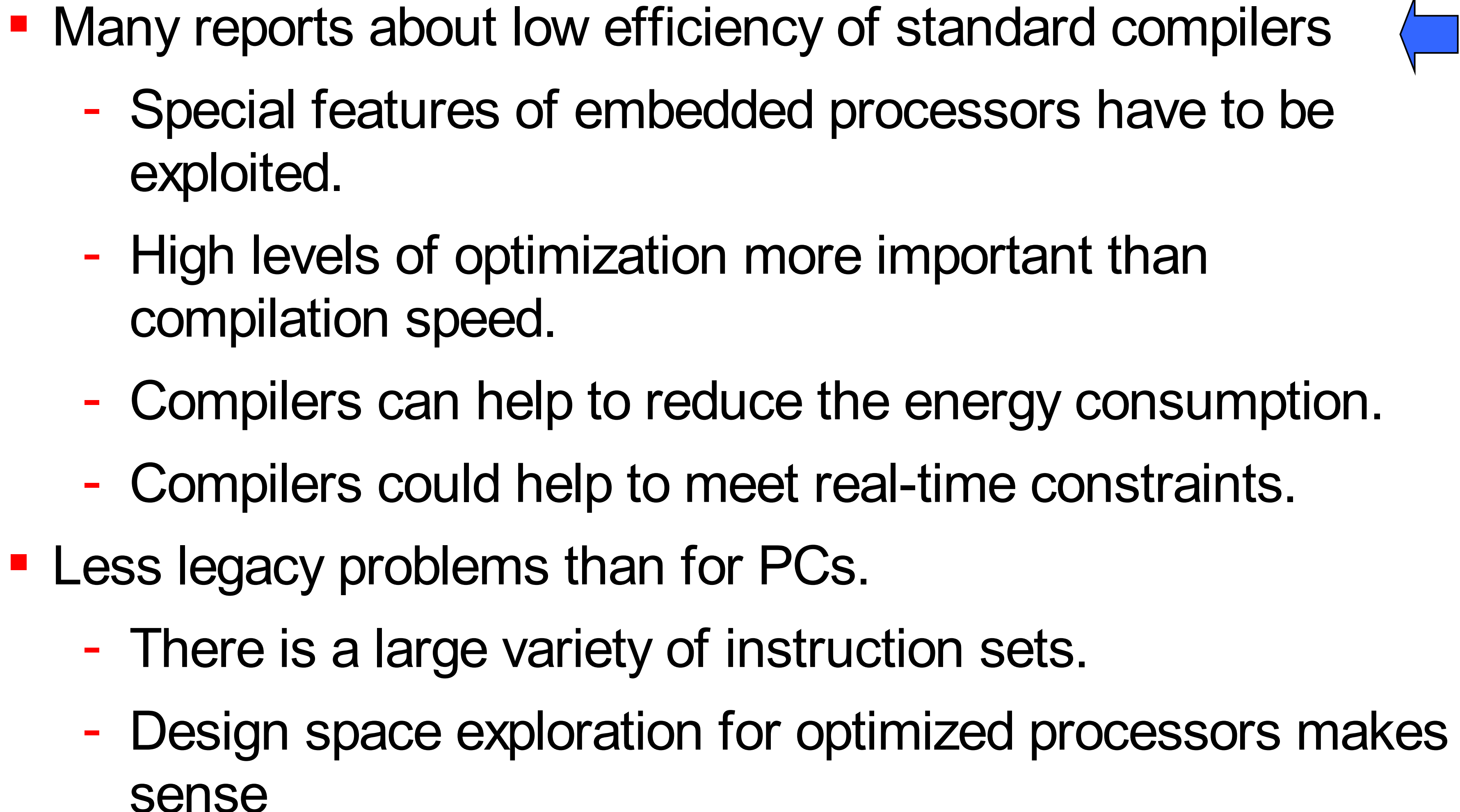


Chapter 5.4

Compilation for Embedded Processors

Peter Marwedel
TU Dortmund, Informatik 12
Germany

Compilers for embedded systems: Why are compilers an issue?

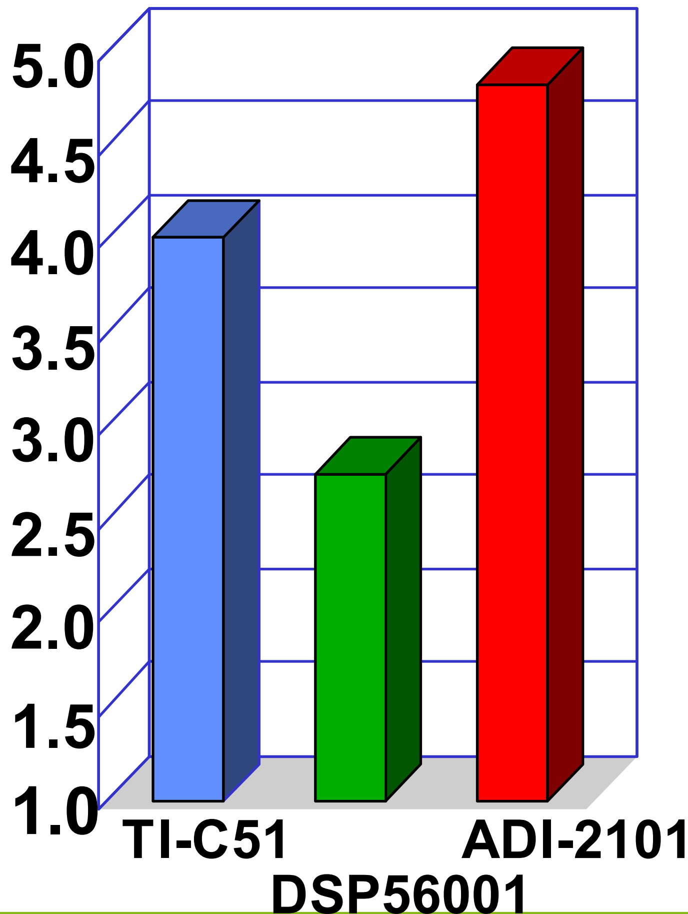


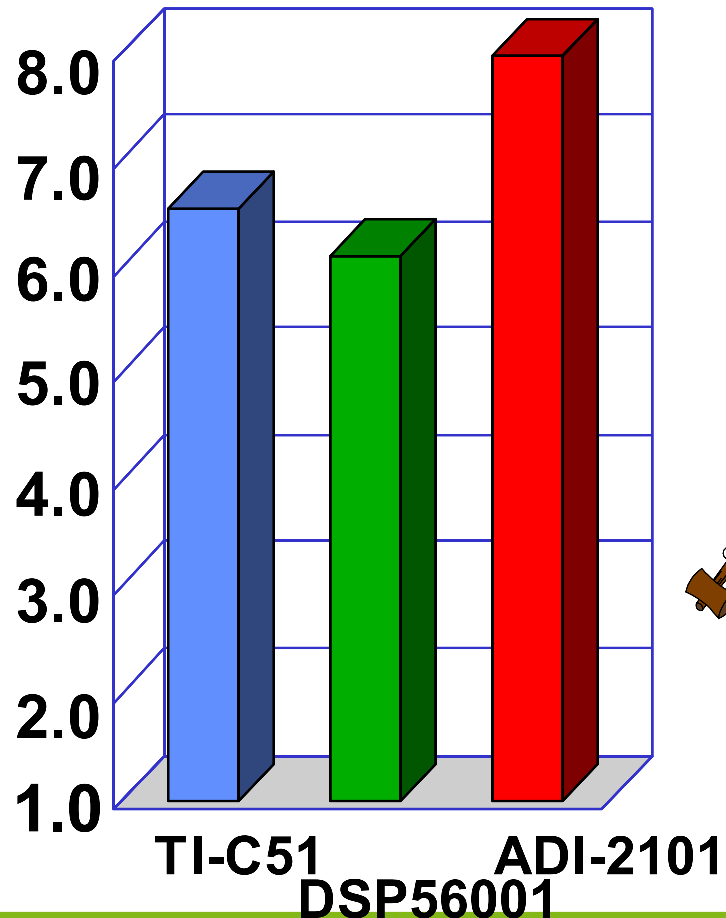
- Many reports about low efficiency of standard compilers
  - Special features of embedded processors have to be exploited.
  - High levels of optimization more important than compilation speed.
  - Compilers can help to reduce the energy consumption.
  - Compilers could help to meet real-time constraints.
- Less legacy problems than for PCs.
  - There is a large variety of instruction sets.
  - Design space exploration for optimized processors makes sense

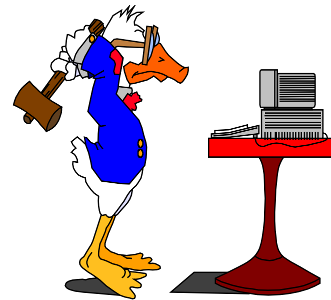
(Lack of) performance of C-compilers for DSPs

DSPStone (Zivojnovic et al.).

Example: ADPCM

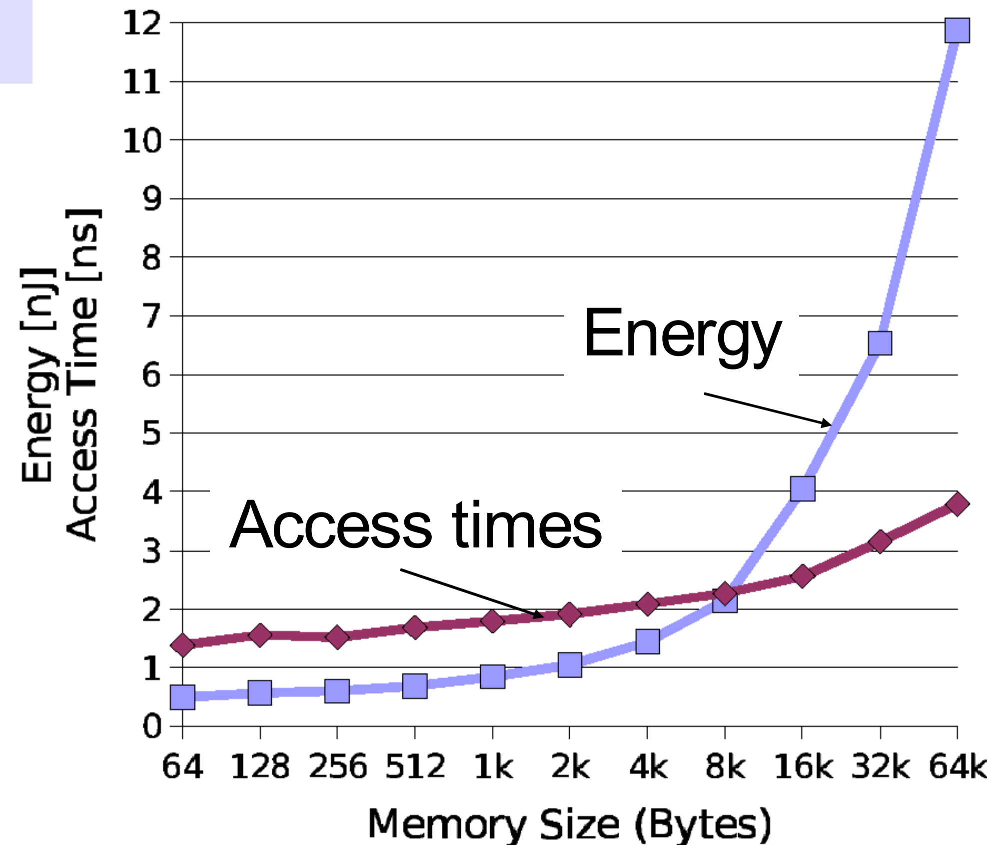






3 key problems for future memory systems

1. (Average) Speed
2. Energy/Power
3. Predictability/WCET



Optimization for low-energy the same as optimization for high performance?

No !

- High-performance if available memory bandwidth fully used; low-energy consumption if memories are at stand-by mode
- Reduced energy if more values are kept in registers

```
int a[1000];
c = a;
for (i = 1; i < 100; i++) {
 b += *c;
 b += *(c+7);
 c += 1;
}
```

```
LDR r3, [r2, #0]
ADD r3,r0,r3
MOV r0,#28
LDR r0, [r2, r0]
ADD r0,r3,r0
ADD r2,r2,#4
ADD r1,r1,#1
CMP r1,#100
BLT LL3
```

2096 cycles
19.92 µJ

```
ADD r3,r0,r2
MOV r0,#28
MOV r2,r12
MOV r12,r11
MOV r11,rr10
MOV r0,r9
MOV r9,r8
MOV r8,r1
LDR r1, [r4, r0]
ADD r0,r3,r1
ADD r4,r4,#4
ADD r5,r5,#1
CMP r5,#100
BLT LL3
```

2231 cycles
16.47 µJ

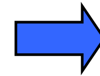
© p. marwedel, informatik 12, 2008

Compiler optimizations for improving energy efficiency

- Energy-aware scheduling
- Energy-aware instruction selection
- Operator strength reduction: e.g. replace * by + and <<
- Minimize the bitwidth of loads and stores
- Standard compiler optimizations with energy as a cost function

E.g.: Register pipelining:

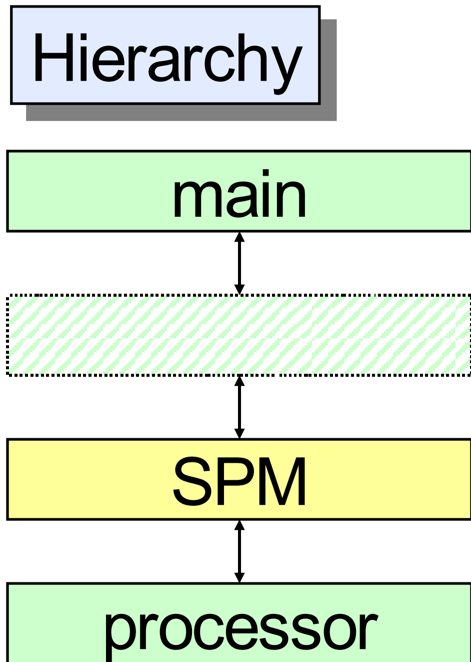
```
for i:= 0 to 10 do
   C:= 2 * a[i] + a[i-1];
```



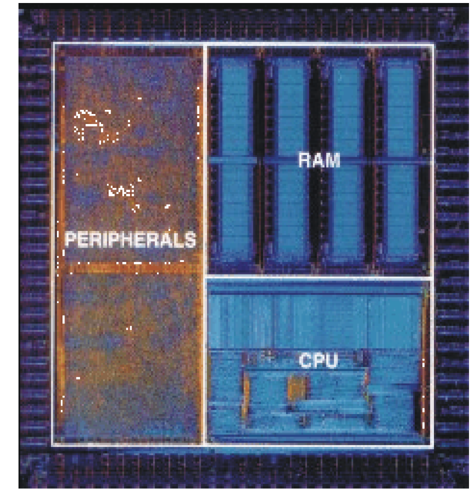
```
R2:=a[0];
for i:= 1 to 10 do
 begin
   R1:= a[i];
   C:= 2 * R1 + R2;
   R2 := R1;
 end;
```

Exploitation of the memory hierarchy

Hierarchical memories using scratch pad memories (SPM)

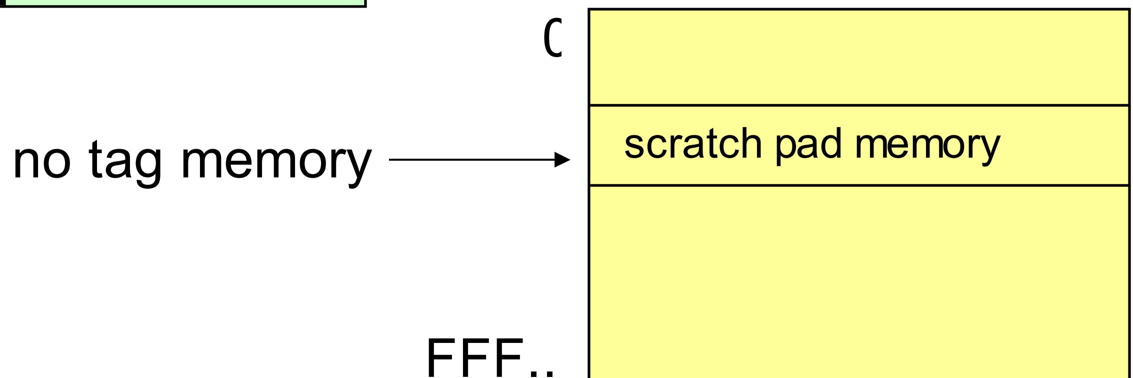


Example



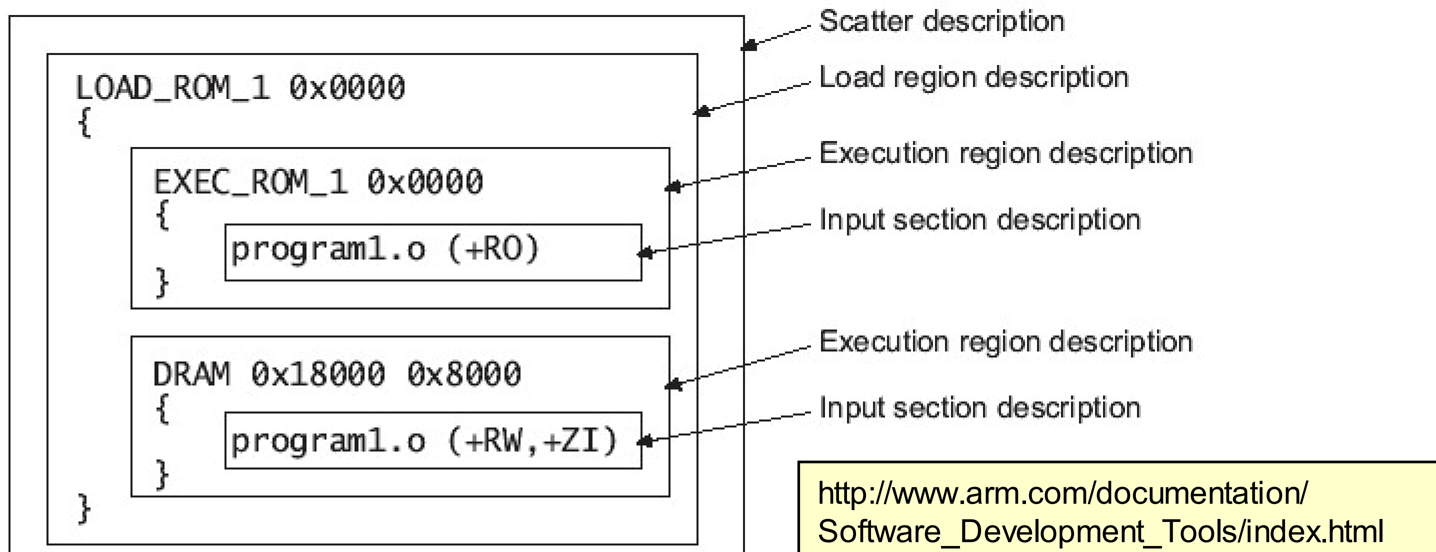
ARM7TDMI cores, well-known for low power consumption

Address space



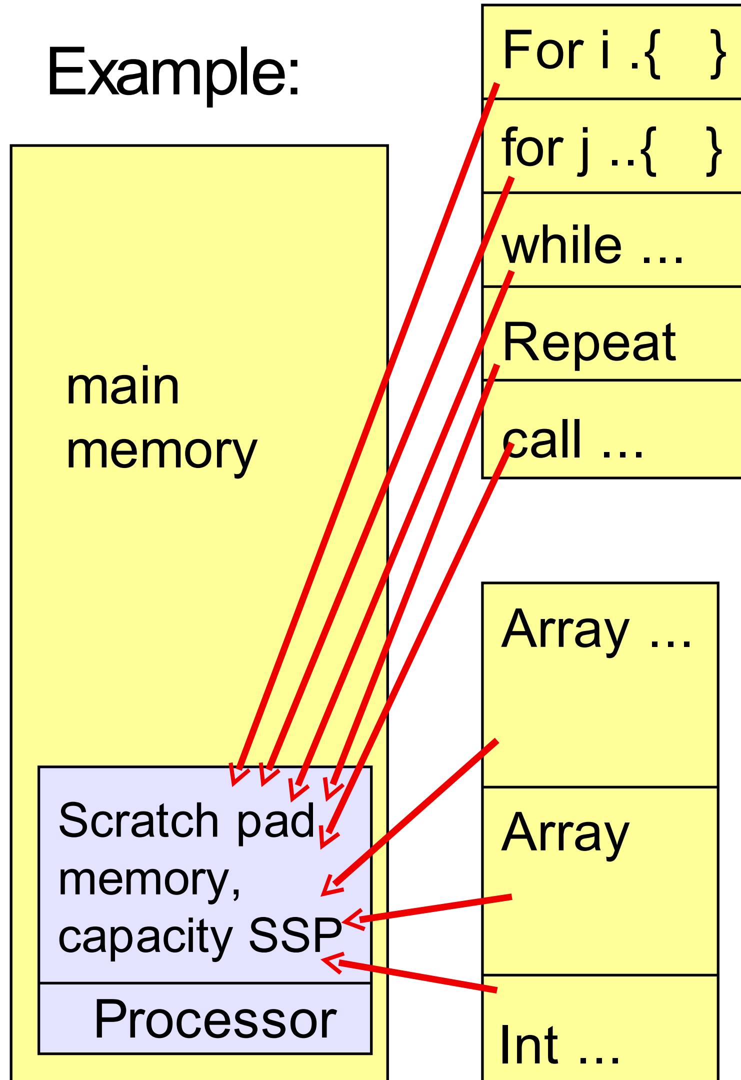
Very limited support in ARMcc-based tool flows

1. **Use pragma in C-source to allocate to specific section:**
   For example:
```
#pragma arm section rwdata = "foo", rodata = "bar"
int x2 = 5; // in foo (data part of region)
int const z2[3] = {1,2,3}; // in bar
```
2. **Input scatter loading file to linker for allocating section to specific address range**



http://www.arm.com/documentation/Software_Development_Tools/index.html

Migration of data and instructions, global optimization model (U. Dortmund)



Which memory object (array, loop, etc.) to be stored in SPM?

Non-overlaying ("Static") allocation:

Gain g_k and size s_k for each segment k. Maximise gain $G = \Sigma g_k$, respecting size of SPM SSP $\geq \Sigma\, s_k$.

Solution: knapsack algorithm.

Overlaying ("dynamic") allocation:

Moving objects back and forth

© p. marwedel, informatik 12, 2008

IP representation
- migrating functions and variables-

Symbols:

$S(var_k)$ = size of variable k

n_k = number of accesses to variable k

$e(var_k)$ = energy **saved** per variable access, if var_k is migrated

$E(var_k)$ = energy **saved** if variable var_k is migrated (= $e(var_k)\ n(var_k)$)

$x(var_k)$ = decision variable, =1 if variable k is migrated to SPM,
=0 otherwise

K = set of variables

Similar for functions I

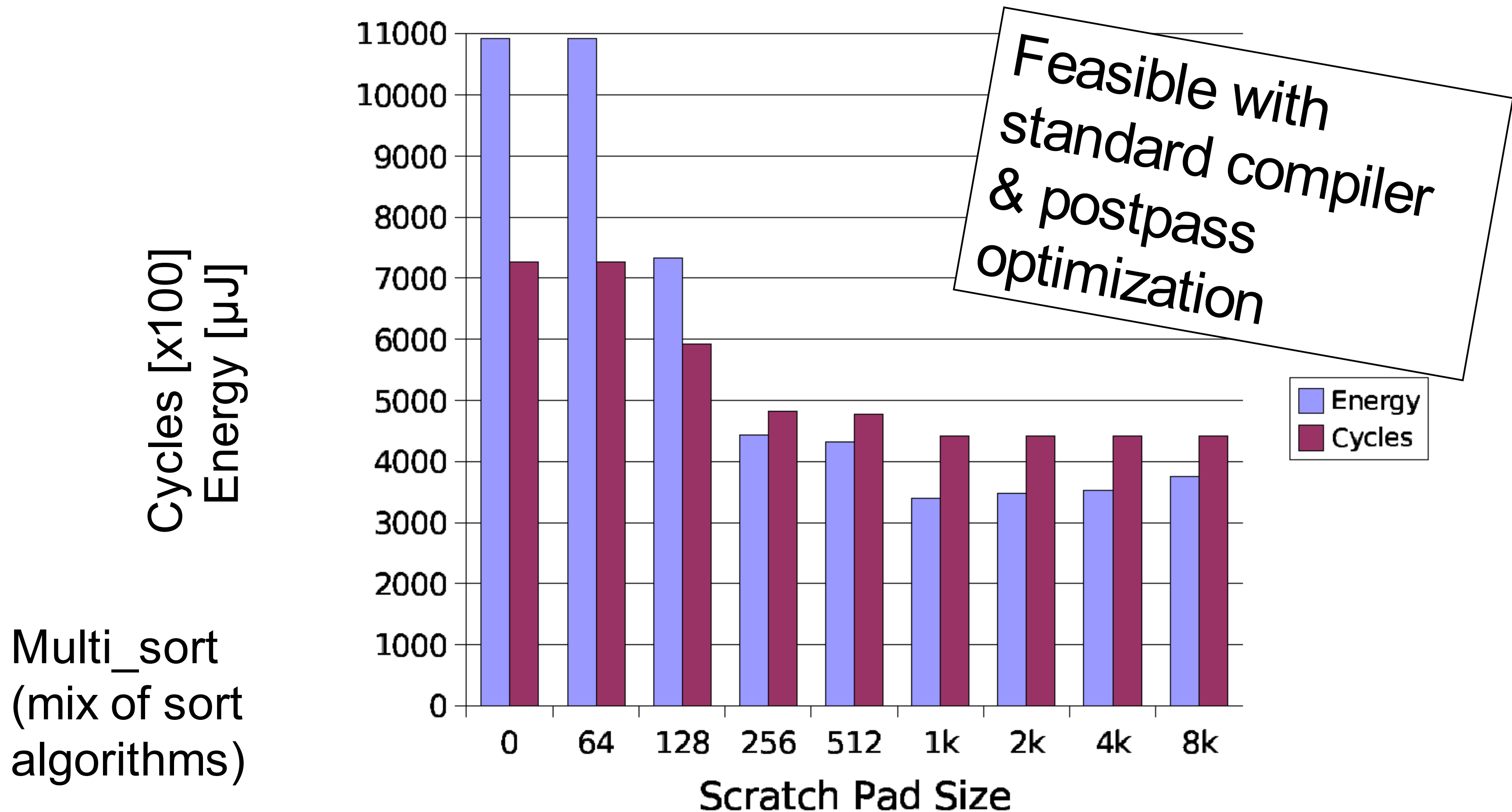
Integer programming formulation:

Maximize $\sum_{k \in K} x(var_k)\ E(var_k) + \sum_{i \in I} x(F_i)\ E(F_i)$

Subject to the constraint

$\sum_{k \in K} S\ (var_k)\ x(var_k) + \sum_{i \in I} S\ (F_i)\ x(F_i) \leq SSP$

Reduction in energy and average run-time



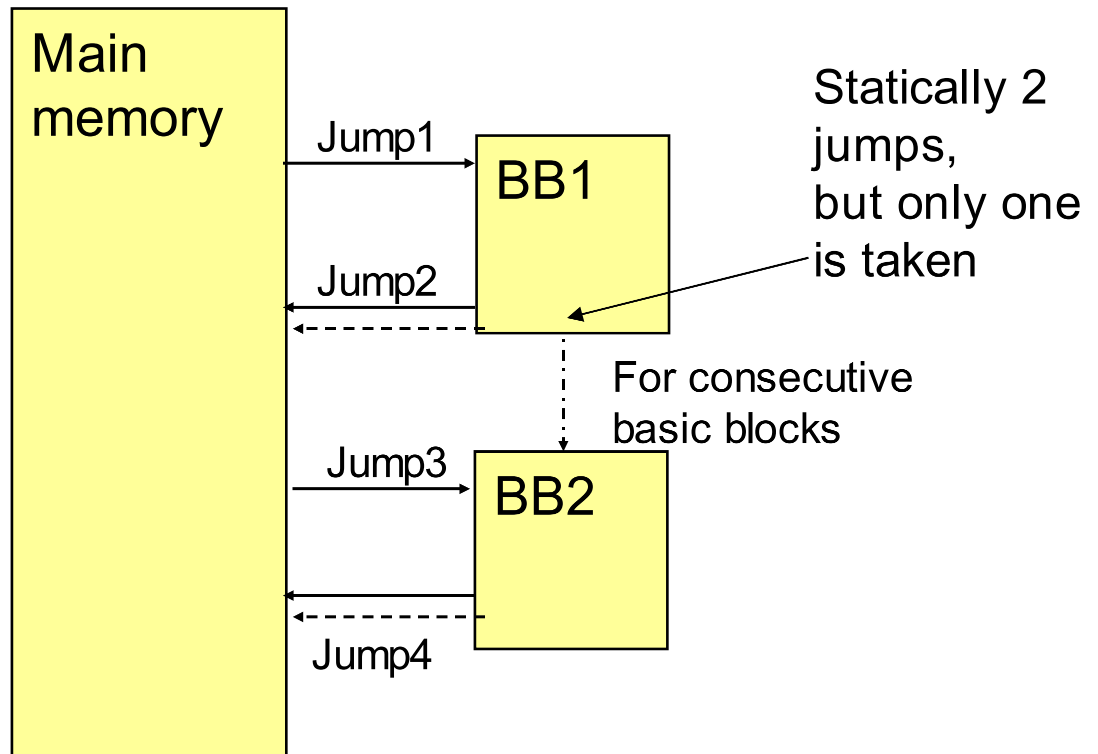
Measured processor / external memory energy + CACTI values for SPM (combined model)

Numbers will change with technology, algorithms remain unchanged.

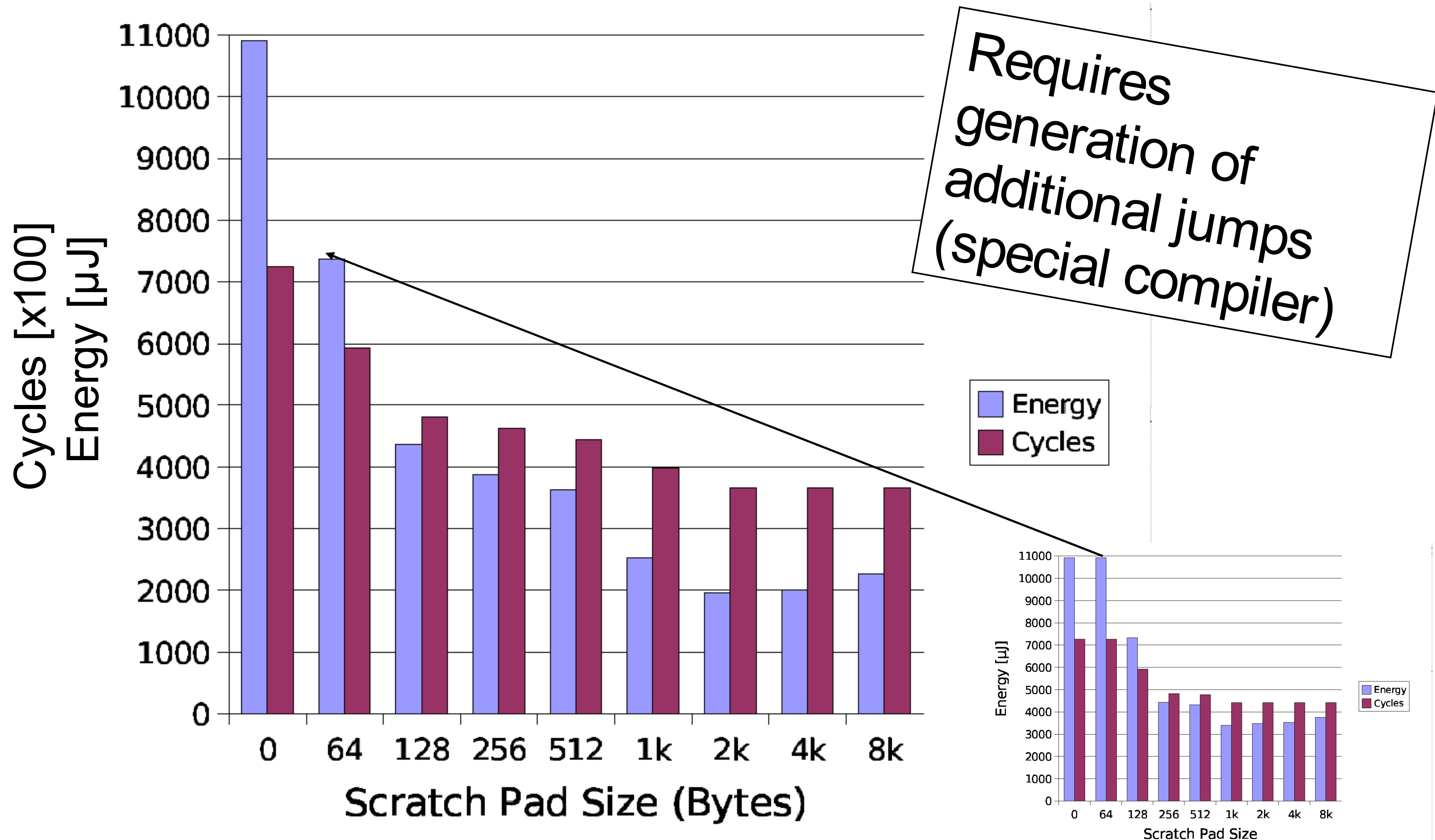
Allocation of basic blocks

Fine-grained granularity smoothens dependency on the size of the scratch pad.

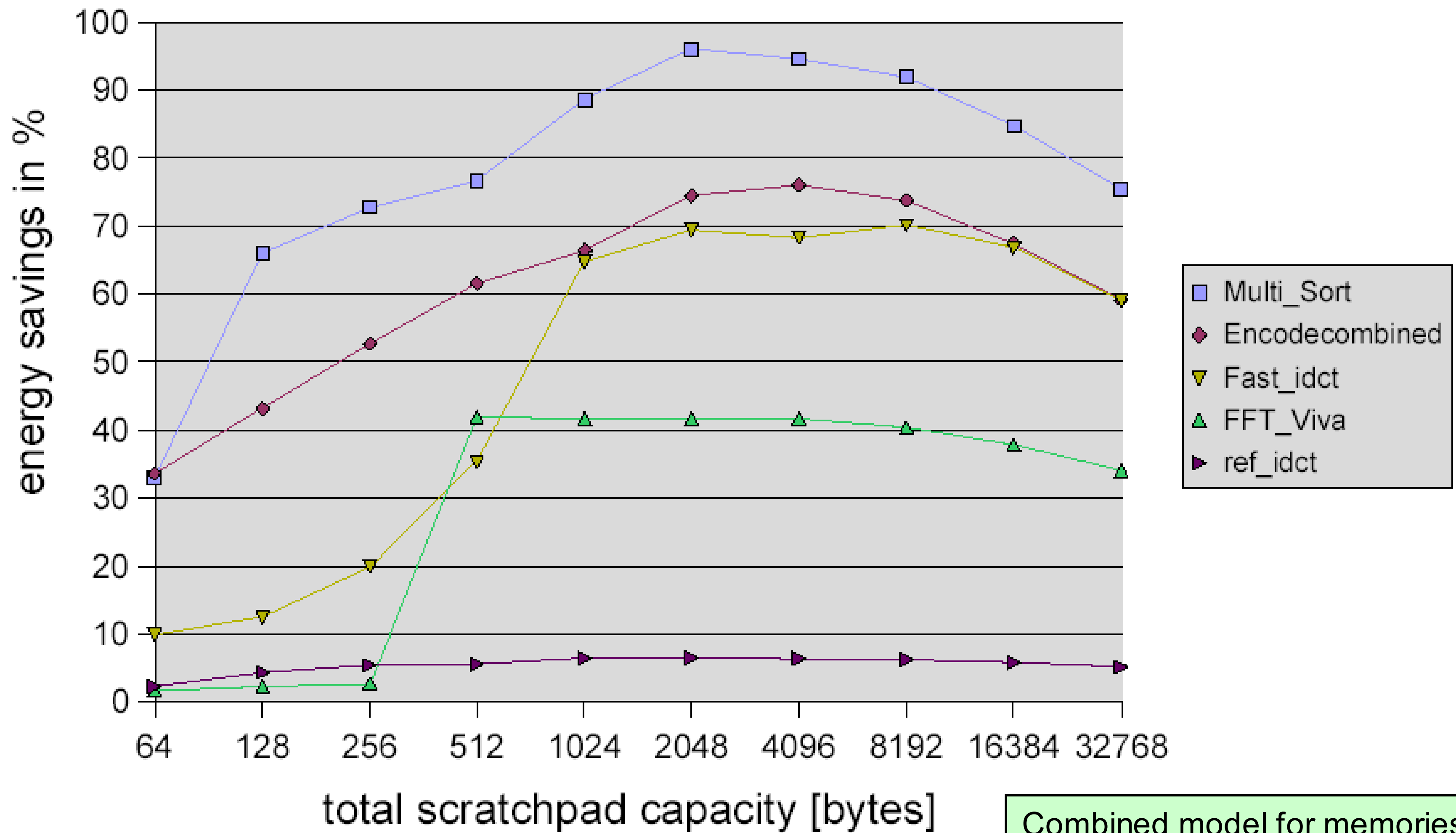
Requires additional jump instructions to return to "main" memory.



Allocation of basic blocks, sets of adjacent basic blocks and the stack

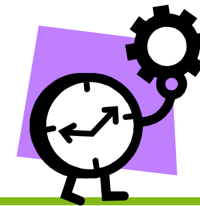


Savings for memory system energy alone



Combined model for memories

Timing predictability



C–Program → encc (with SP Size input) → executable → ARMulator → Actual Performance; executable → aiT → WCET

aiT:

- WCET analysis tool
- support for scratchpad memories by specifying different memory access times
- also features experimental cache analysis for ARM7

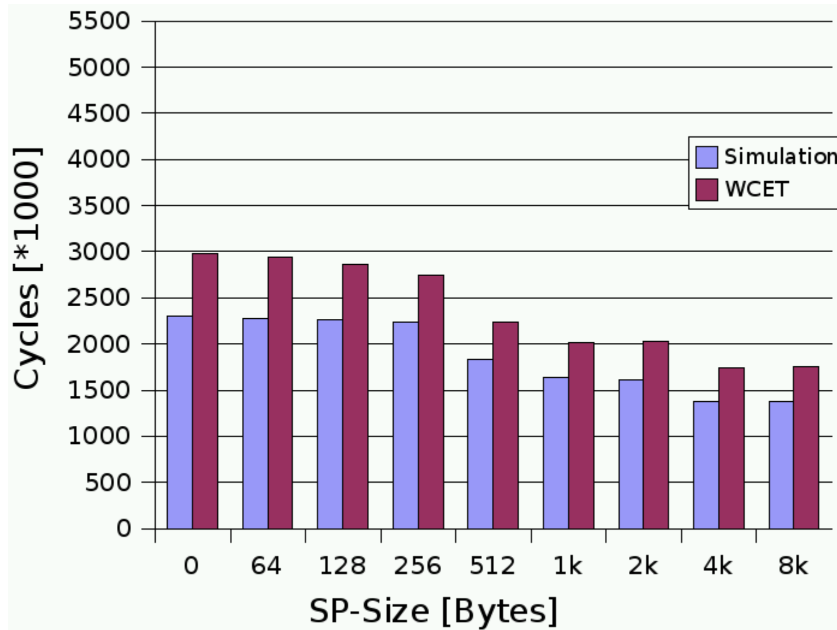
Architectures considered

ARM7TDMI with 3 different memory architectures:

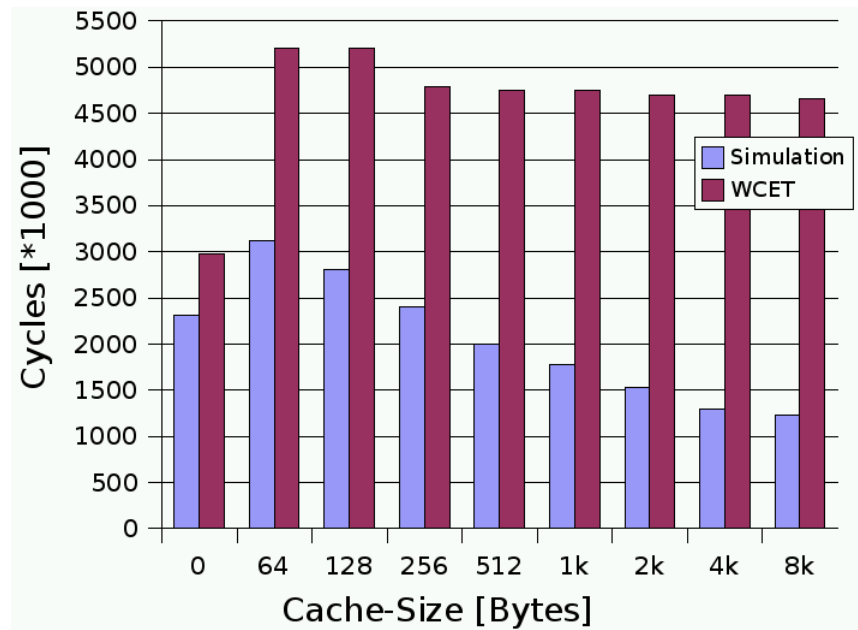
2. **Main memory**
   LDR-cycles: (CPU,IF,DF)=(3,2,2)
   STR-cycles: (2,2,2)
   * = (1,2,0)
3. **Main memory + unified cache**
   LDR-cycles: (CPU,IF,DF)=(3,12,6)
   STR-cycles: (2,12,3)
   * = (1,12,0)
4. **Main memory + scratch pad**
   LDR-cycles: (CPU,IF,DF)=(3,0,2)
   STR-cycles: (2,0,0)
   * = (1,0,0)

Results for G.721

Using Scratchpad:



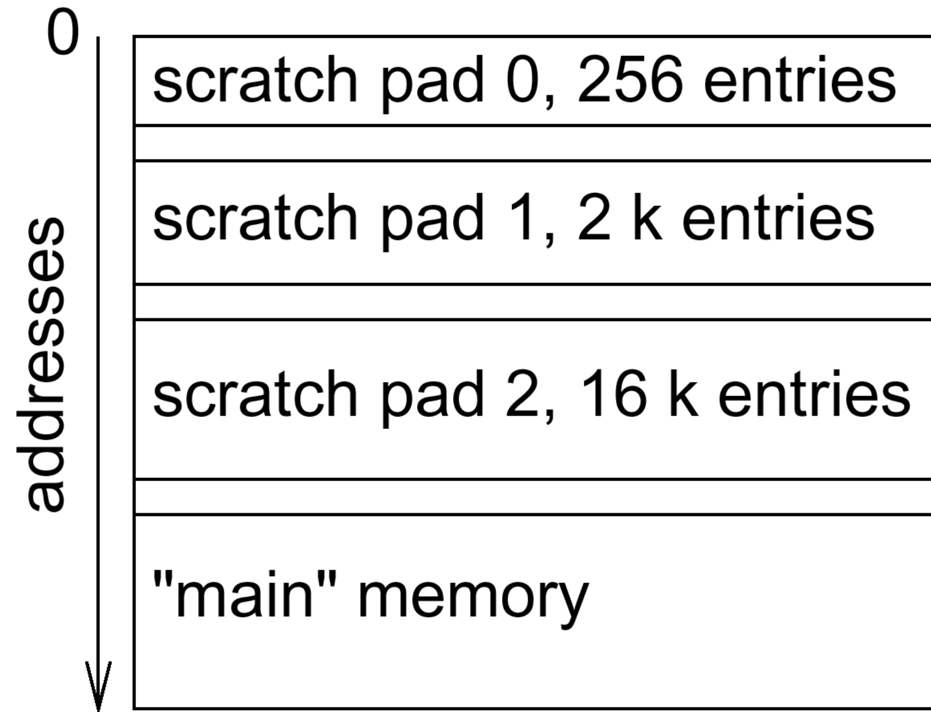
Using Unified Cache:



References:

- Wehmeyer, Marwedel: Influence of Onchip Scratchpad Memories on WCET: 4th Intl Workshop on worst-case execution time (WCET) analysis, Catania, Sicily, Italy, June 29, 2004
- Second paper on SP/Cache and WCET at DATE, March 2005

Multiple scratch pads



Optimization for multiple scratch pads

Minimize $$C = \sum_j e_j \cdot \sum_i x_{j,i} \cdot n_i$$

With e_j: energy per access to memory *j*,
and $x_{j,i}$= 1 if object *i* is mapped to memory *j*, =0 otherwise,
and n_i: number of accesses to memory object *i*,
subject to the constraints:

$$\forall j : \sum_i x_{j,i} \cdot S_i \leq SSP_j$$

$$\forall i : \sum_j x_{j,i} = 1$$

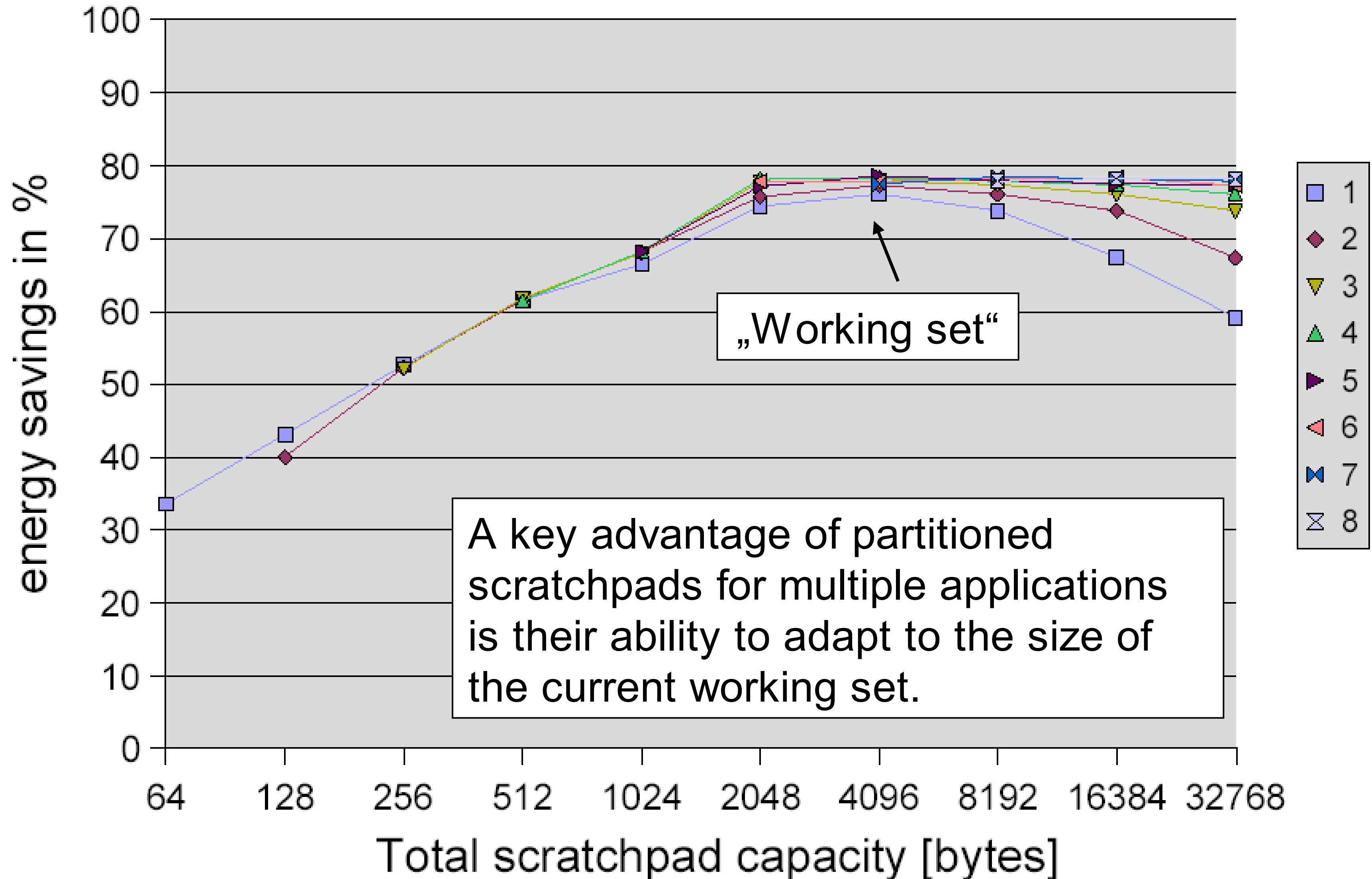
With S_i: size of memory object *i*,
SSP_j: size of memory *j*.

Considered partitions

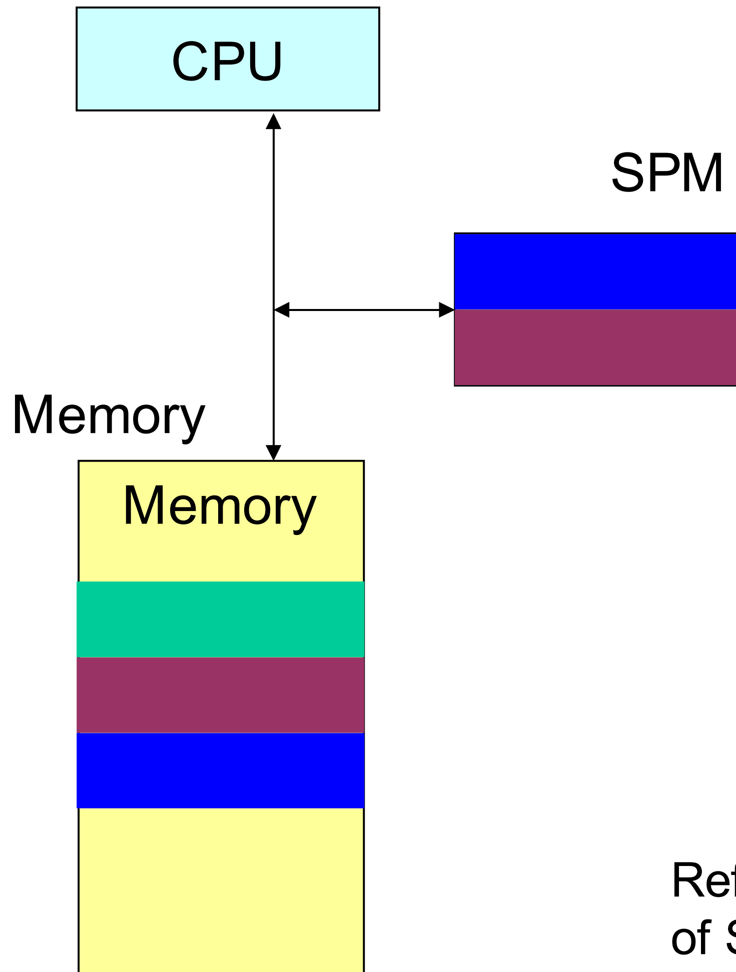
# of partitions	number of partitions of size:						
	4K	2K	1K	512	256	128	64
7	0	1	1	1	1	1	2
6	0	1	1	1	1	2	0
5	0	1	1	1	2	0	0
4	0	1	1	2	0	0	0
3	0	1	2	0	0	0	0
2	0	2	0	0	0	0	0
1	1	0	0	0	0	0	0

Table 1: Example of all considered memory partitions for a total capacity of 4096 bytes

Results for parts of GSM coder/decoder



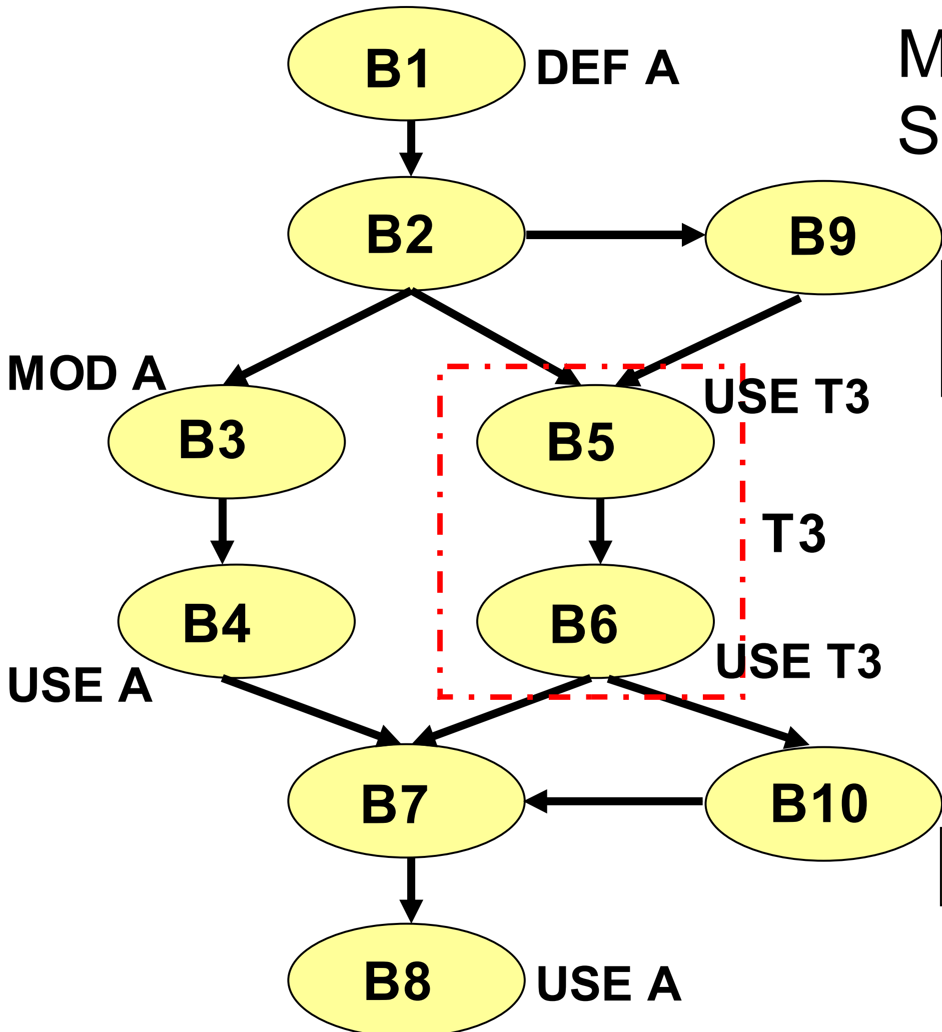
Dynamic replacement within scratch pad



- Effectively results in a kind of **compiler-controlled segmentation/ paging** for SPM
- Address assignment within SPM required (paging or segmentation-like)

Reference: Verma, Marwedel: Dynamic Overlay of Scratchpad Memory for Energy Minimization, ISSS 2004

Dynamic replacement of *data* within scratch pad: based on liveness analysis



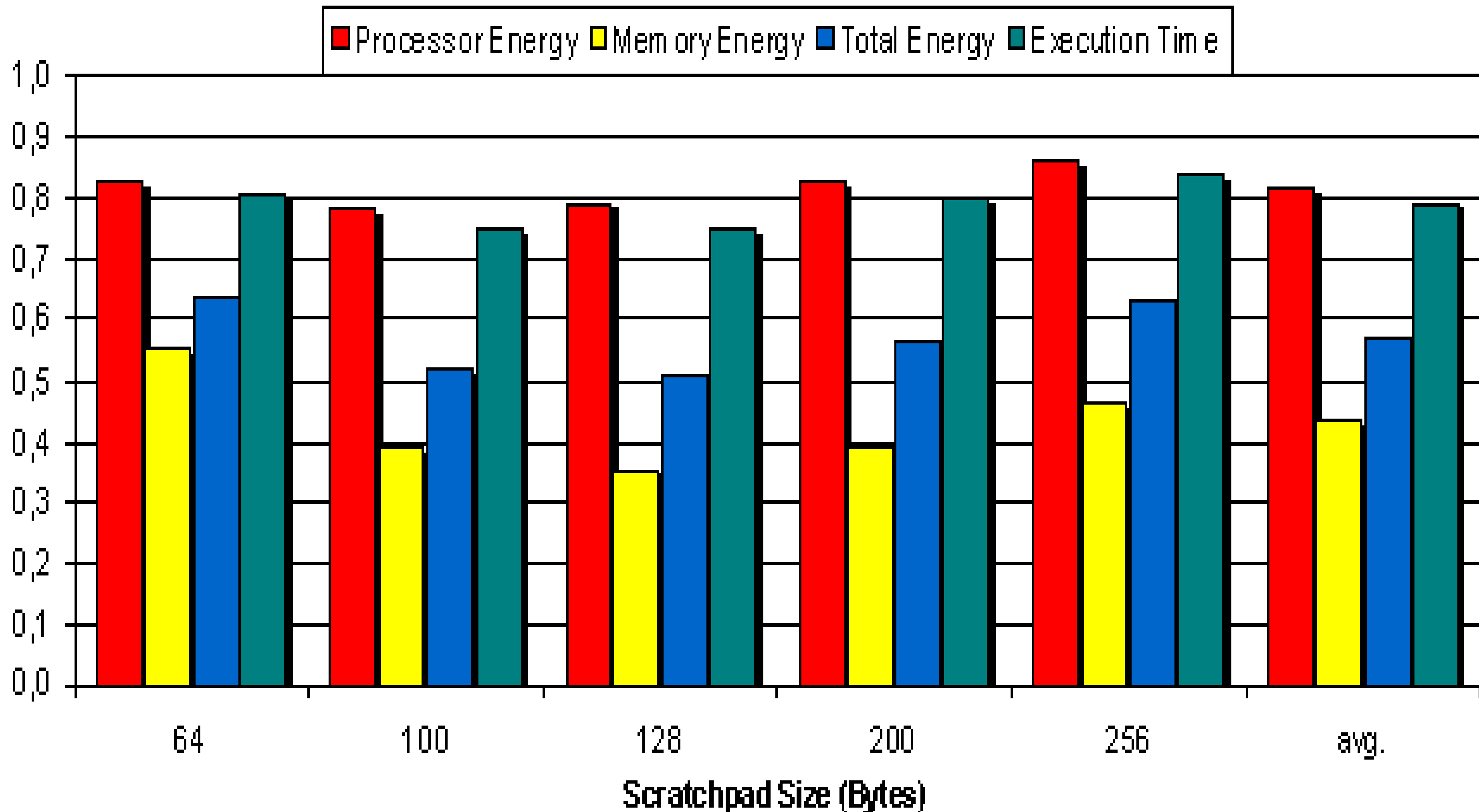
MO = {A, T1, T2, T3, T4}
SP Size = |A| = |T1| …= |T4|

SPILL_STORE(A);
SPILL_LOAD(T3);

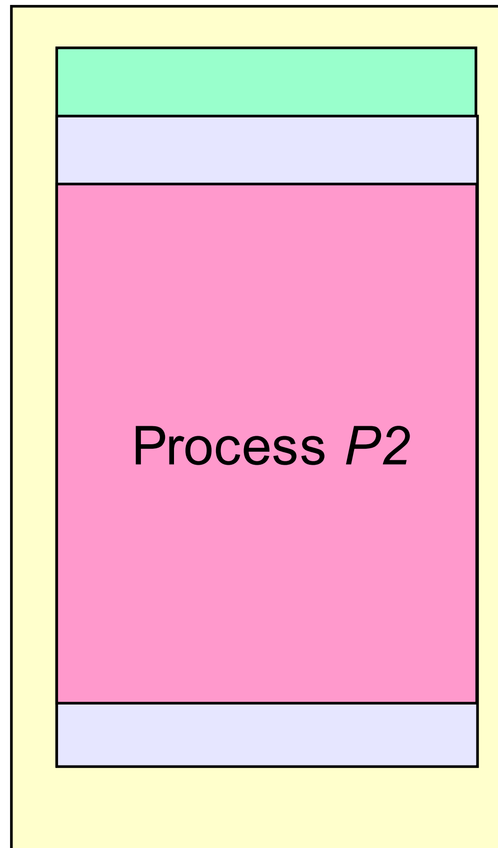
Solution:
A → SP & T3 → SP

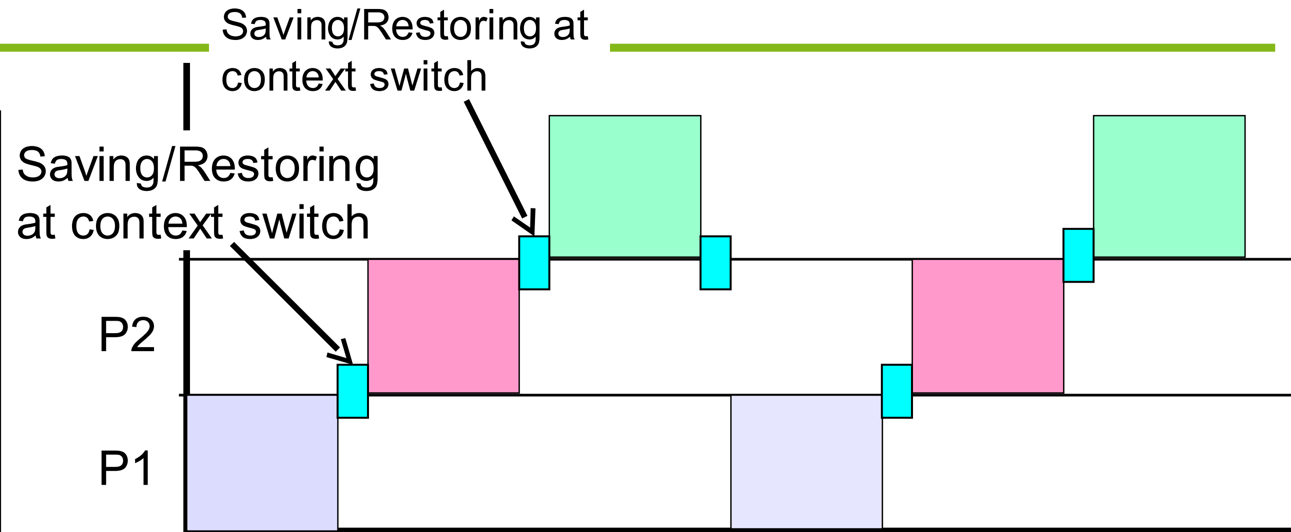
SPILL_LOAD(A);

Dynamic replacement within scratch pad
- Results for edge detection relative to static allocation -



Saving/Restoring Context Switch

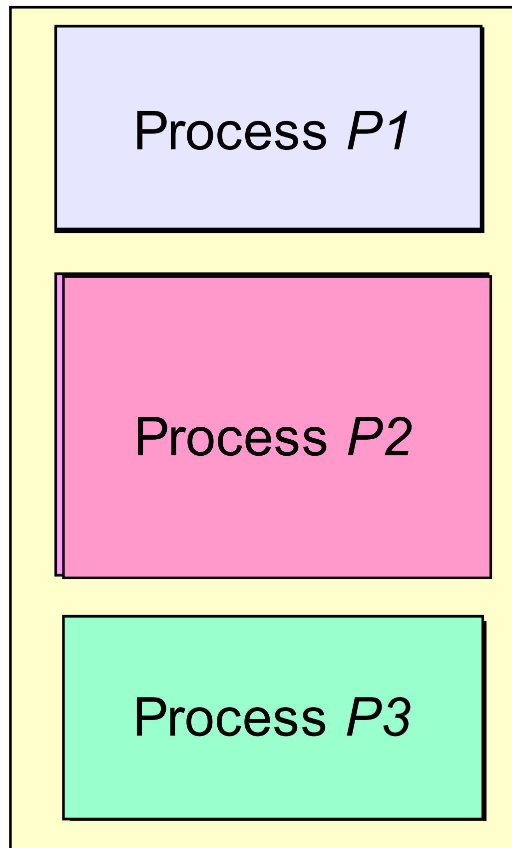


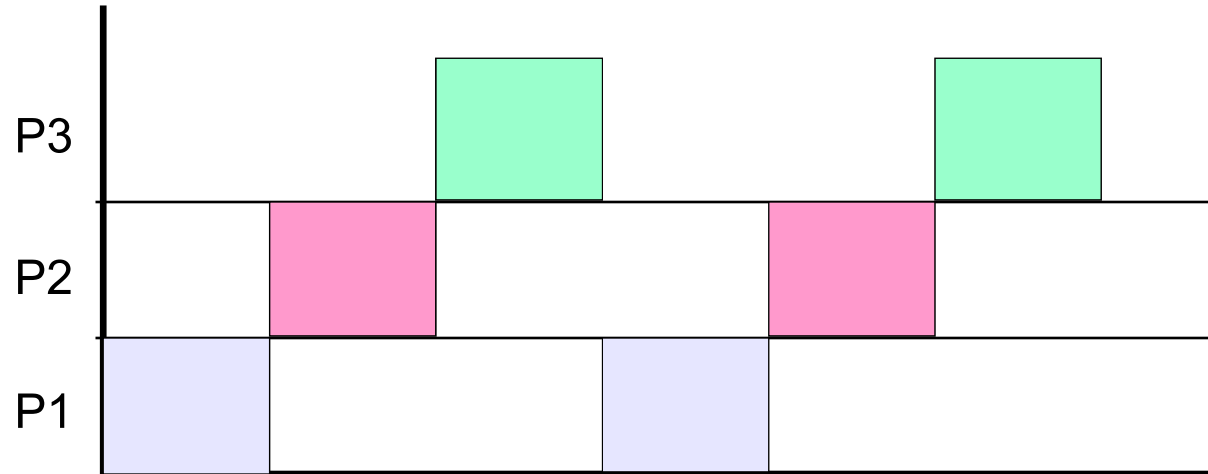


Saving Context Switch (Saving)

- Utilizes SPM as a common region shared all processes
- Contents of processes are copied on/off the SPM at context switch
- Good for small scratchpads

Non-Saving Context Switch

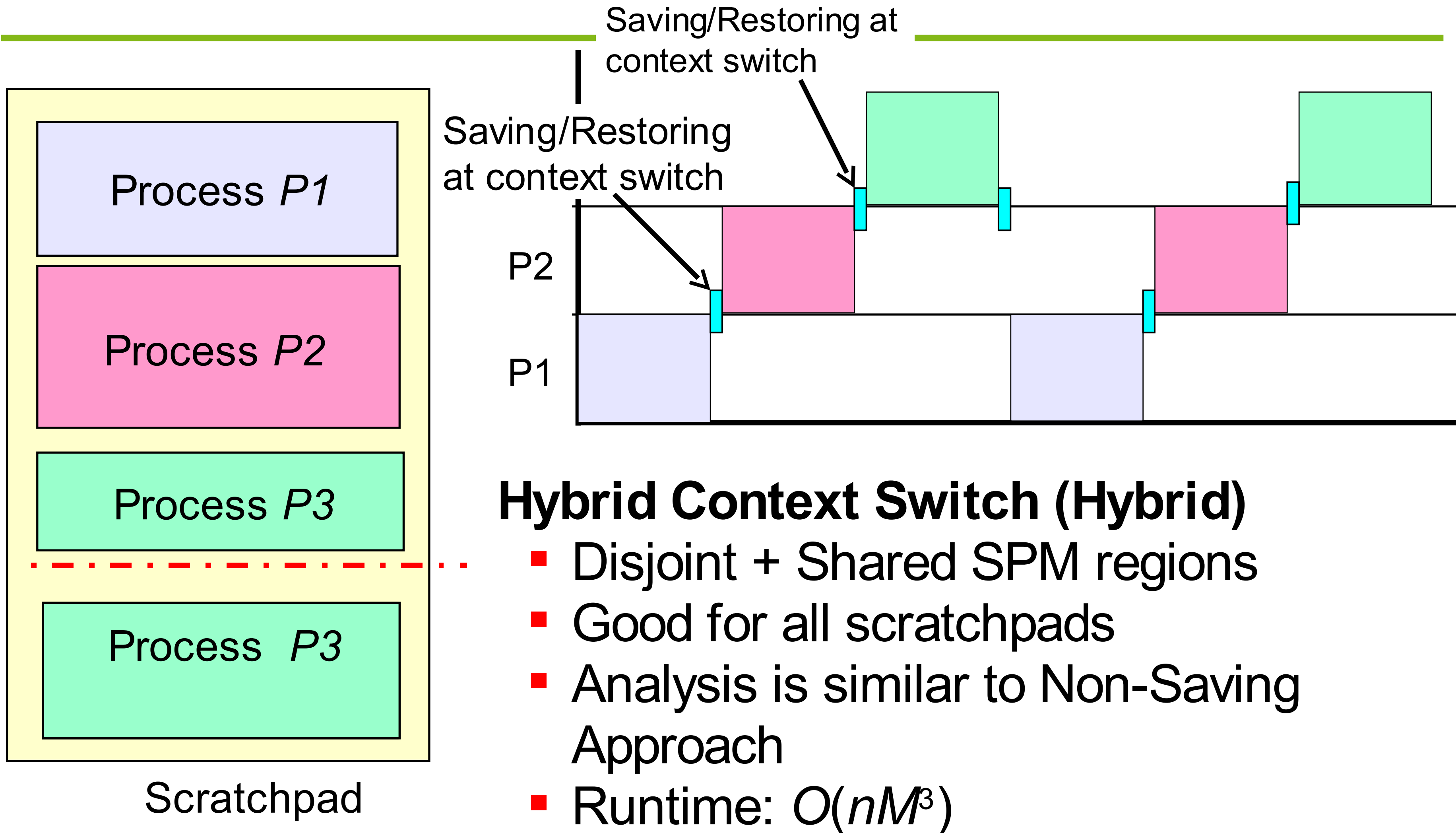




Non-Saving Context Switch

- Partitions SPM into disjoint regions
- Each process is assigned a SPM region
- Copies contents during initialization
- Good for large scratchpads

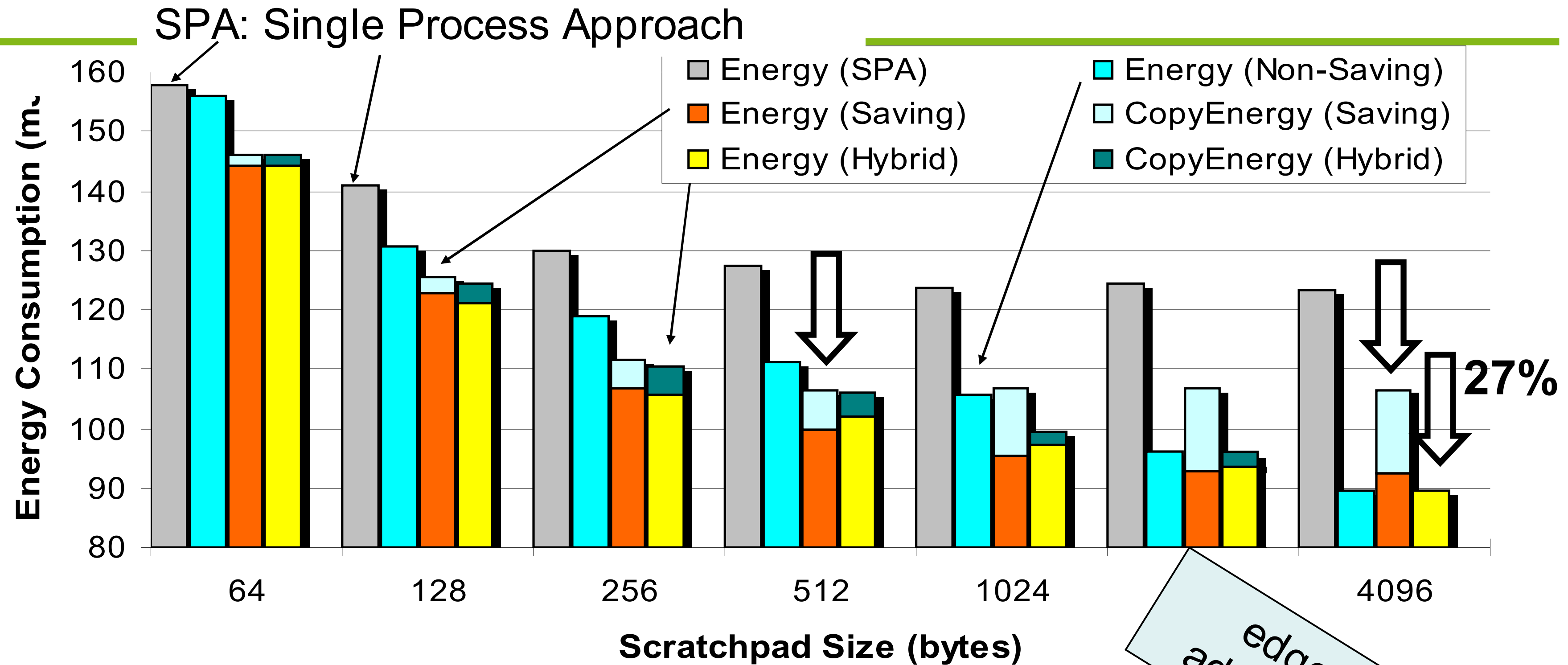
Hybrid Context Switch



Hybrid Context Switch (Hybrid)

- Disjoint + Shared SPM regions
- Good for all scratchpads
- Analysis is similar to Non-Saving Approach
- Runtime: $O(nM^3)$

Multi-process Scratchpad Allocation: Results



- For small SPMs (64B-512B) Saving is better
- For large SPMs (1kB- 4kB) Non-Saving is better
- Hybrid is the best for all SPM sizes.
- Energy reduction @ 4kB SPM is 27% for Hybrid approach

Summary

High-level transformations

- Loop nest splitting
- Array folding

Impact of memory architecture on execution times & energy.

The SPM provides

- Runtime efficiency
- Energy efficiency
- Timing predictability

Achieved savings are sometimes dramatic, for example:

- savings of ~ 95% of the memory system energy