

Optimizations

- Compilation for Embedded Processors -

Peter Marwedel
TU Dortmund
Informatik 12
Germany

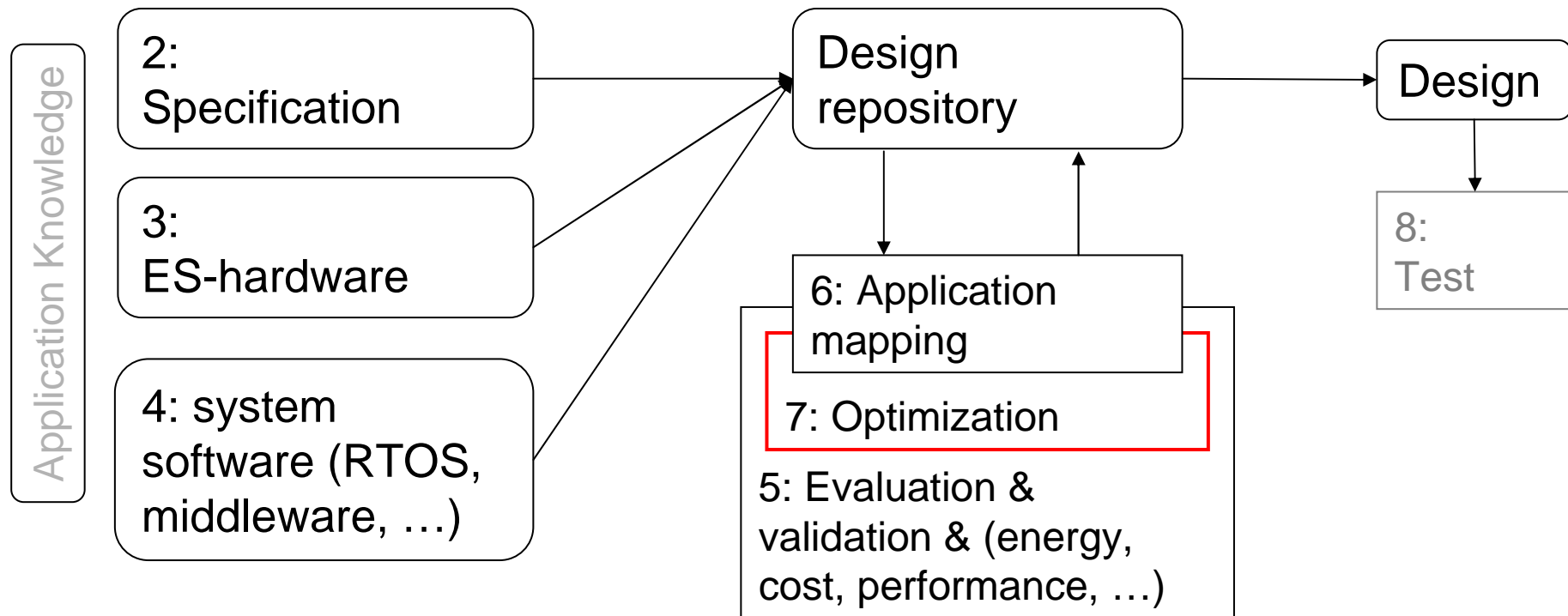


Graphics: © Alexandra Nolte, Gesine Marwedel, 2003

2011年 01 月 12 日


These slides use Microsoft clip arts.
Microsoft copyright restrictions apply.

Structure of this course



Numbers denote sequence of chapters

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

Energy-aware compilation (1): Optimization for low-energy the same as 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

```
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 μ J

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

2231 cycles
16.47 μ J

```
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
```

Energy-aware compilation (2)

- 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;
```

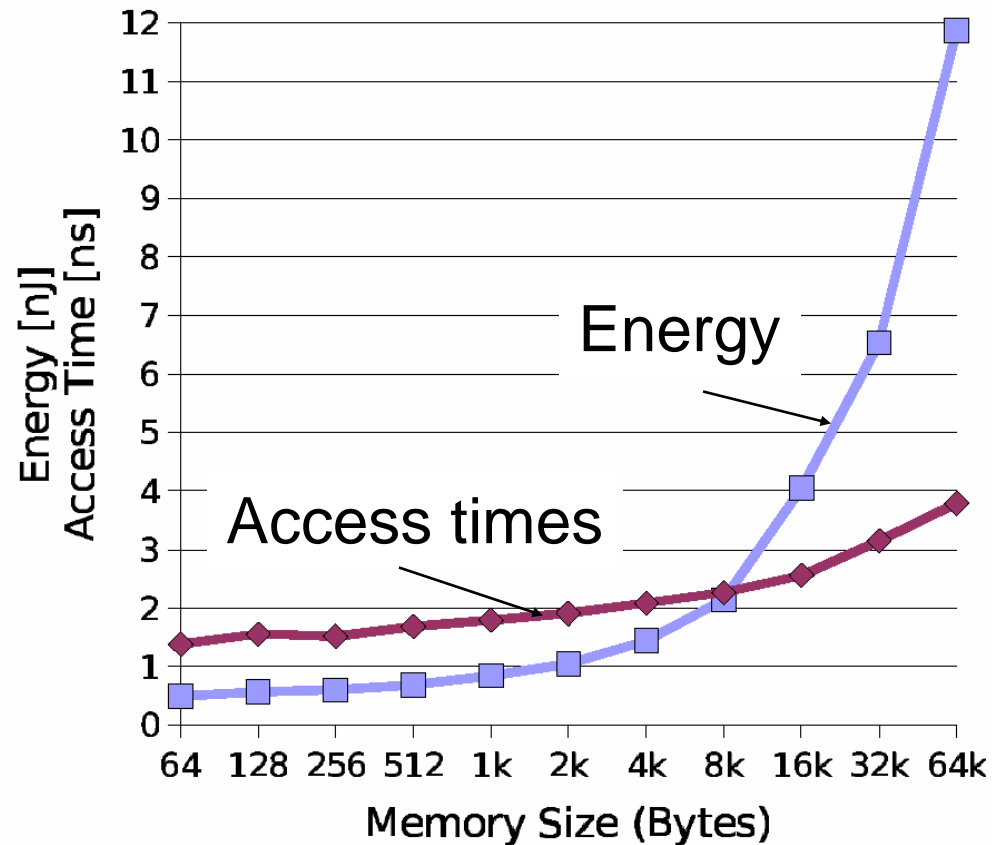
Exploitation of the memory hierarchy

Energy-aware compilation (3)

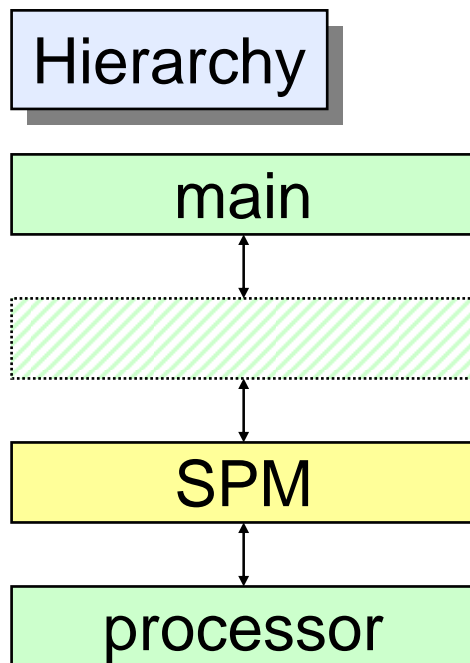
- Energy-aware scheduling: the order of the instructions can be changes as long as the meaning does not change.
Goal: reduction of the number of signal transitions
Popular (can be done as a post-pass optimization with no change to the compiler).
- Energy-aware instruction selection: among valid instruction sequences, select those minimizing energy consumption
- Exploitation of the memory hierarchy: huge difference between the energy consumption of small and large memories

3 key problems for future memory systems

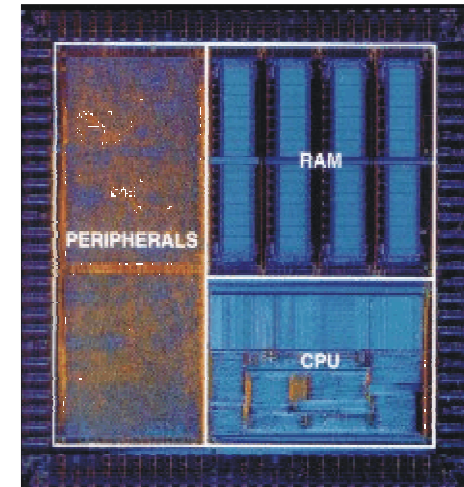
1. (Average) Speed
2. Energy/Power
3. Predictability/WCET



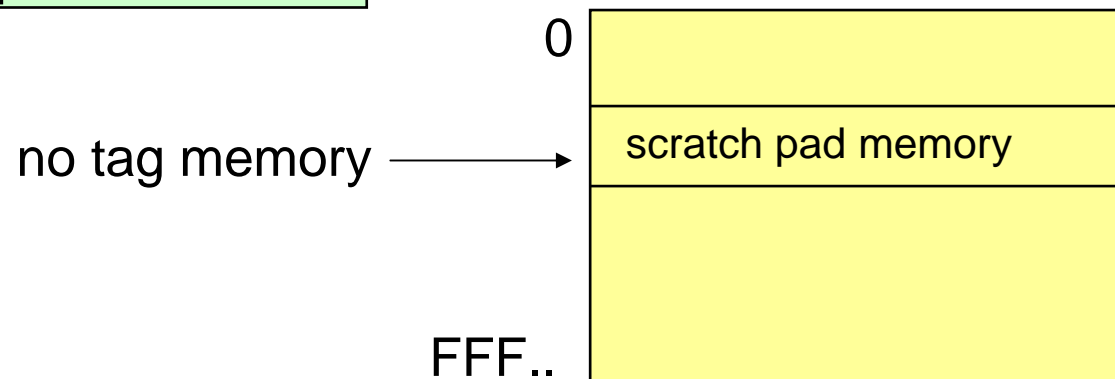
Hierarchical memories using scratch pad memories (SPM)



Example



Address space



ARM7TDMI
cores, well-known
for low power
consumption

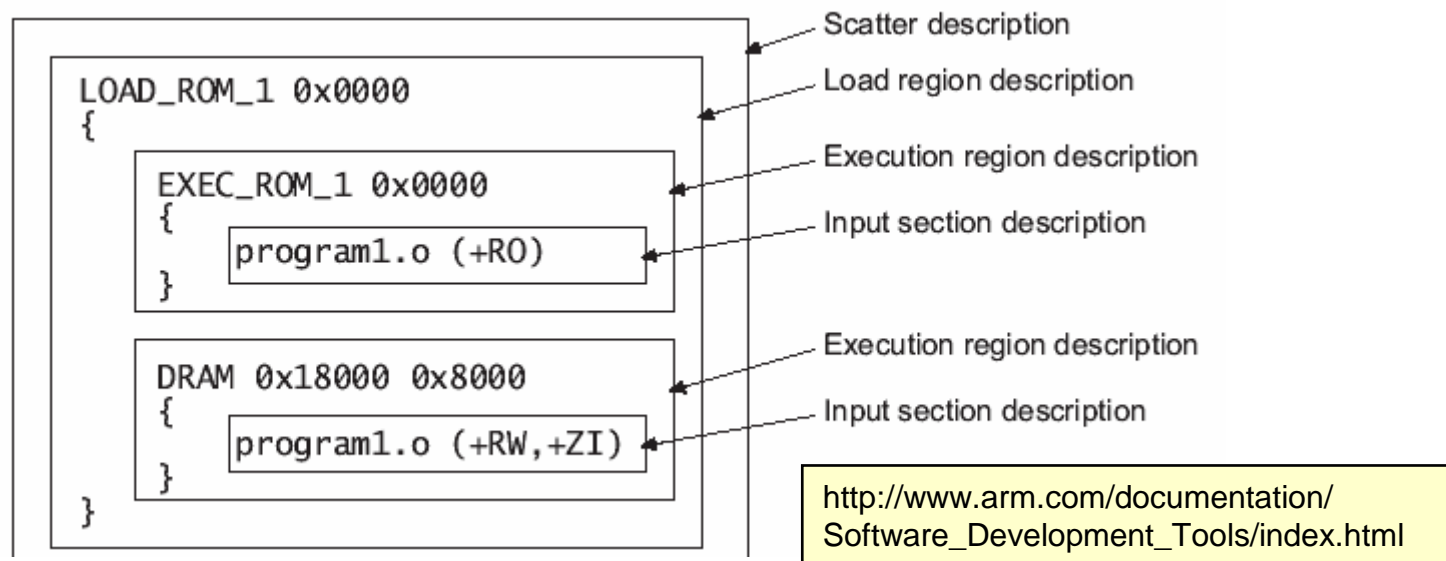
Very limited support in ARMcc-based tool flows

1. Use pragma in C-source to allocate to specific section:

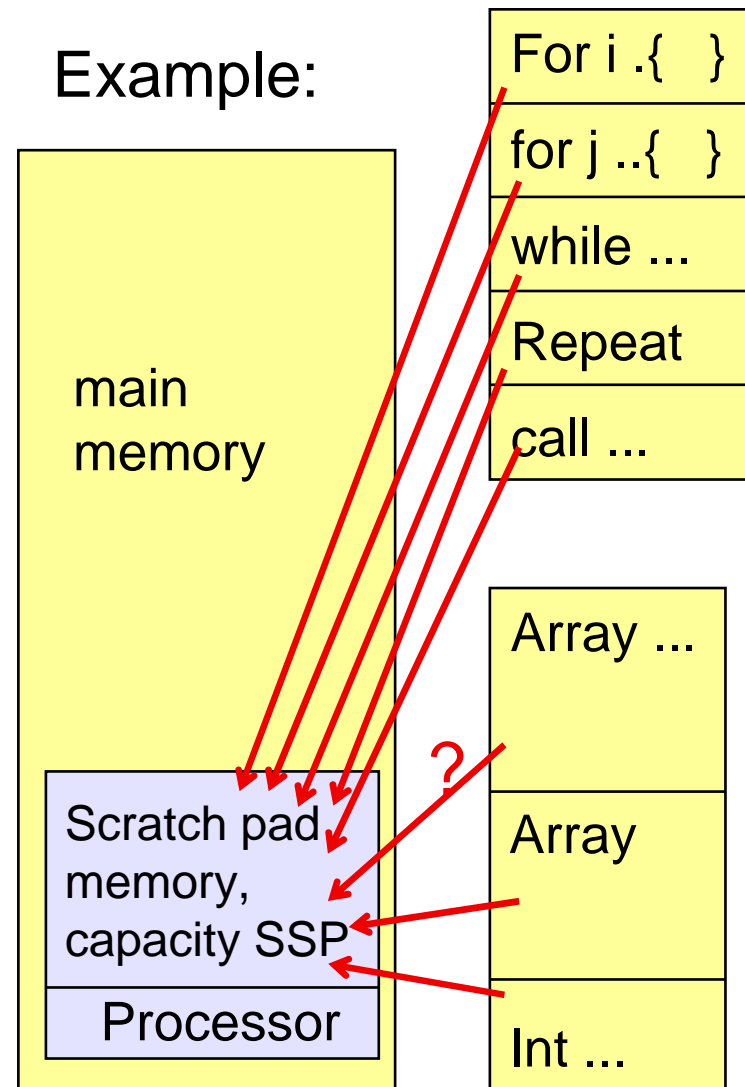
For example:

```
#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



Migration of data & instructions, global optimization model (TU Dortmund)



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

Non-overlapping (“Static”) allocation:

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

Solution: knapsack algorithm.

Overlaying (“dynamic”) allocation:

Moving objects back and forth

ILP representation

- migrating functions and variables-

Symbols:

$S(var_k)$ = size of variable k

$n(var_k)$ = 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

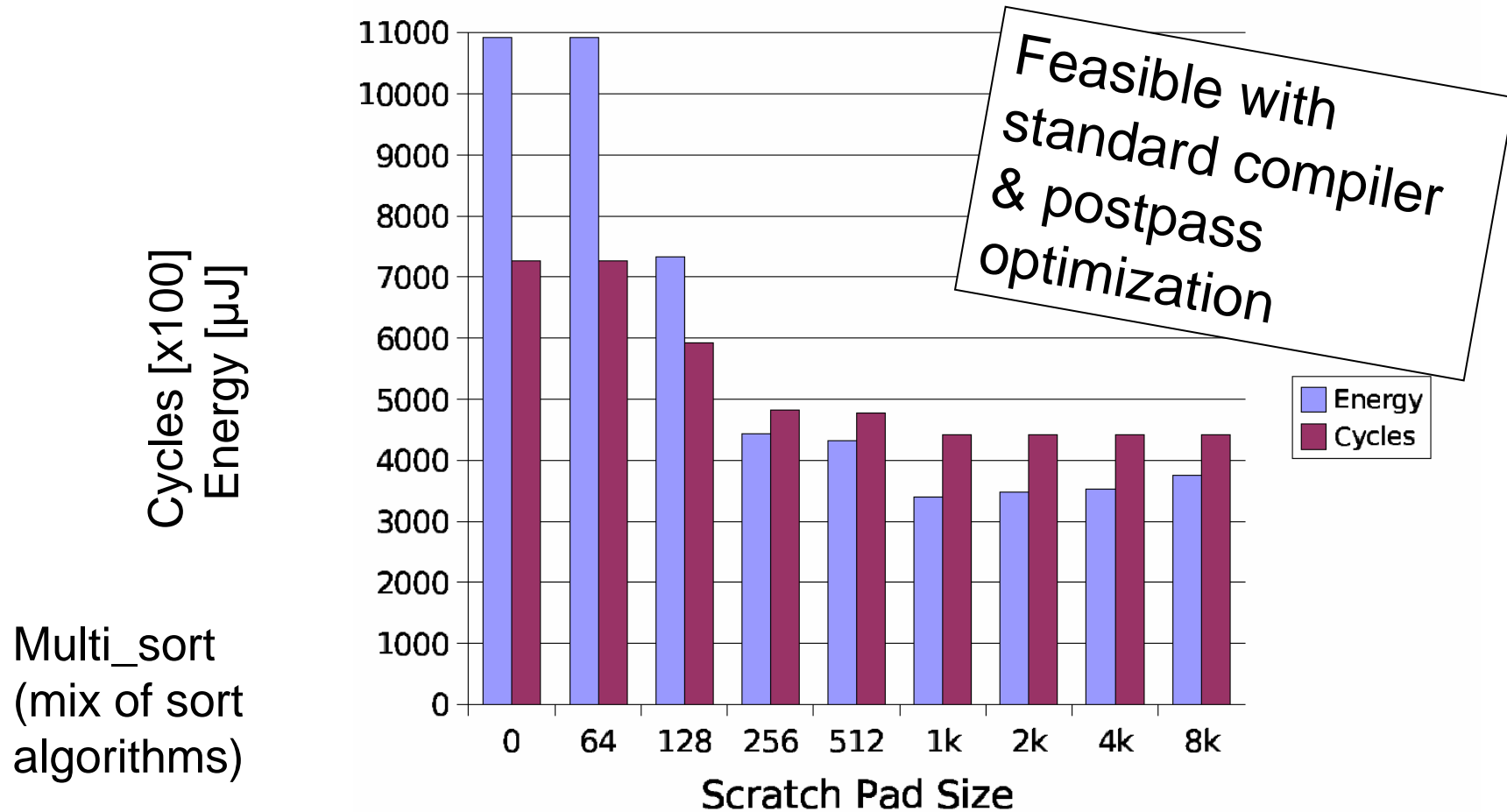
Integer programming formulation:

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

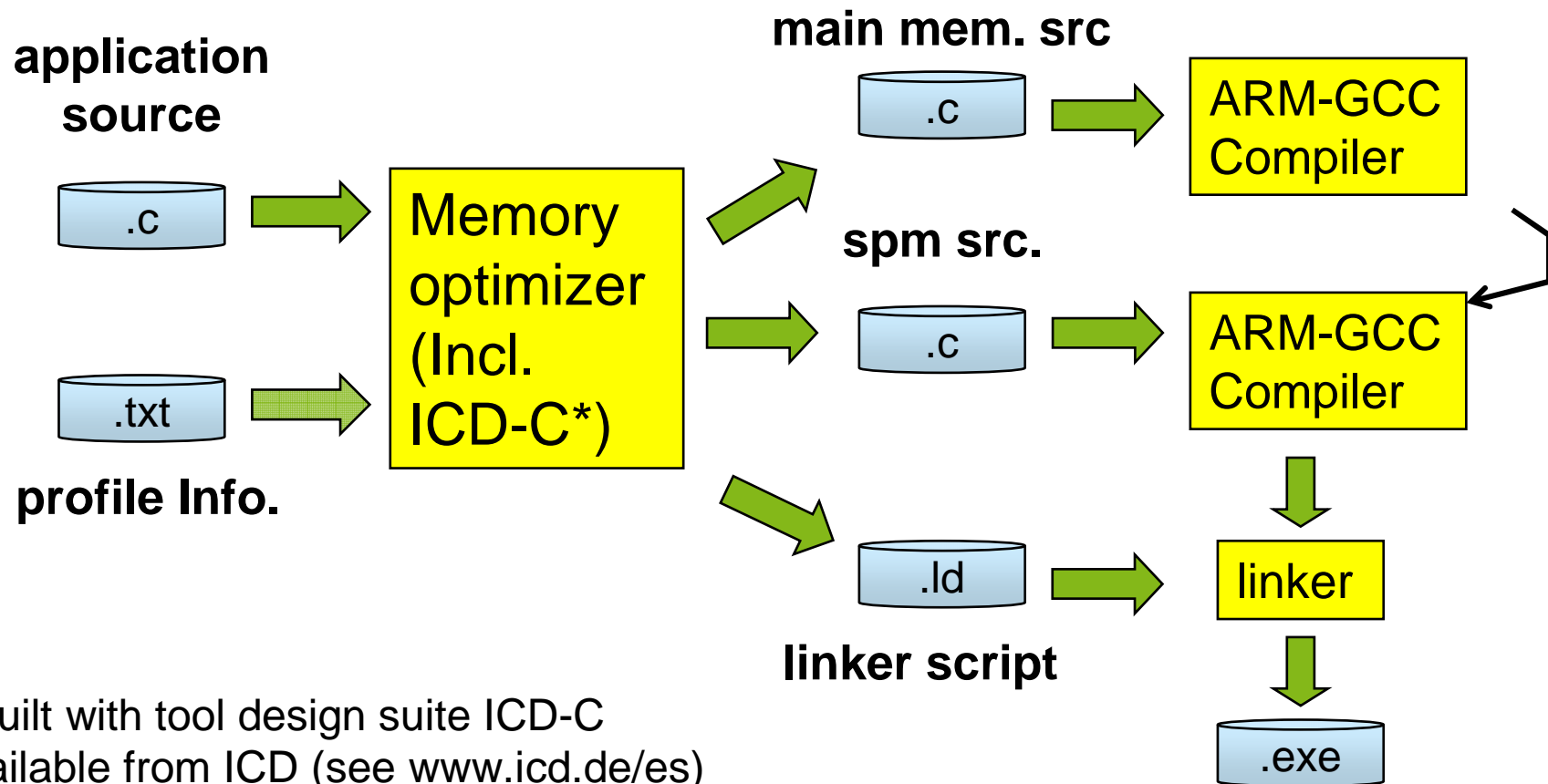


Measured processor / external memory energy +
CACTI values for SPM (combined model)

Numbers will change with technology,
algorithms remain unchanged.

Using these ideas with an gcc-based tool flow

Source is split into 2 different files by specially developed memory optimizer tool *.

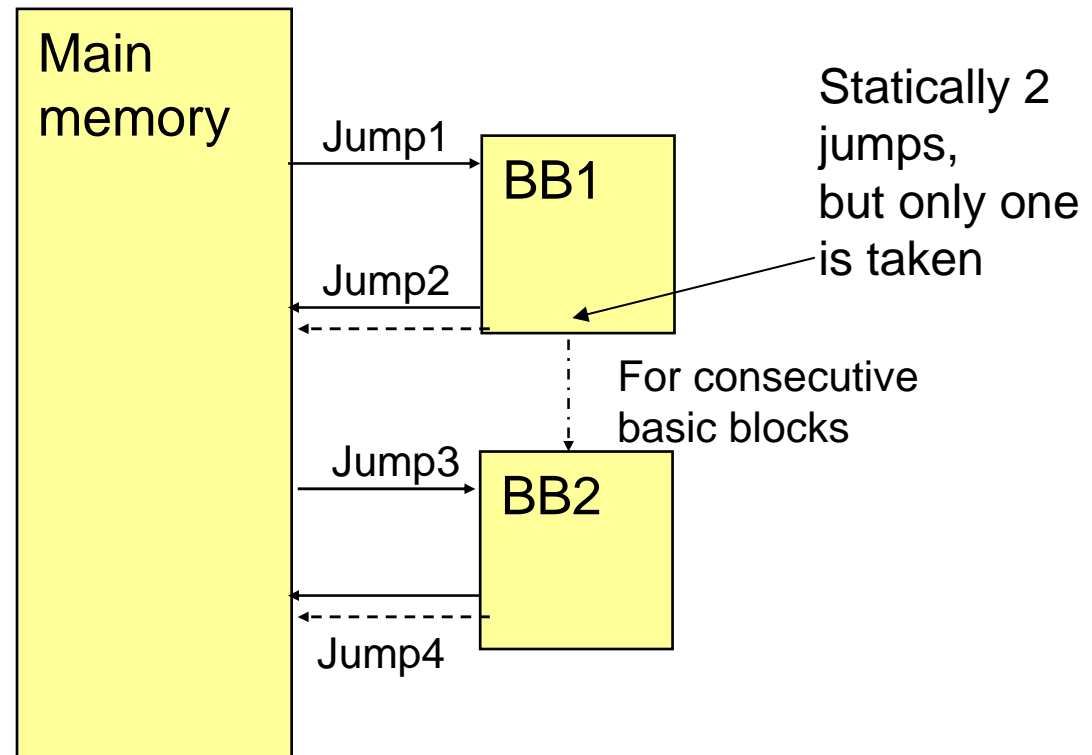


* Built with tool design suite ICD-C
available from ICD (see www.icd.de/es)

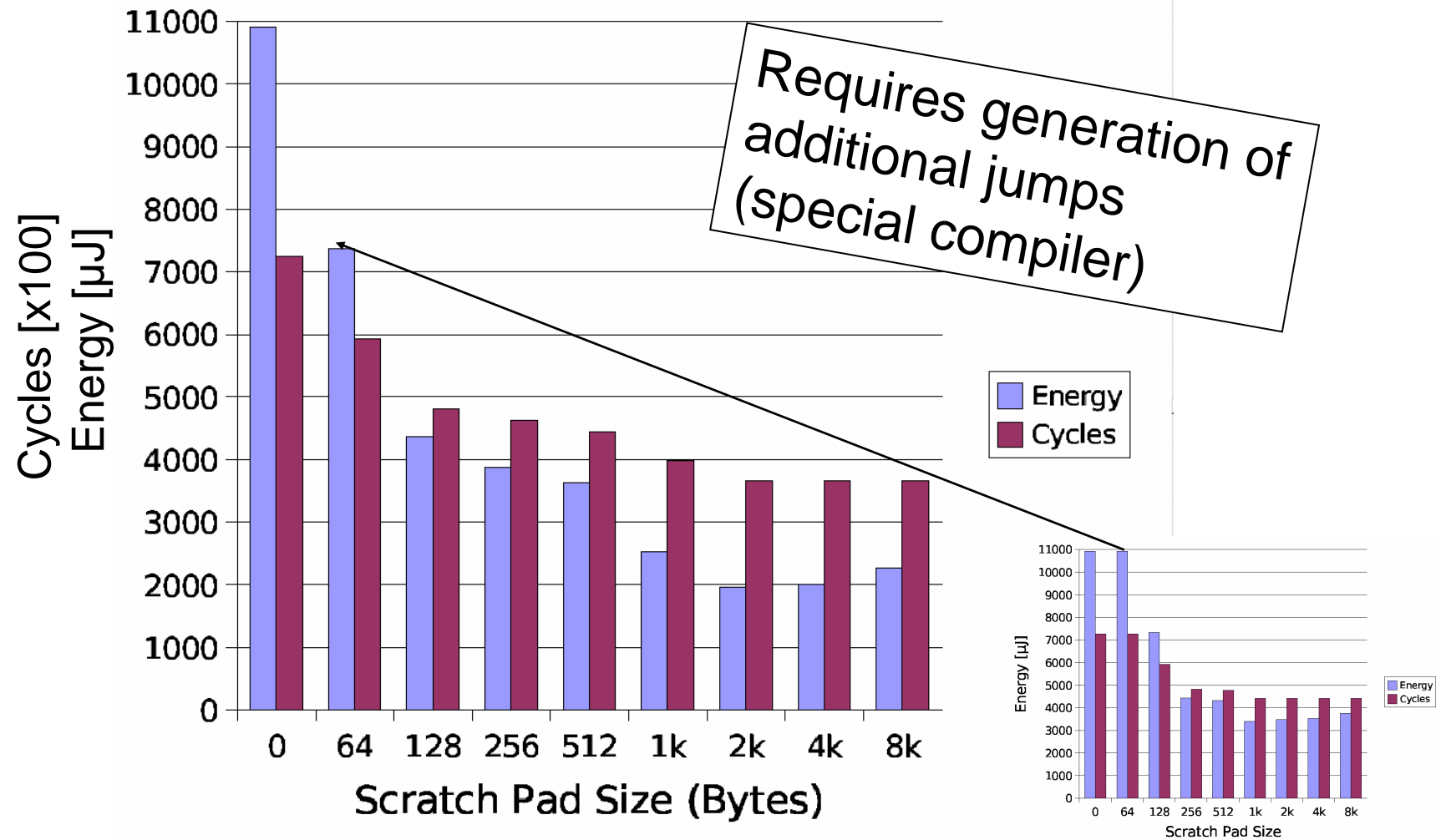
Allocation of basic blocks

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

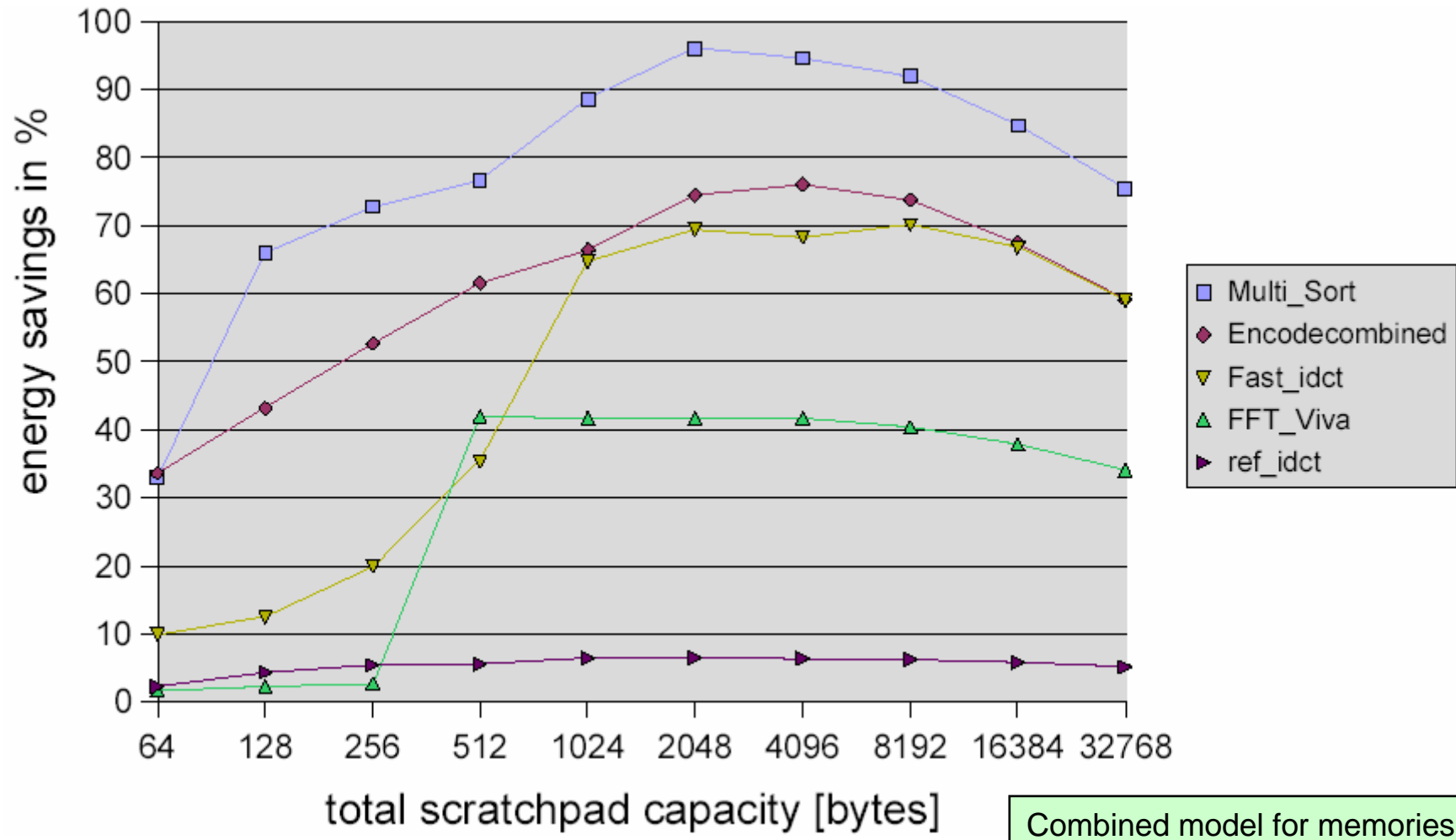
Requires additional
jump instructions to
return to "main"
memory.



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



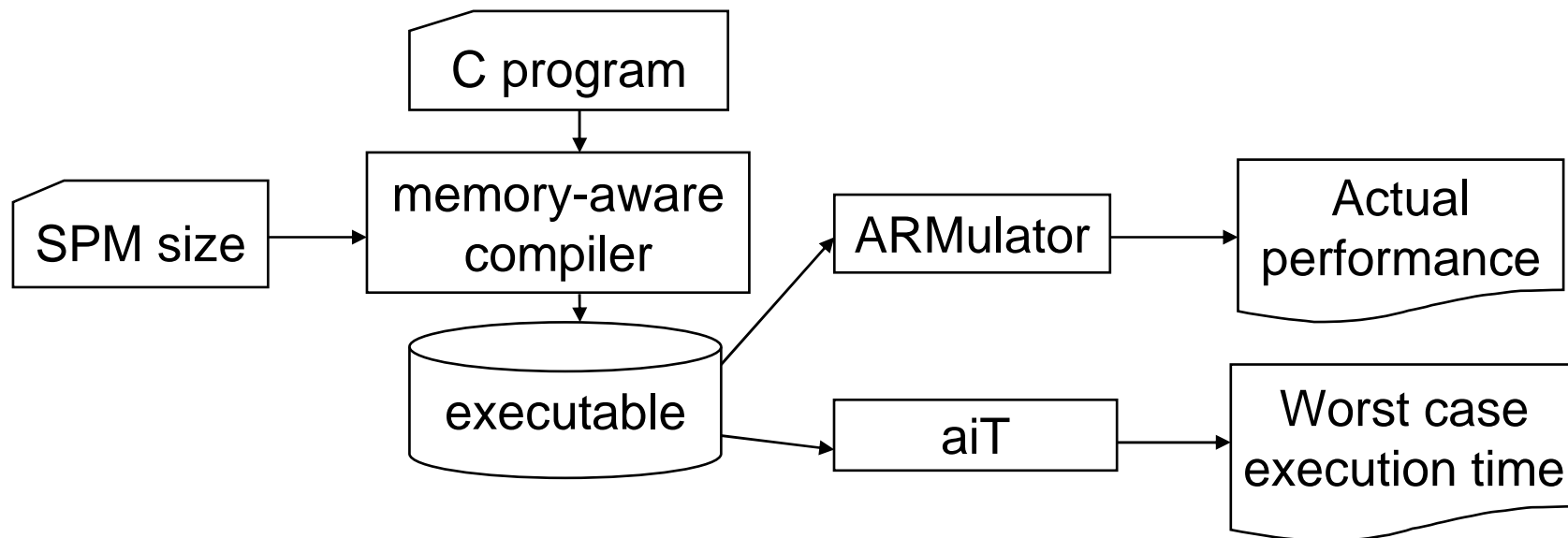
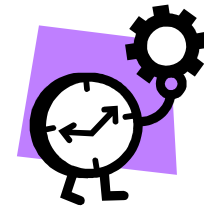
Savings for memory system energy alone



Scratch-pad/tightly coupled memory based predictability

Pre run-time scheduling is often the only practical means of providing predictability in a complex system. [Xu, Parnas]

- 👉 Time-triggered, statically scheduled operating systems
- 👉 Let's do the same for the memory system
 - 👉 Are SPMs really more timing predictable?
 - 👉 Analysis using the aiT timing analyzer



Architectures considered

ARM7TDMI with 3 different memory architectures:

1. Main memory

LDR-cycles: (CPU,IF,DF)=(3,2,2)

STR-cycles: (2,2,2)

* = (1,2,0)

2. Main memory + unified cache

LDR-cycles: (CPU,IF,DF)=(3,12,6)

STR-cycles: (2,12,3)

* = (1,12,0)

3. Main memory + scratch pad

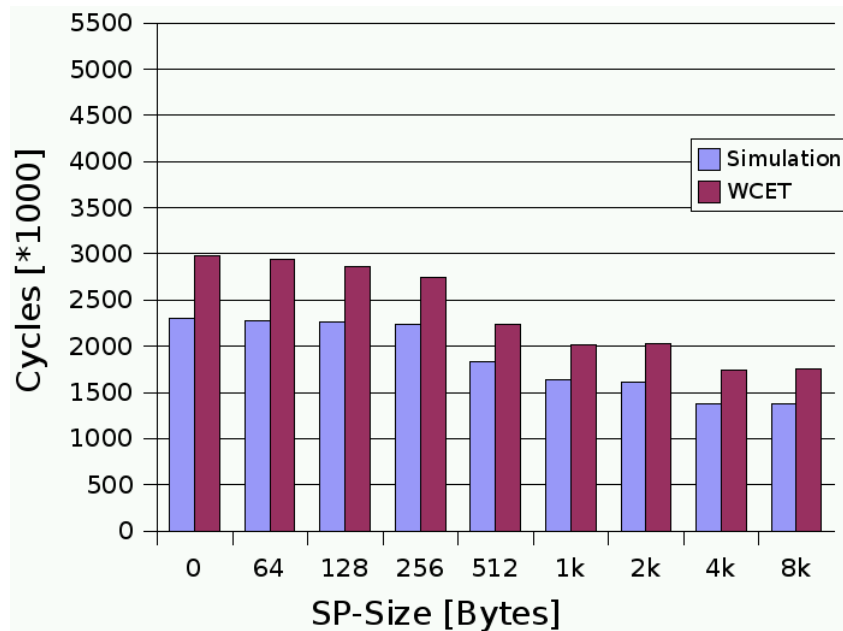
LDR-cycles: (CPU,IF,DF)=(3,0,2)

STR-cycles: (2,0,0)

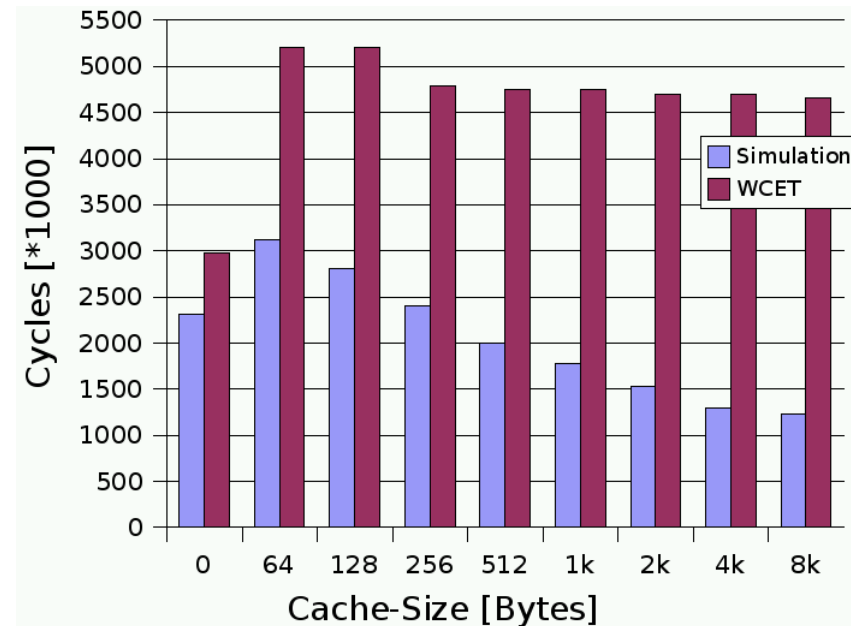
* = (1,0,0)

Results for G.721

Using Scratchpad:



Using Unified Cache:



References:

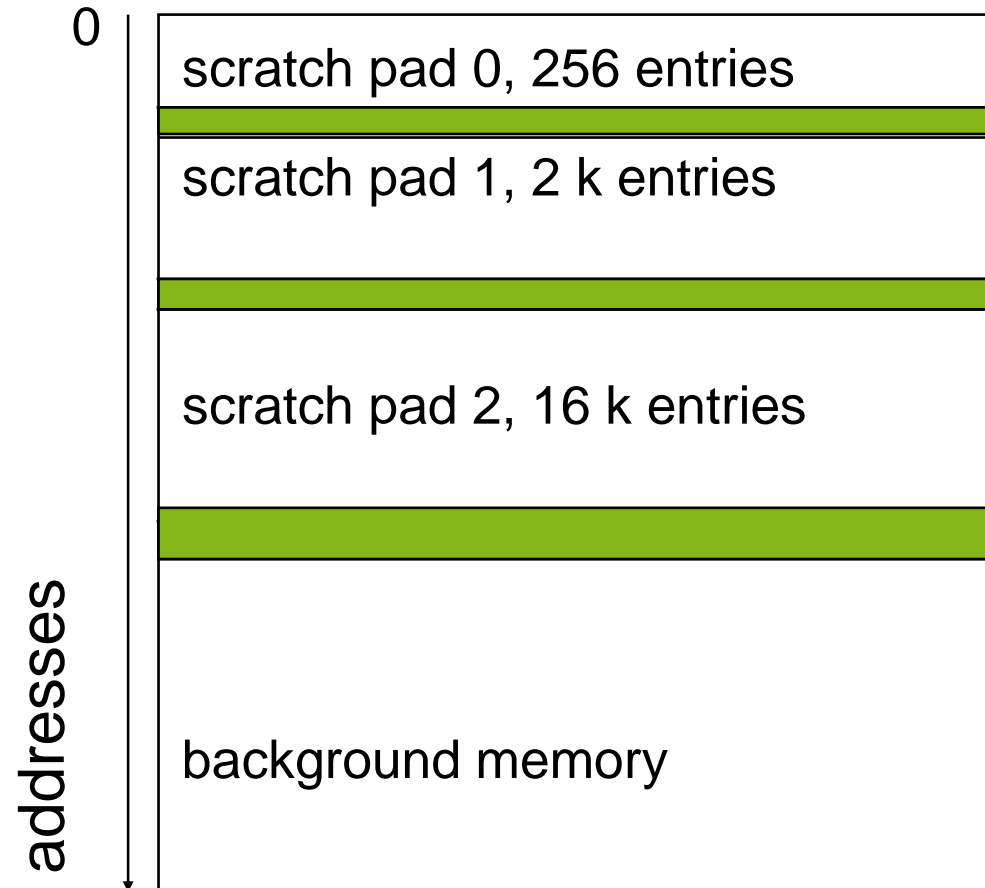
- Wehmeyer, Marwedel: Influence of Onchip Scratchpad Memories on WCET: 4th Intl Workshop on worst-case execution time (WCET) analysis, Catania, Sicily, Italy, June 29, 2004
- Second paper on SP/Cache and WCET at DATE, March 2005

Multiple scratch pads

Small is beautiful:

One small SPM is beautiful (😊).

May be, several smaller SPMs are even more beautiful?



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:

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

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

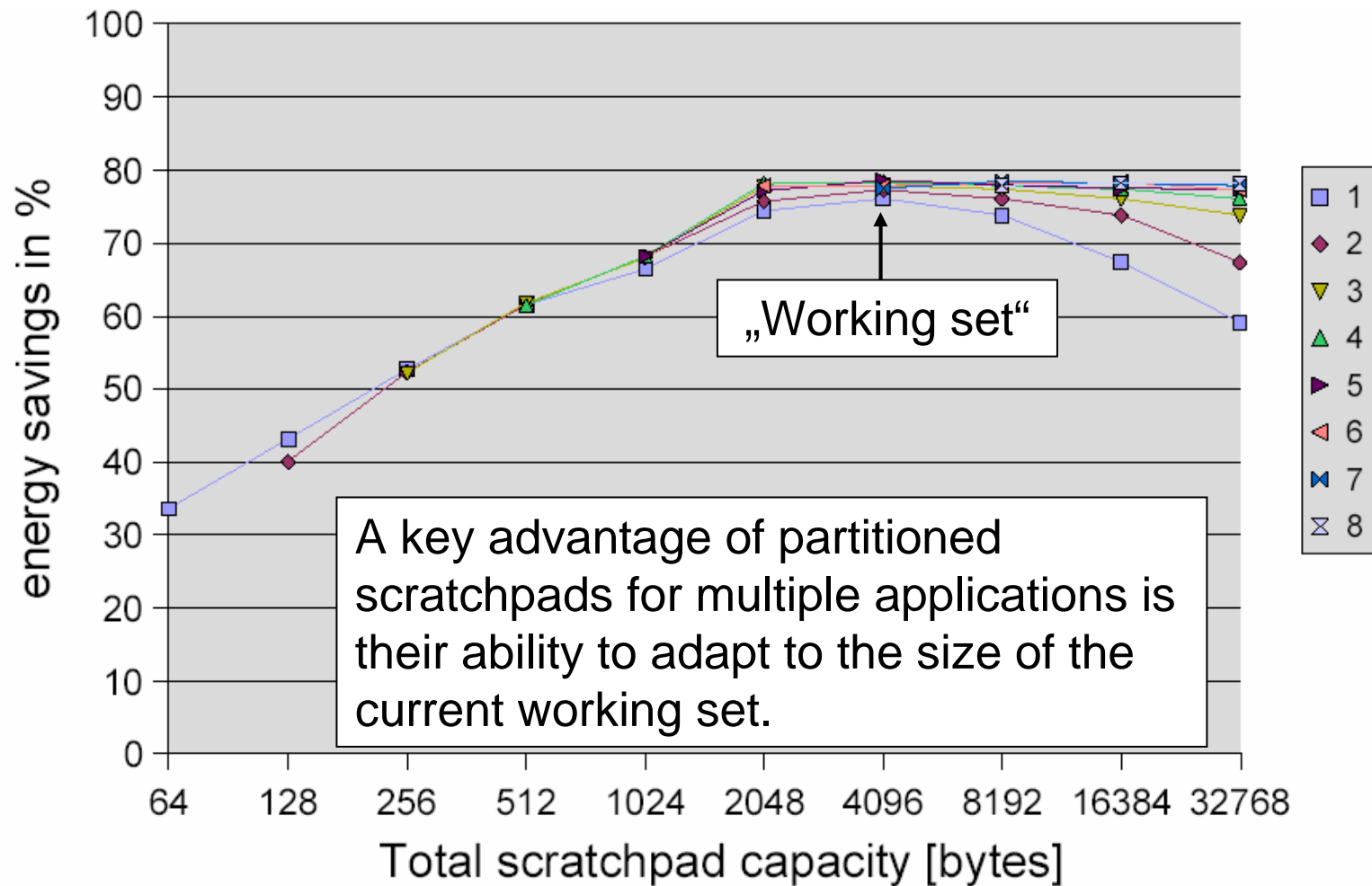
With S_i : size of memory object i ,
 SSP_j : size of memory j .

Considered partitions

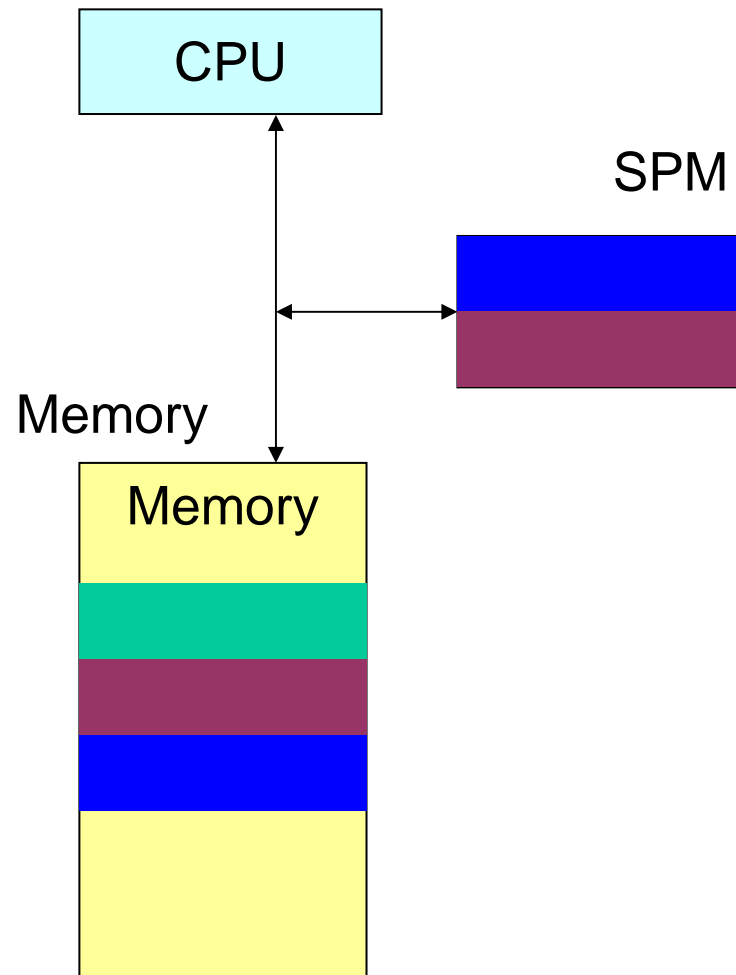
Example of considered memory partitions for a total capacity of 4096 bytes

# 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

Results for parts of GSM coder/ decoder



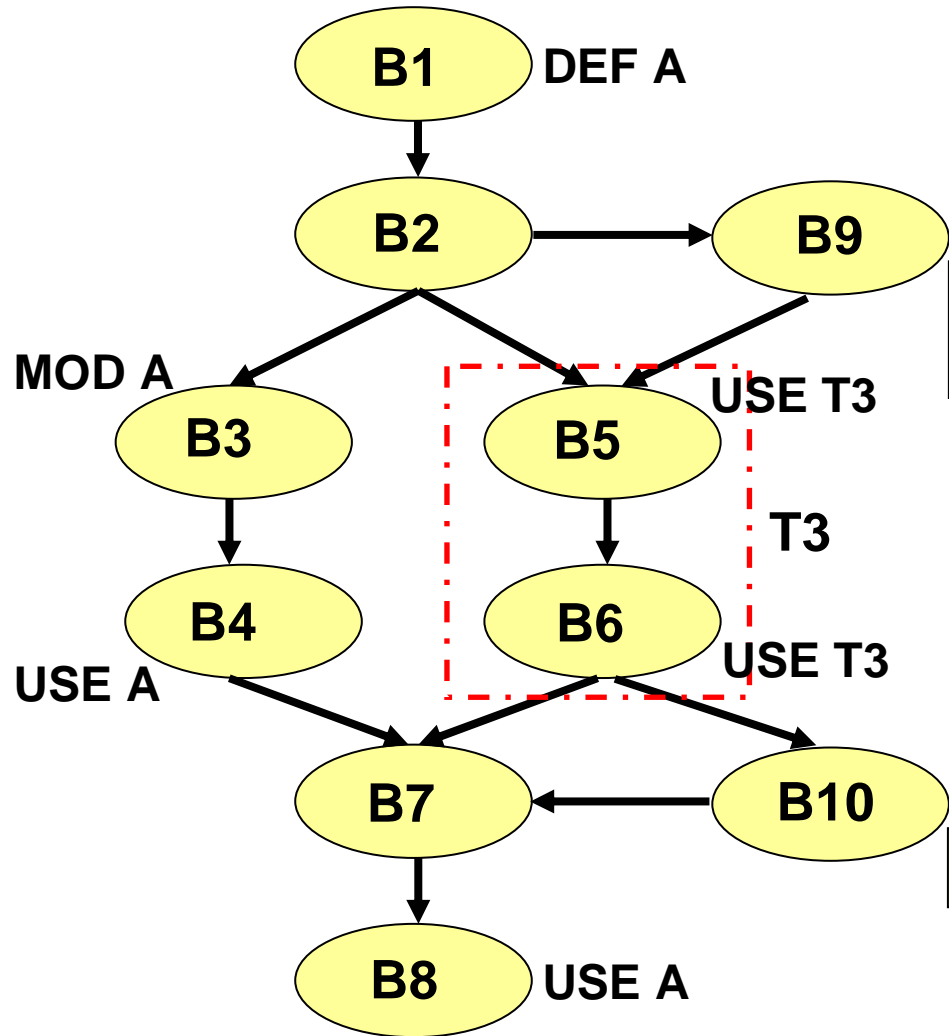
Dynamic replacement within scratch pad



- 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|

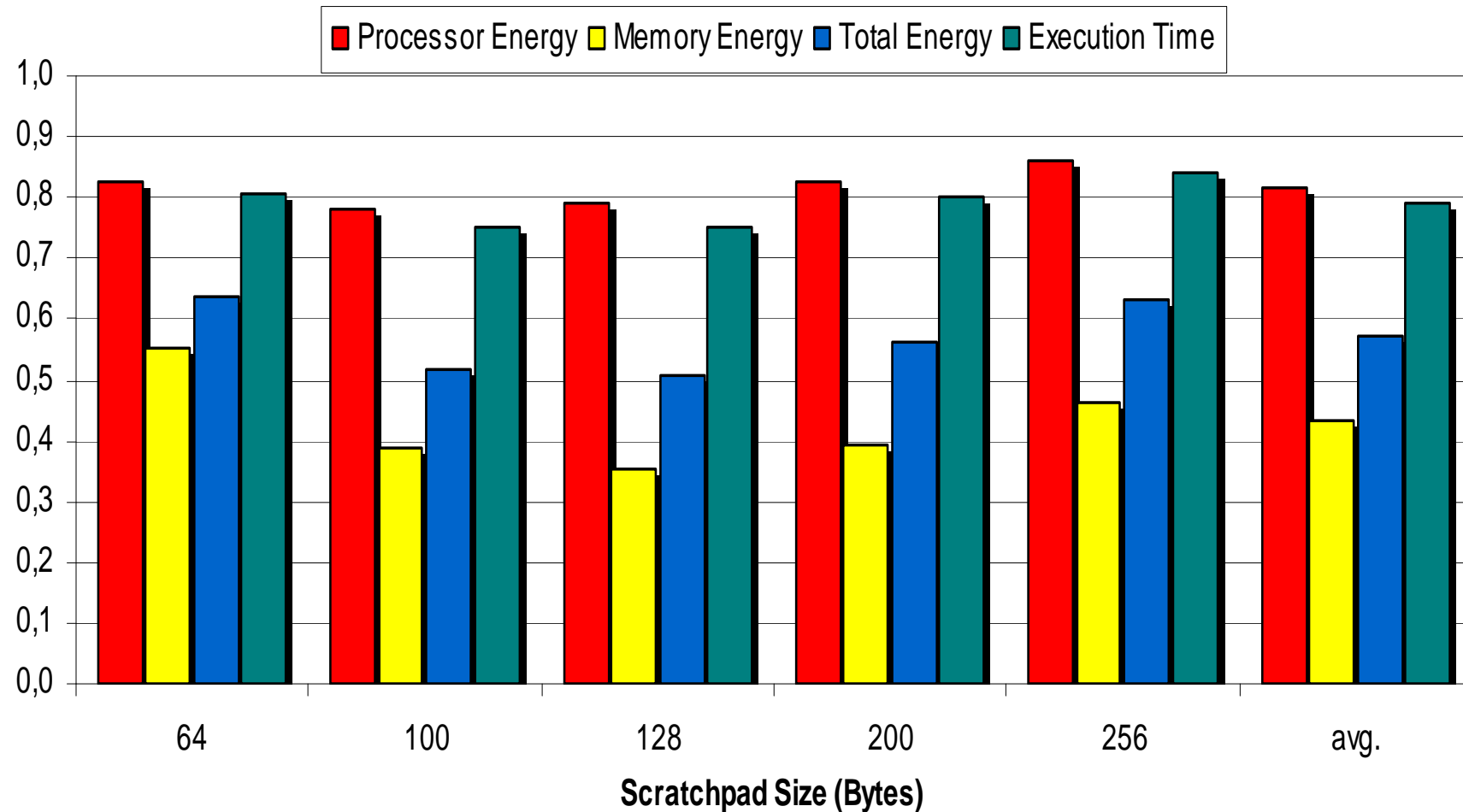
SPILL_STORE(A);
 SPILL_LOAD(T3);

Solution:
 A → SP & T3 → SP

SPILL_LOAD(A);

Dynamic replacement within SPM

Edge detection relative to static allocation



Hardware-support for block-copying



The DMA unit was modeled in VHDL, simulated, synthesized. Unit only makes up 4% of the processor chip.

The unit can be put to sleep when it is unused.

Code size reductions of up to **23%** for a 256 byte SPM were determined using the DMA unit instead of the overlaying allocation that uses processor instructions for copying.

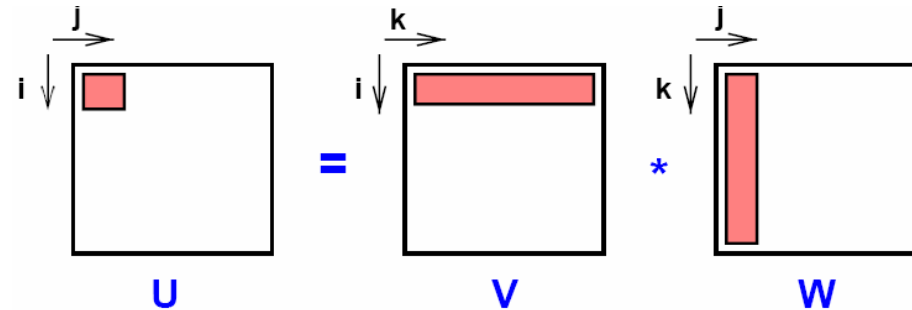
[Lars Wehmeyer, Peter Marwedel: Fast, Efficient and Predictable Memory Accesses, *Springer*, 2006]

References to large arrays (1)

- Regular accesses -

```

for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    for (k=0; k<n; k++)
      U[i][j]=U[i][j] + V[i][k] * W[k][j]
  
```



Tiling

```

for (it=0; it<n; it=it+Sb)
  {read_tile V[it:it+Sb-1, 1:n]
  for (jt=0; jt<n; jt=jt+Sb)
    {read_tile U[it:it+Sb-1, jt:jt+Sb-1];
    read_tile W[1:n,jt:jt+Sb-1];
    U[it:it+Sb-1,jt:jt+Sb-1]=U[it:it+Sb-1,jt:jt+Sb-1]
    + V[it:it+Sb-1,1:n]
    * W [1:n, jt:jt+Sb-1];
    write_tile U[it:it+Sb-1,jt:jt+Sb-1]
  }
}
  
```

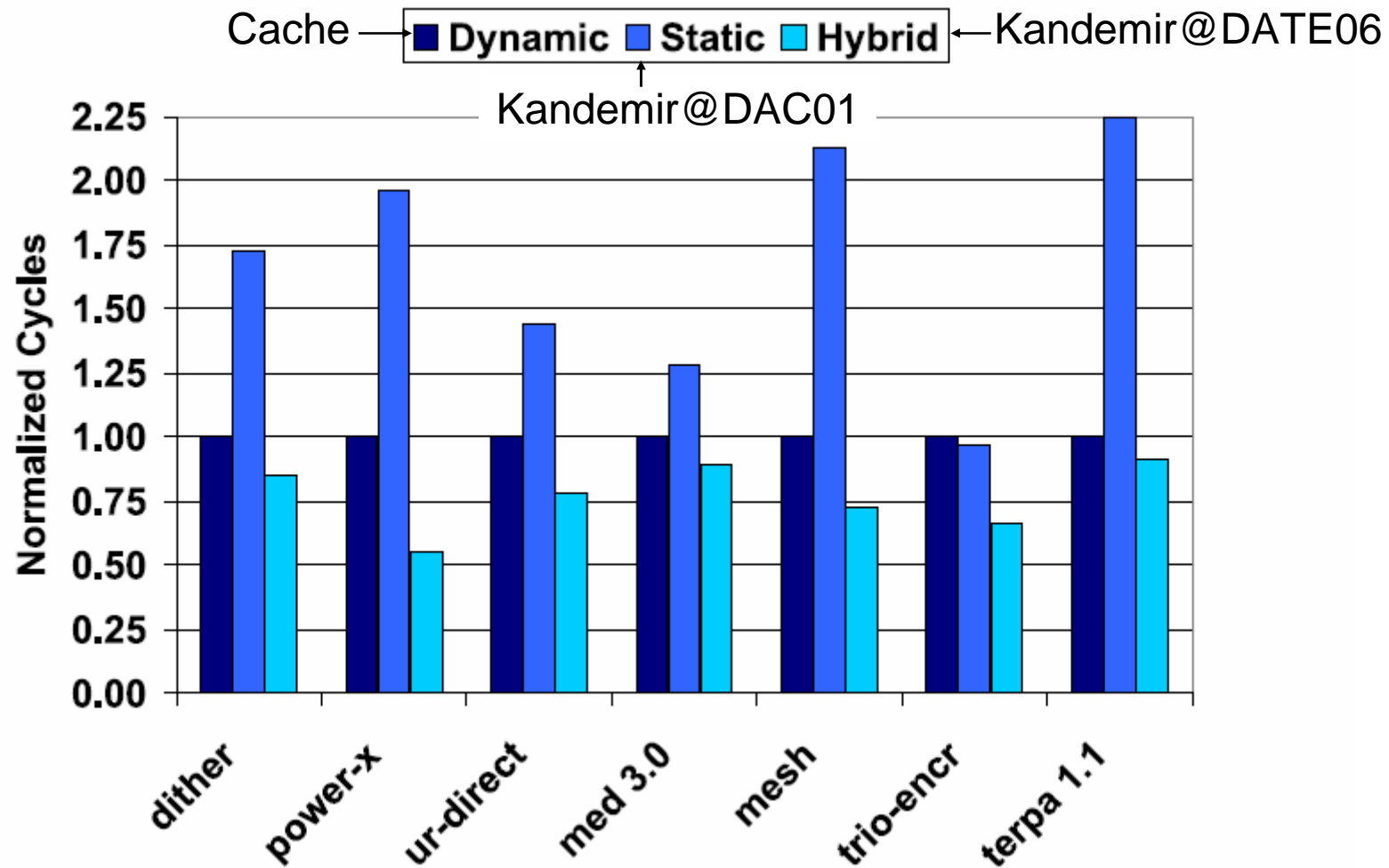
[M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, A. Parikh: Dynamic Management of Scratch-Pad Memory Space, *DAC*, 2001, pp. 690-695]

References to large arrays - Irregular accesses -

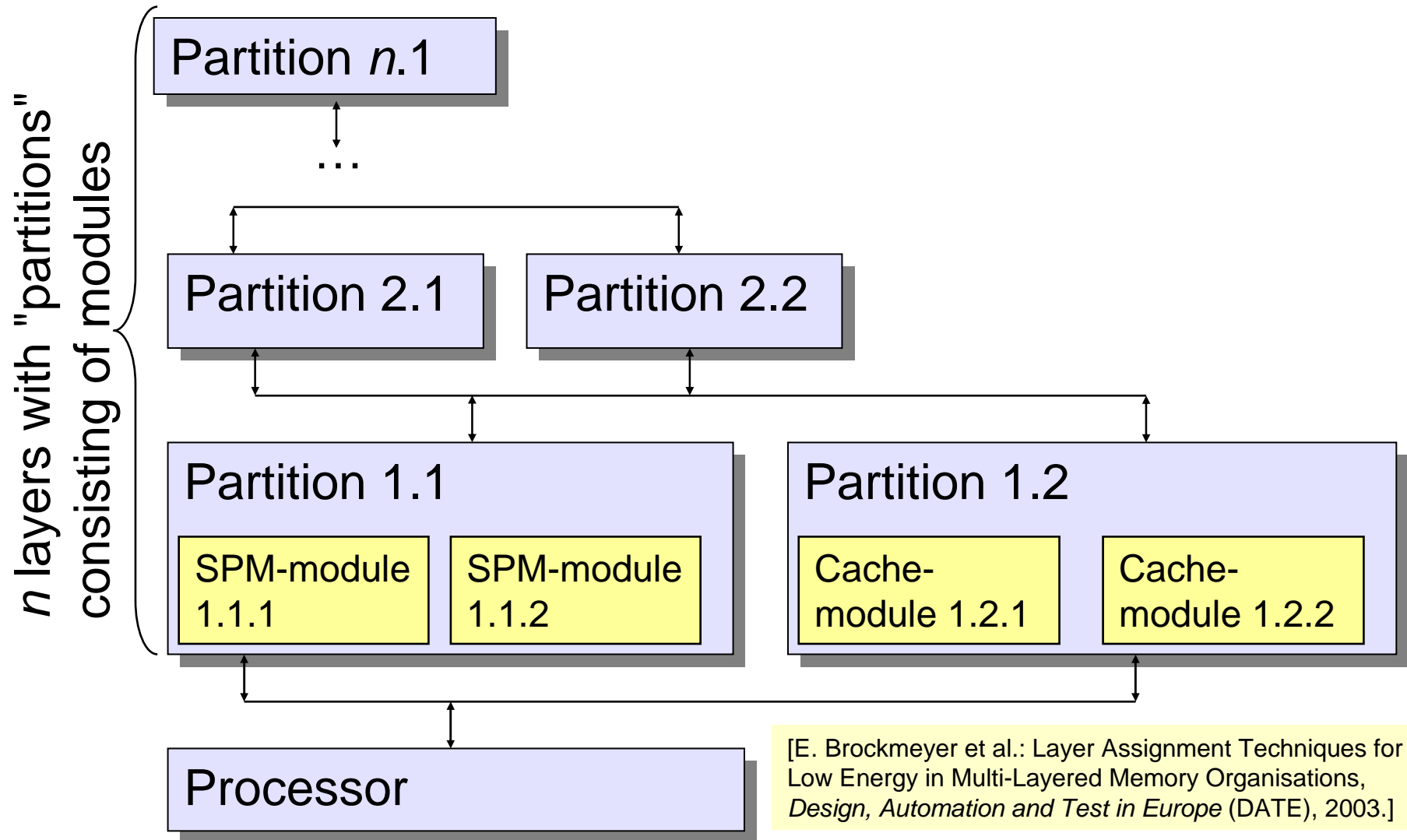
```
for each loop nest  $L$  in program  $P$  {  
  apply loop tiling to  $L$  based on the access patterns of  
  regular array references;  
  for each assignment to index array  $X$   
    update the block minimum and maximum values of  $X$ ;  
  compute the set of array elements that are irregularly  
  referenced in the current inter-tile iteration;  
  compare the memory access costs for using  
  and not using SPM;  
  if (using SPM is beneficial)  
    execute the intra-tile loop iterations by using the SPM  
  else  
    execute the intra-tile loop iterations by not  
    using the SPM  
}
```

[G. Chen, O. Ozturk, M. Kandemir, M. Karakoy: Dynamic Scratch-Pad Memory Management for Irregular Array Access Patterns, *DATE*, 2006]

Results for irregular approach



Hierarchical memories: Memory hierarchy layer assignment (MHLA) (IMEC)



Memory hierarchy layer assignment (MHLA)

- Copy candidates -

```
int A[250]
for (i=0; i<10; i++)
  for (j=0; j<10; j++)
    for (k=0; k<10; k++)
      for (l=0; l<10; l++)
        f(A[j*10+l])
size=0; reads(A)=10000
```

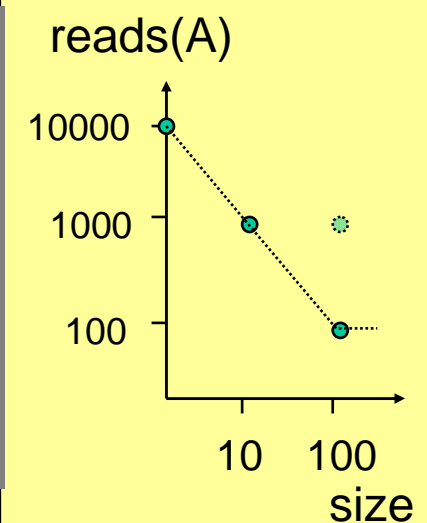
```
int A[250]
for (i=0; i<10; i++)
  for (j=0; j<10; j++)
    {A"[0..9]=A[j*10..j*10+9];
     for (k=0; k<10; k++)
       for (l=0; l<10; l++)
         f(A"[l])}
size=10; reads(A)=1000
```

Copy candidate

A', A" in small memory

```
int A[250]
for (i=0; i<10; i++)
  {A'[0..99]=A[0..99];
   for (j=0; j<10; j++)
     for (k=0; k<10; k++)
       for (l=0; l<10; l++)
         f(A'[j*10+l])}
size=100; reads(A)=1000
```

```
int A[250]
A'[0..99]=A[0..99];
for (i=0; i<10; i++)
  for (j=0; j<10; j++)
    for (k=0; k<10; k++)
      for (l=0; l<10; l++)
        f(A'[j*10+l])
size=100; reads(A)=100
```

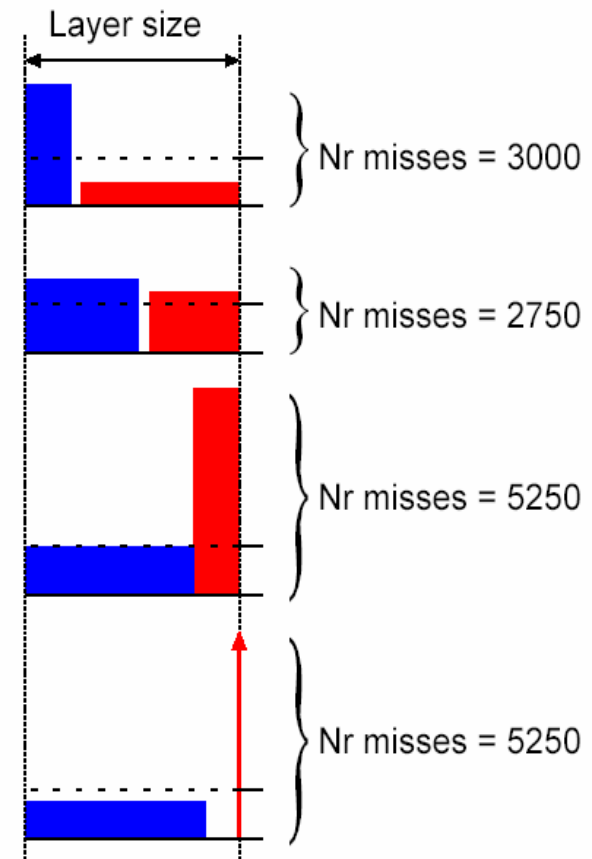
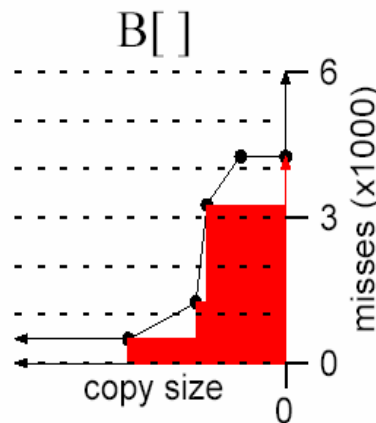
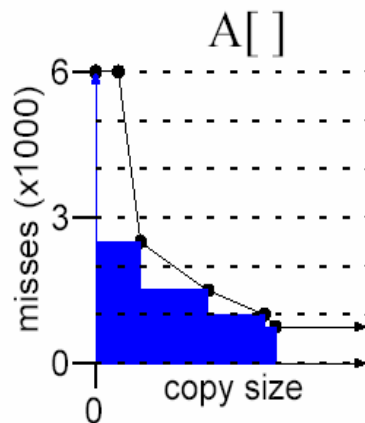


Memory hierarchy layer assignment (MHLA)

- Goal -

Goal: For each variable: find permanent layer, partition and module & select copy candidates such that energy is minimized.

Conflicts between variables



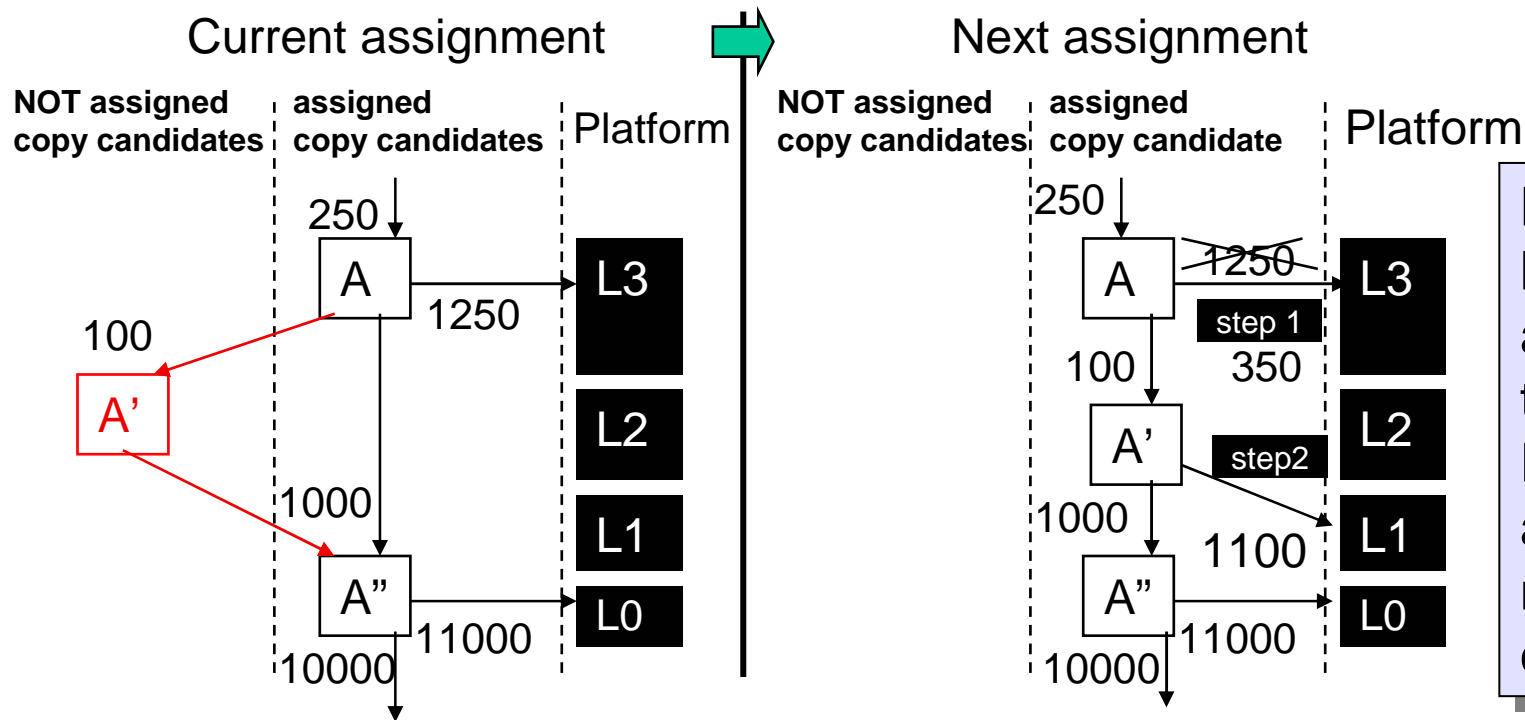
[E. Brockmeyer et al.: Layer Assignment Techniques for Low Energy in Multi-Layered Memory Organisations, *Design, Automation and Test in Europe (DATE)*, 2003.]

Memory hierarchy layer assignment (MHLA)

- Approach -

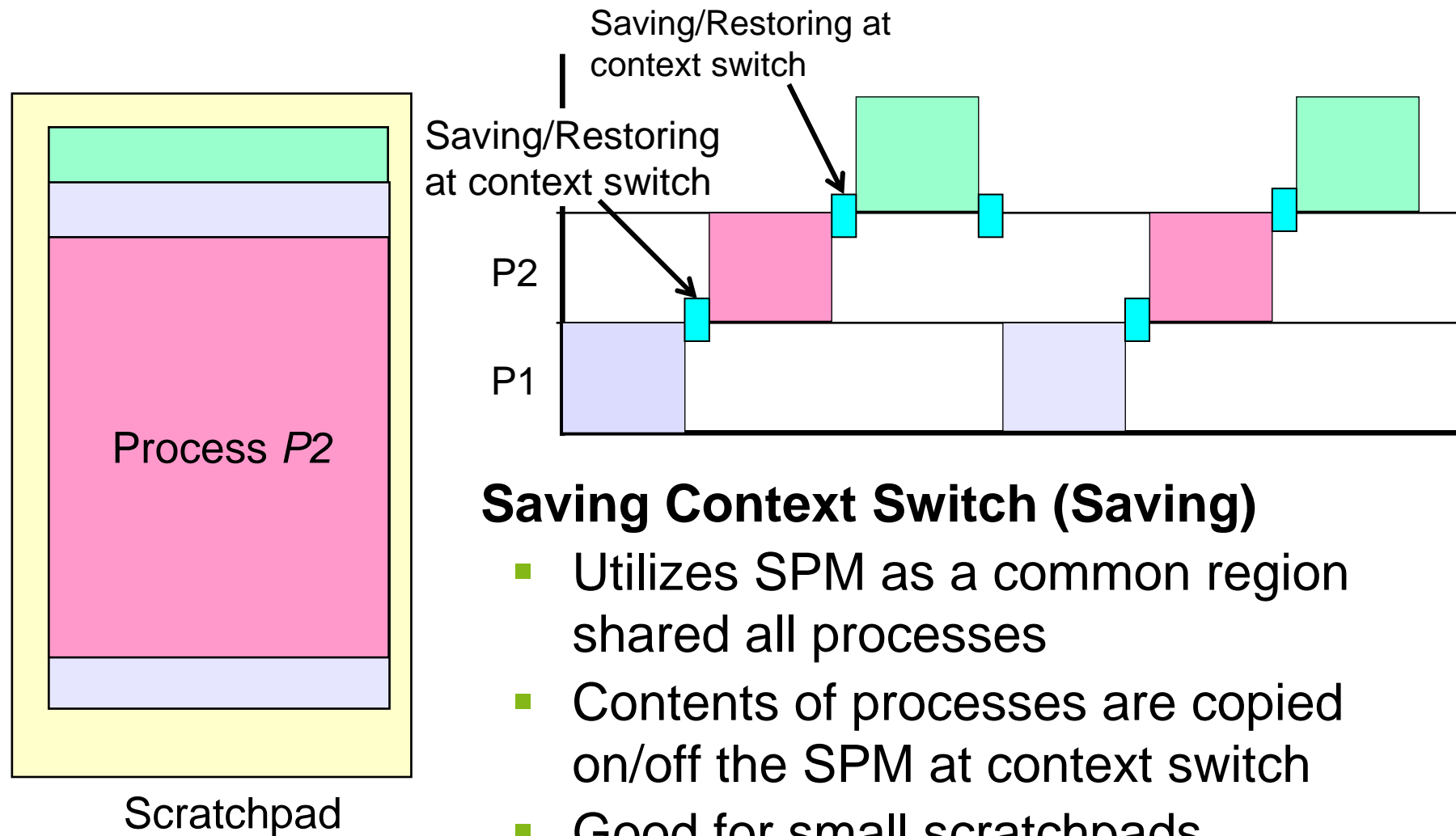
Approach:

- start with initial variable allocation
- incrementally improve initial solution such that total energy is minimized.



More general hardware architecture than the Dortmund approach, but no global optimization.

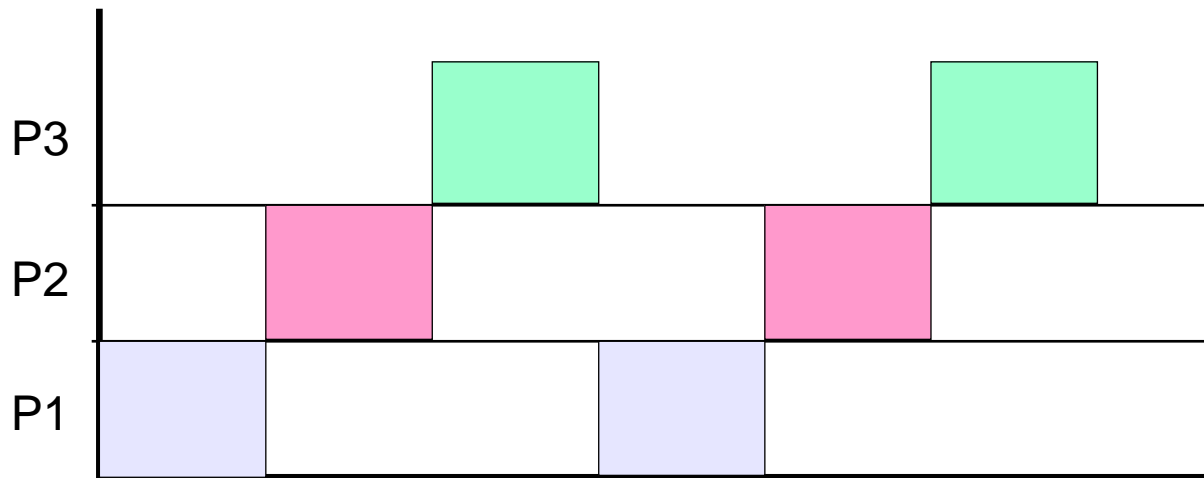
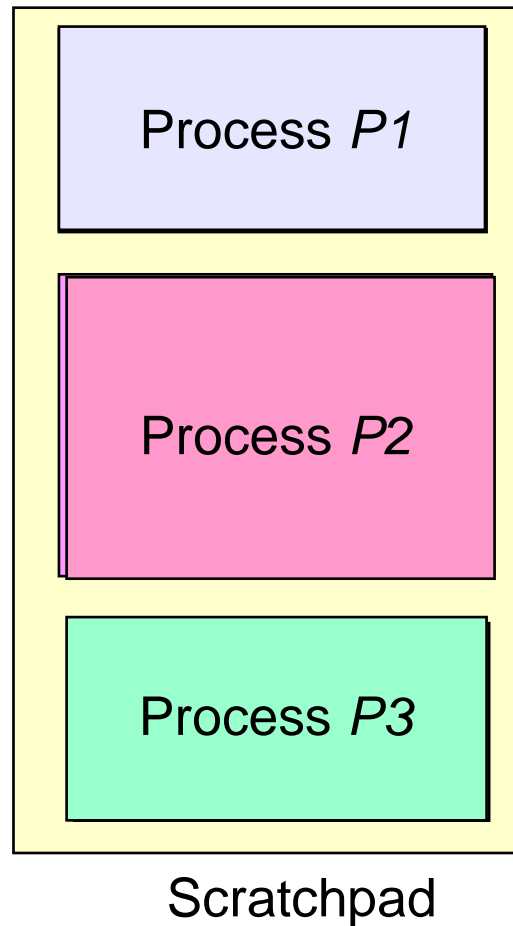
Saving/Restoring Context Switch



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

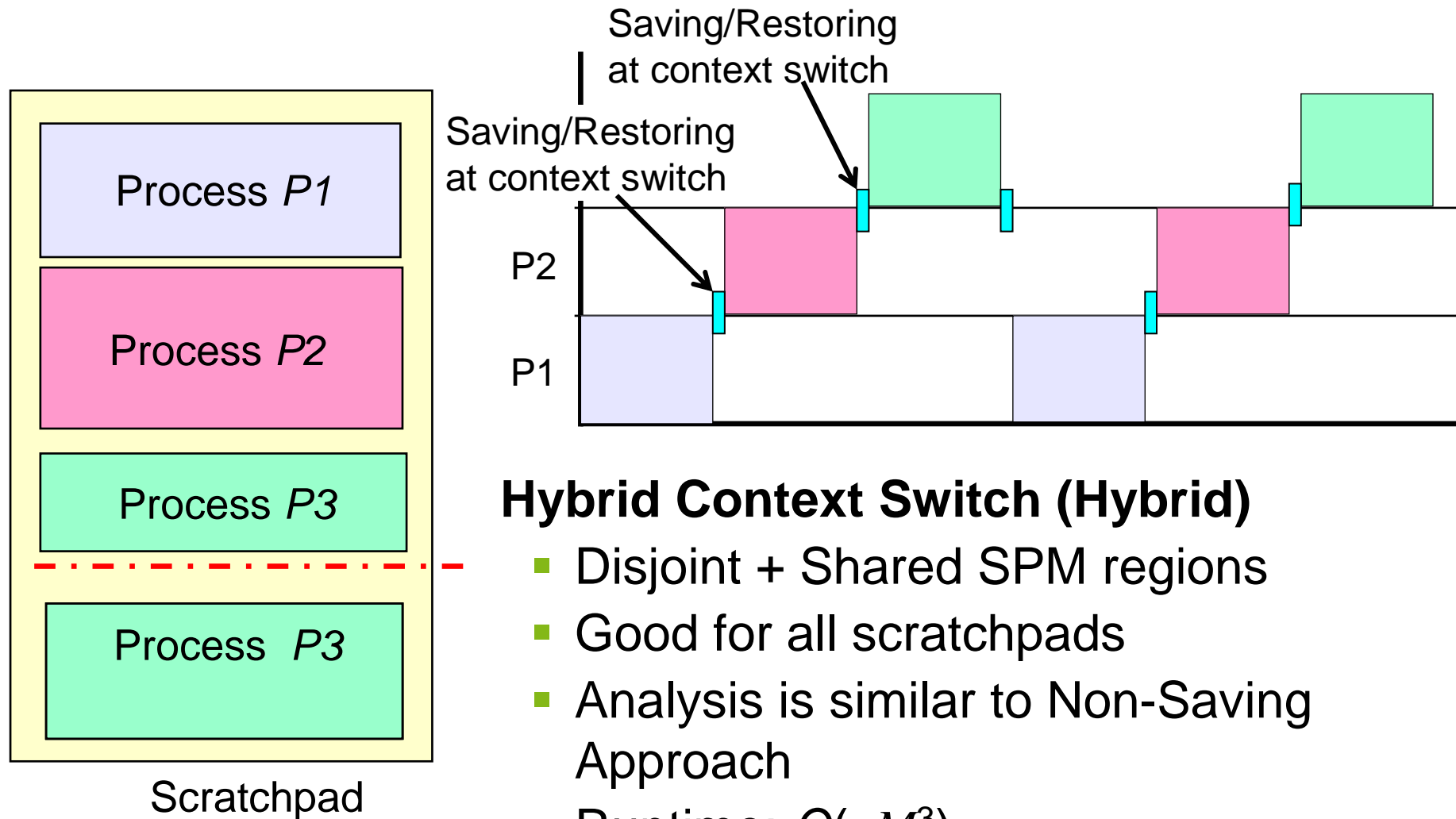
Non-Saving Context Switch



Non-Saving Context Switch

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

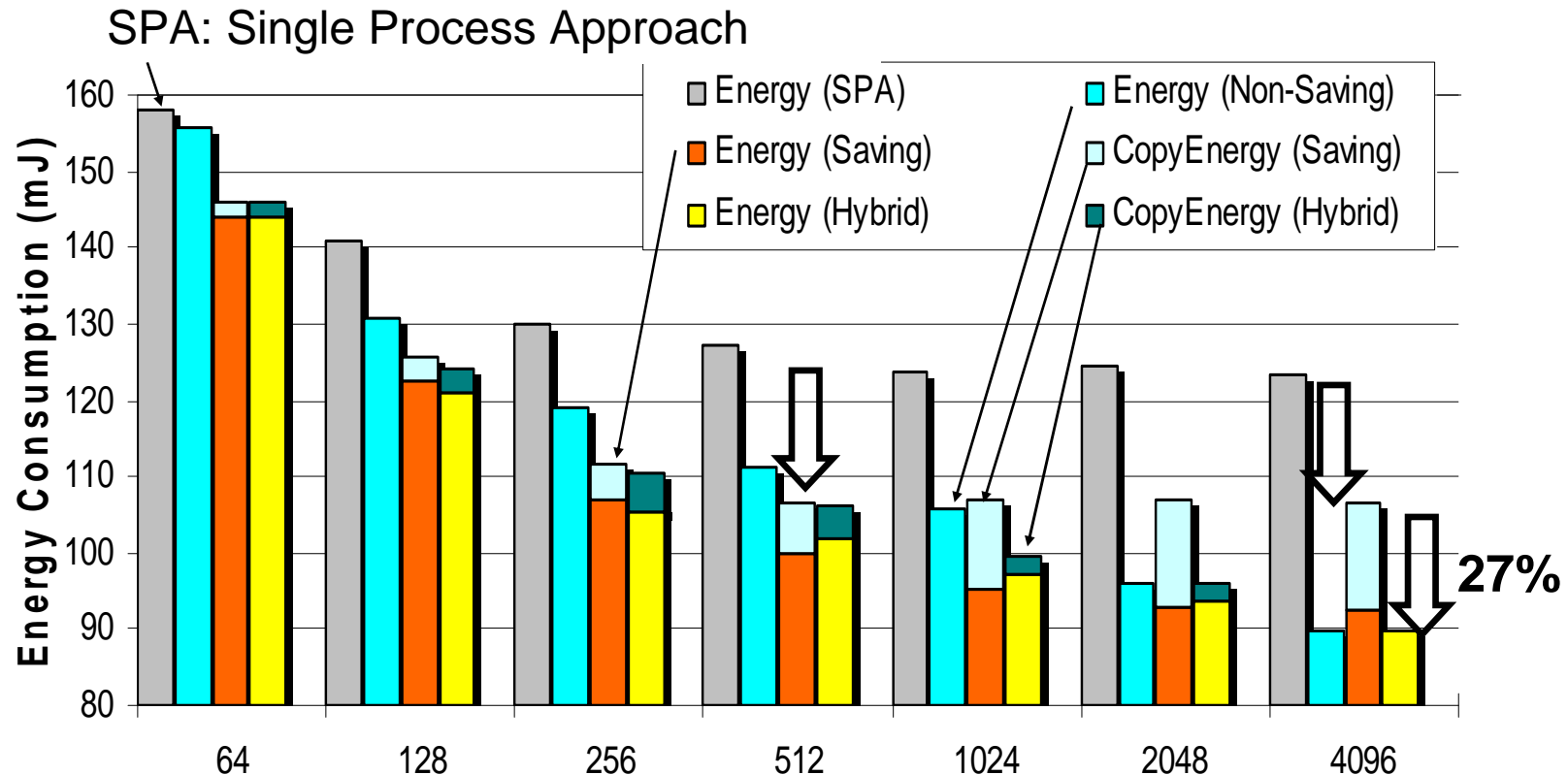
Hybrid Context Switch



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

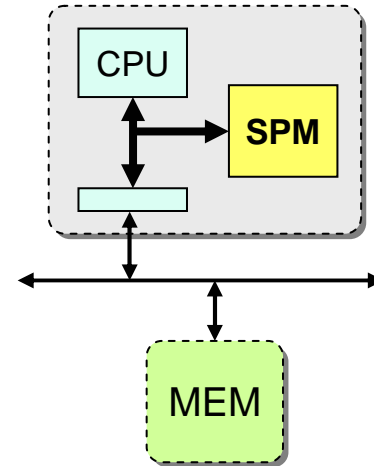
edge detection,
adpcm, g721, mpeg

Dynamic set of multiple applications

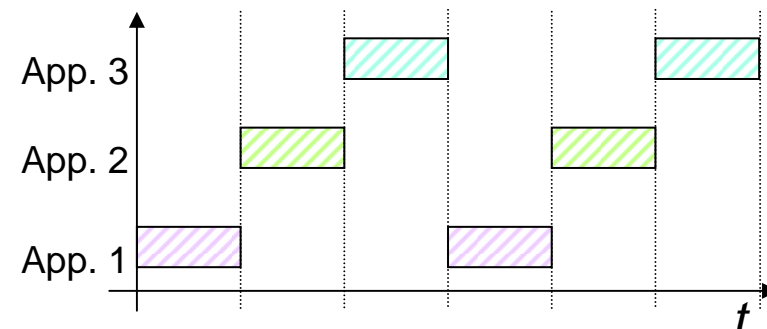
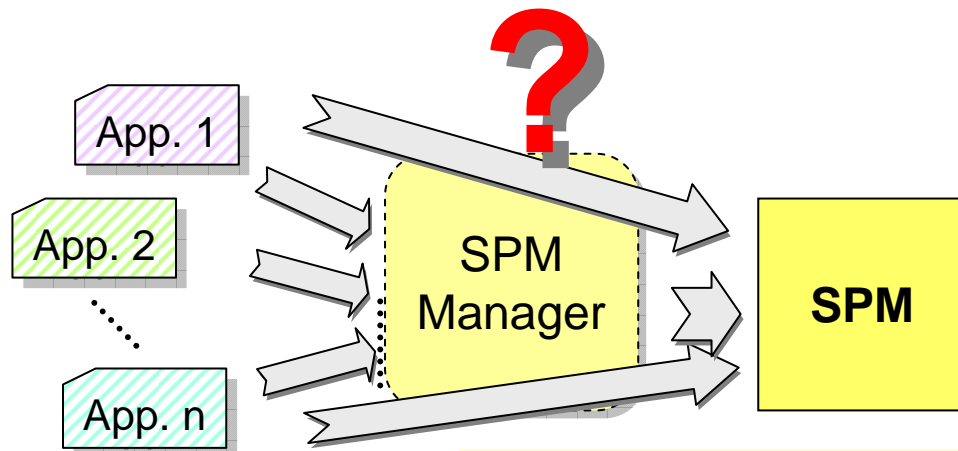
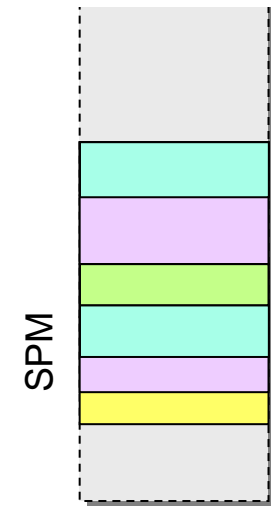
Compile-time partitioning of SPM no longer feasible

➔ Introduction of SPM-manager

- Runtime decisions, but compile-time supported



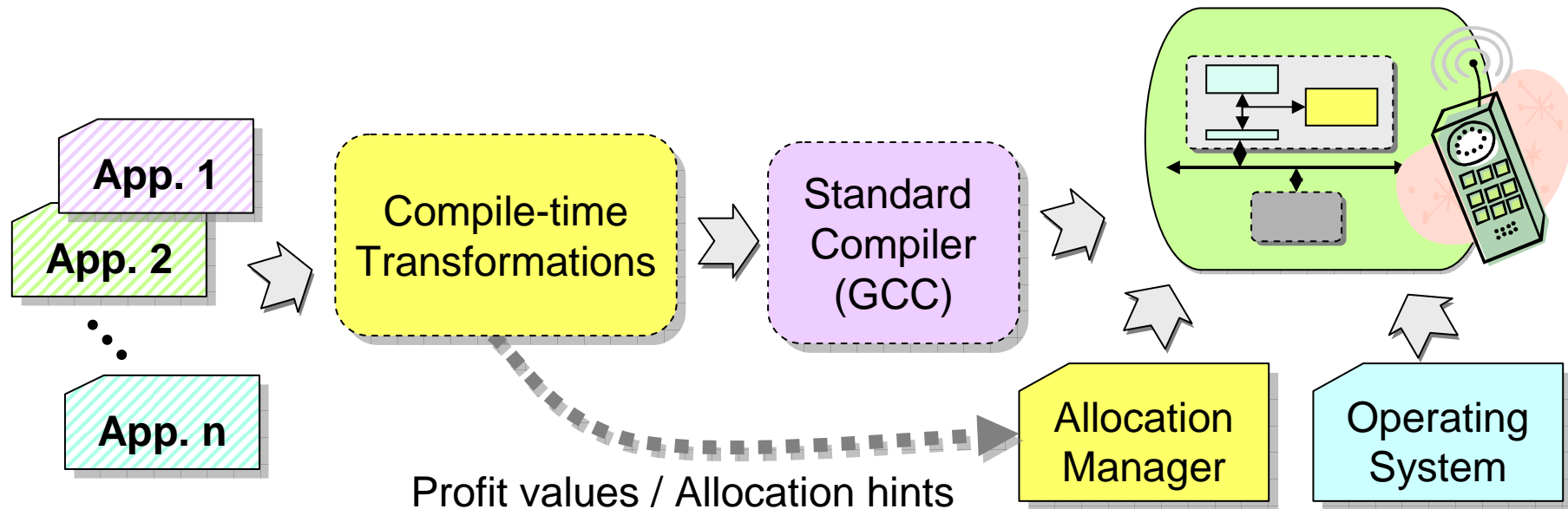
Address space:



[R. Pyka, Ch. Faßbach, M. Verma, H. Falk, P. Marwedel: Operating system integrated energy aware scratchpad allocation strategies for multi-process applications, *SCOPES*, 2007]

Approach overview

- 2 steps: compile-time analysis & runtime decisions
- No need to know all applications at compile-time
- Capable of managing runtime allocated memory objects
- Integrates into an embedded operating system

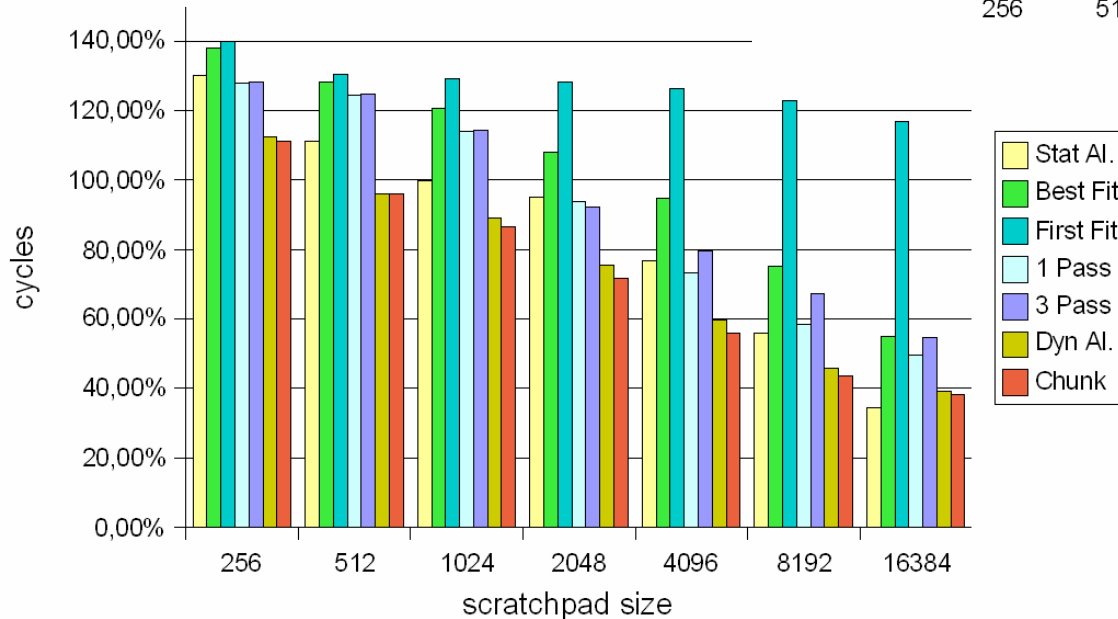
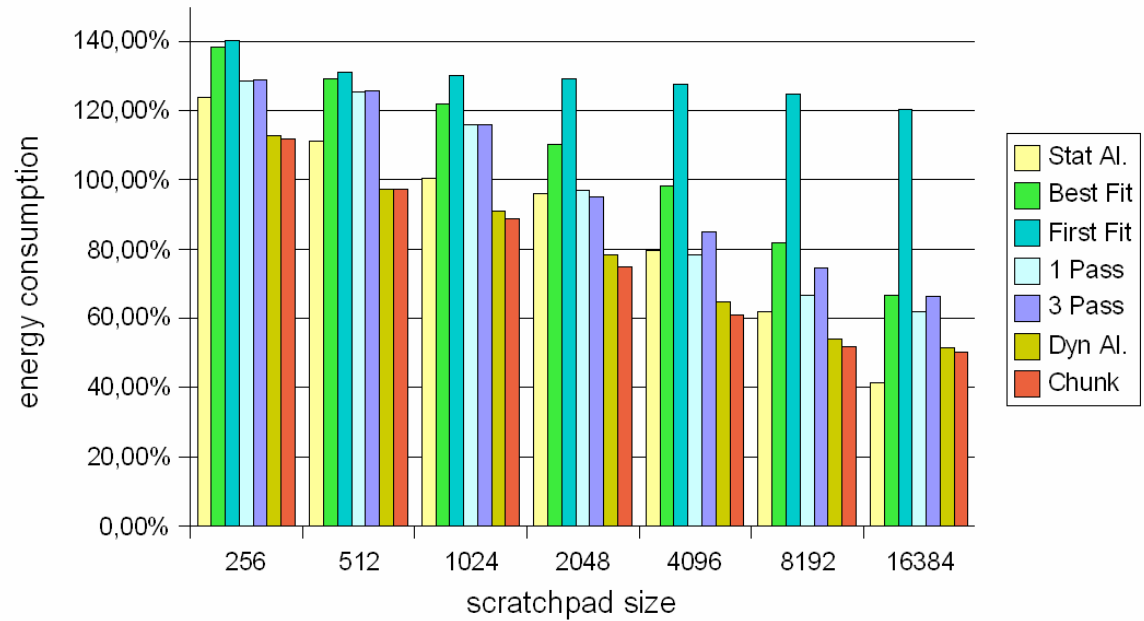


Using MPArm simulator from U. Bologna

Results

► MEDIA+ Energy

- **Baseline: Main memory only**
- **Best: Static for 16k → 58%**
- **Overall best: Chunk → 49%**



► MEDIA+ Cycles

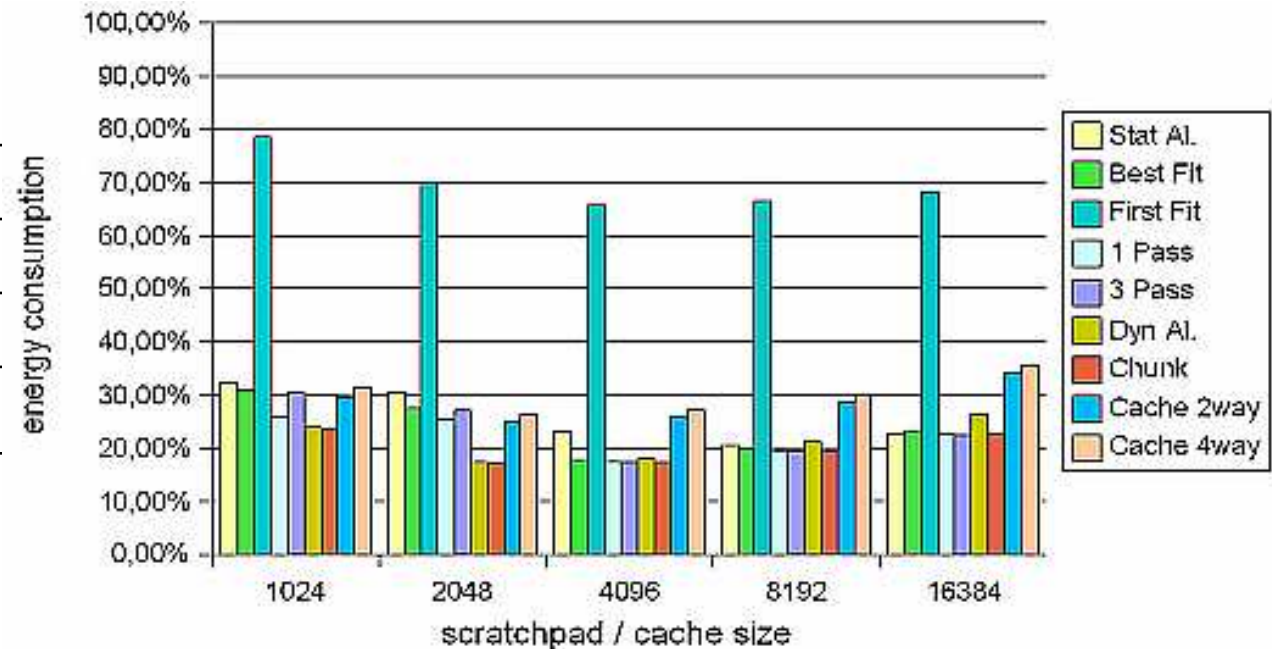
- **Baseline: Main memory only**
- **Best: Static for 16k → 65%**
- **Overall best: Chunk → 61%**

Comparison of SPMM to Caches for SORT

- Baseline: Main memory only
- SPMM peak energy reduction by 83% at 4k Bytes scratchpad
- Cache peak: 75% at 2k 2-way cache
- SPMM capable of outperforming caches
- OS and libraries are not considered yet

Chunk allocation results:

SPM Size	Δ 4-way
1024	74,81%
2048	65,35%
4096	64,39%
8192	65,64%
16384	63,73%



Summary

Impact of memory architecture on execution times & energy consumption

- The SPM provides
 - Runtime efficiency, energy efficiency, timing predictability
- Allocation strategies
 - Static allocation
 - Partitioning
 - Timing predictability
 - Dynamic allocation
 - Tiling
 - Multiple hierarchy levels
 - Multiple processes
 - Dynamic sets of processes
- Savings dramatic, e.g. ~ 95% of the memory energy

