technische universität
dortmund

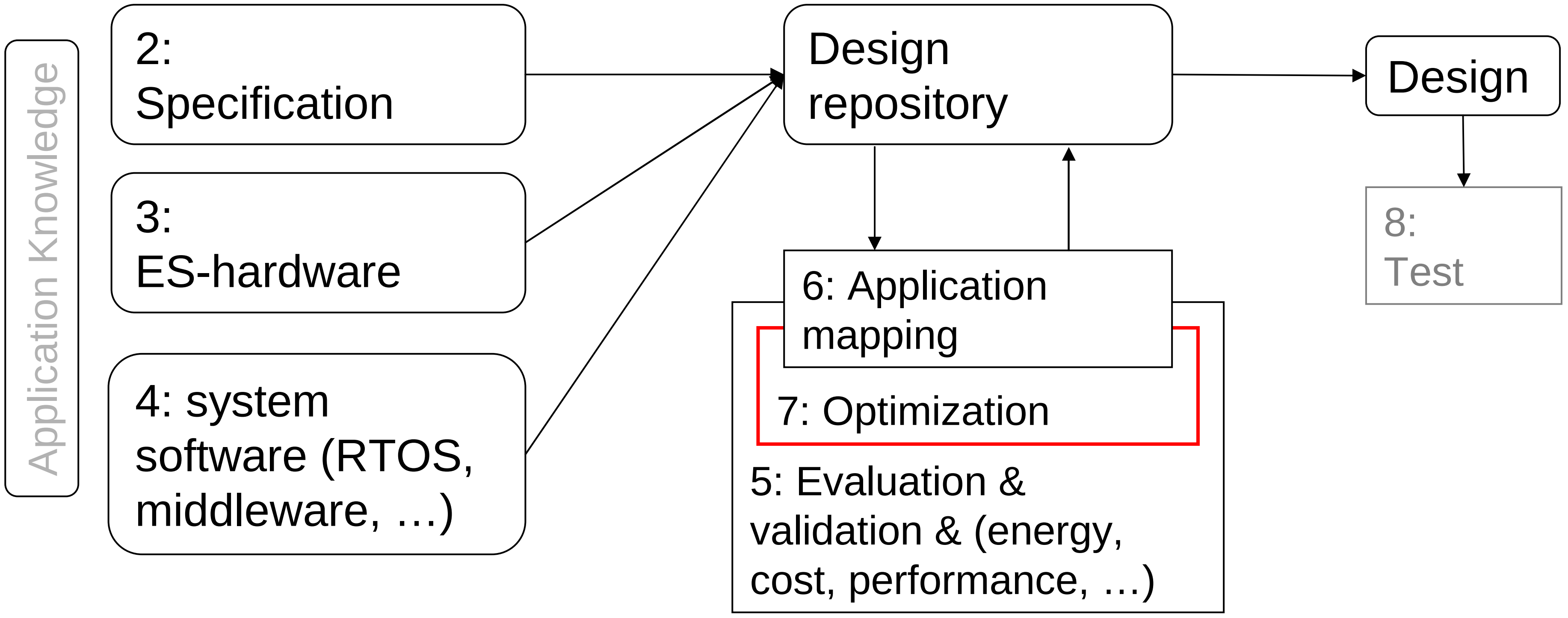
fakultät für informatik
informatik 12

Optimizations
- Compilation for Embedded Processors -

Peter Marwedel
TU Dortmund
Informatik 12
Germany

2011年 01 月 09 日

Structure of this course

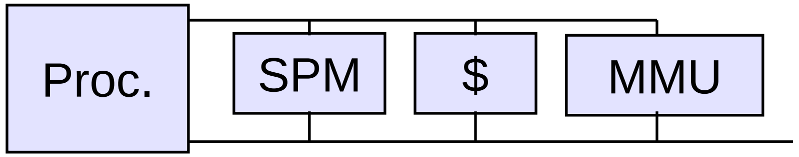


Numbers denote sequence of chapters

SPM+MMU (1)

How to use SPM in a system with virtual addressing?

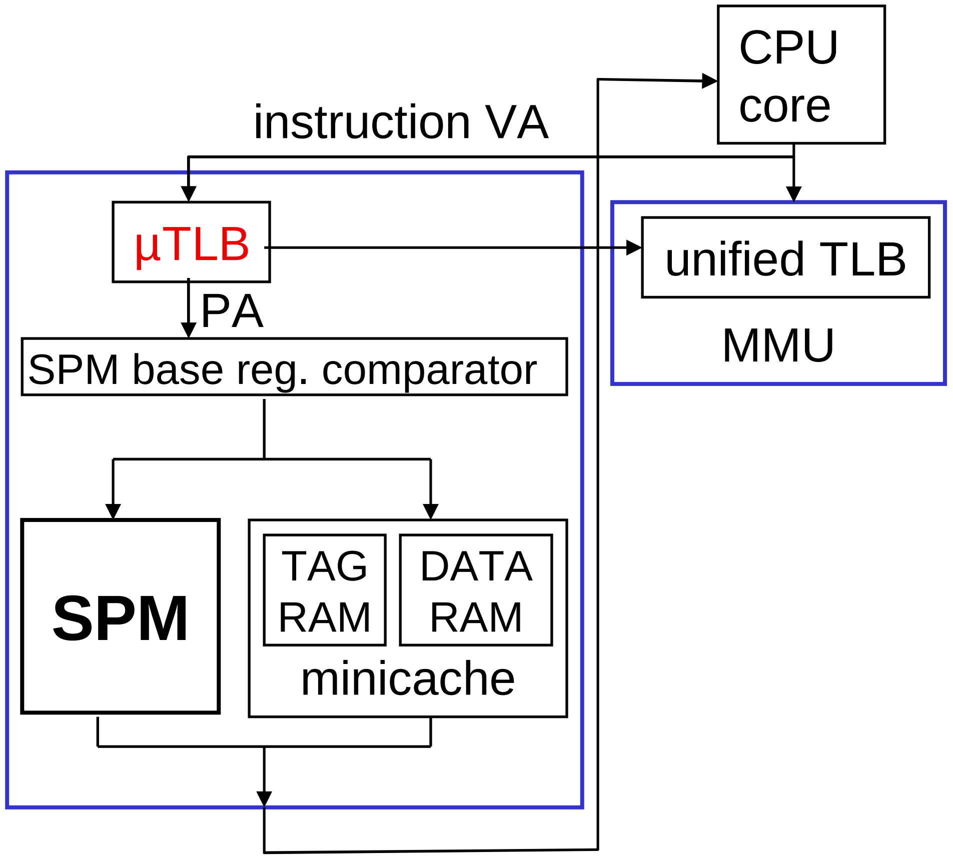
- **Virtual SPM**
  Typically accesses MMU
  + SPM in parallel
  ☞ not energy efficient
- **Real SPM**
  ☞ suffers from potentially long VA translation
- Egger, Lee, Shin (Seoul Nat. U.): Introduction of small **µTLB** translating recent addresses fast.



[B. Egger, J. Lee, H. Shin: Scratchpad memory management for portable systems with a memory management unit, *CASES*, 2006, p. 321-330 (best paper)]

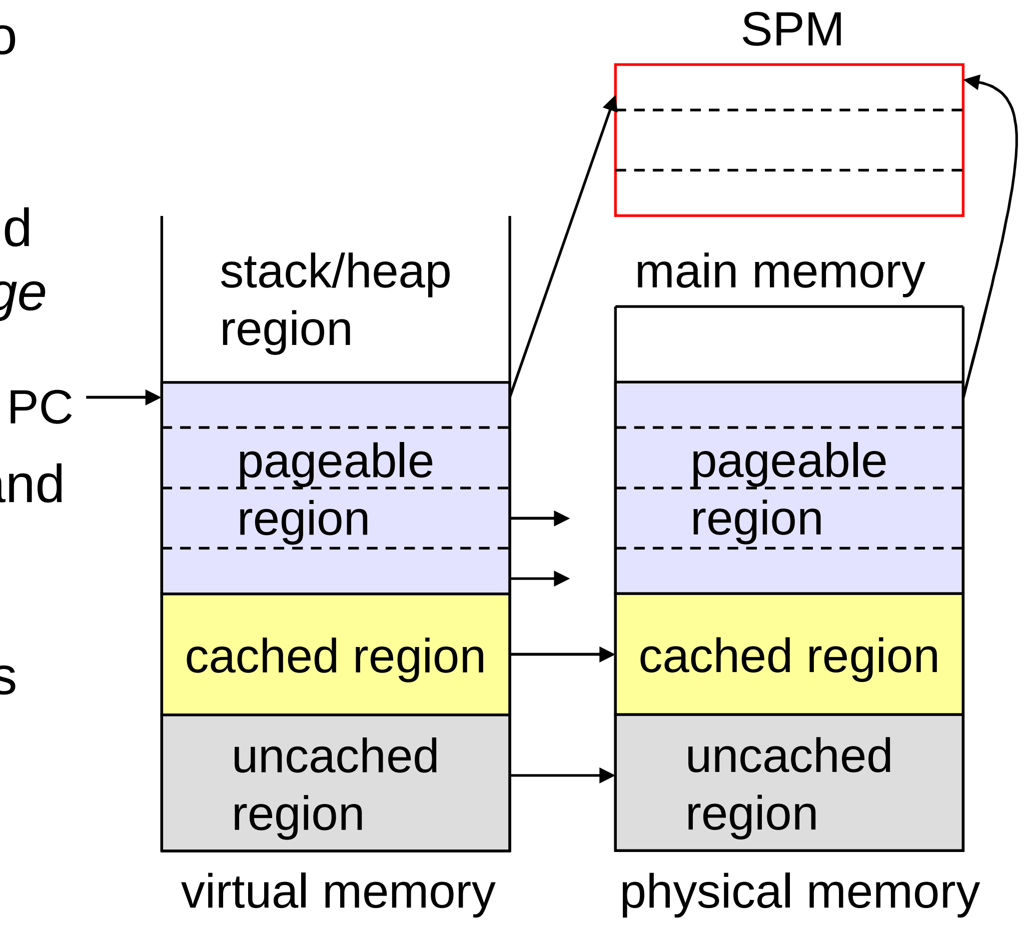
SPM+MMU (2)

- µTLB generates physical address in 1 cycle
- if address corresponds to SPM, it is used
- otherwise, mini-cache is accessed
- Mini-cache provides reasonable performance for non-optimized code
- µTLB miss triggers main TLB/MMU
- SPM is used only for instructions
- instructions are stored in pages
- pages are classified as cacheable, non-cacheable, and “pageable” (= suitable for SPM)

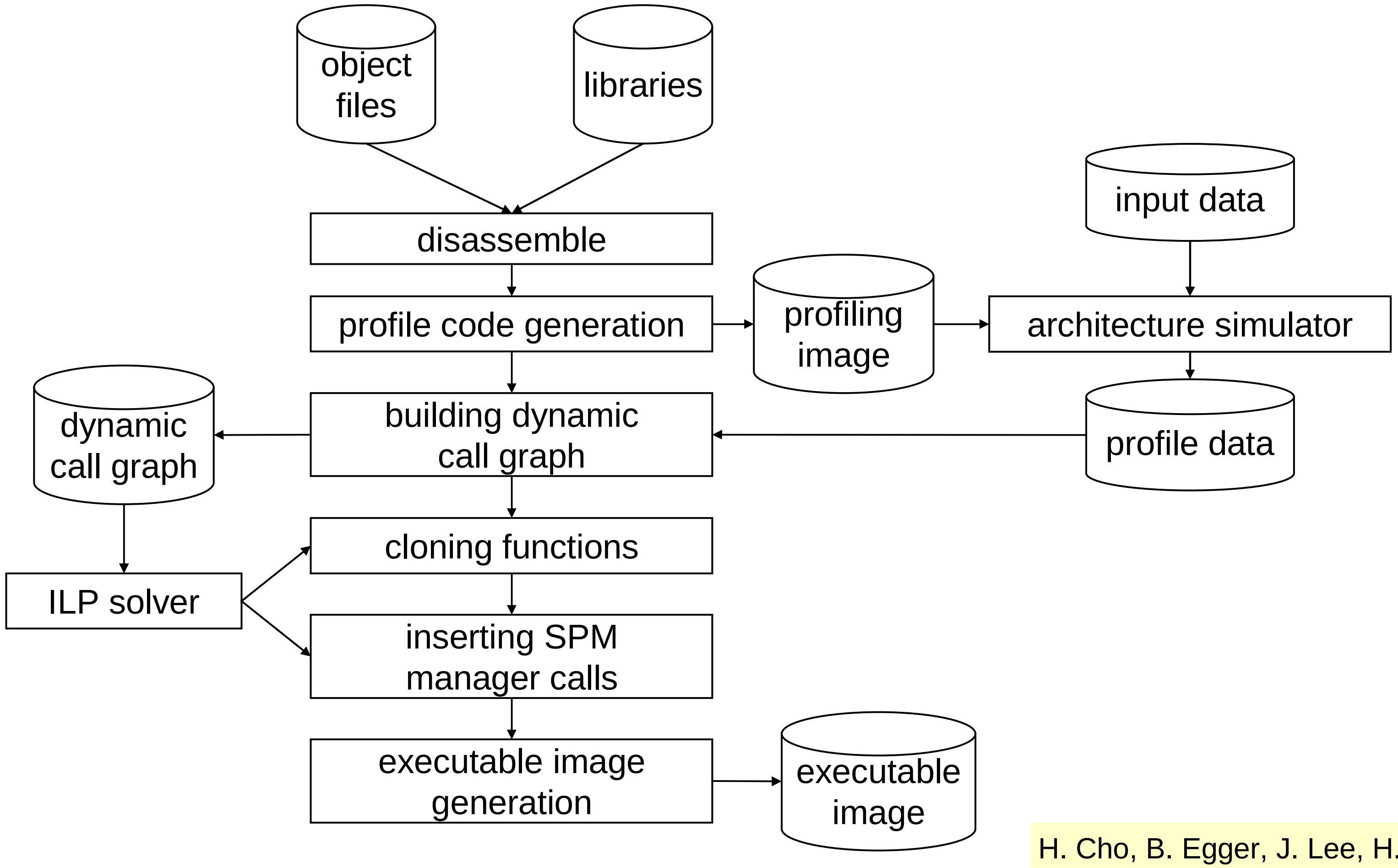


SPM+MMU (3)

- Application binaries are modified: frequently executed code put into pageable pages.
- Initially, page-table entries for pageable code are marked invalid
- If invalid page is accessed, a *page table exception* invokes SPM manager (SPMM).
- SPMM allocates space in SPM and sets page table entry
- If SPMM detects more requests than fit into SPM, SPM eviction is started
- Compiler does not need to know SPM size

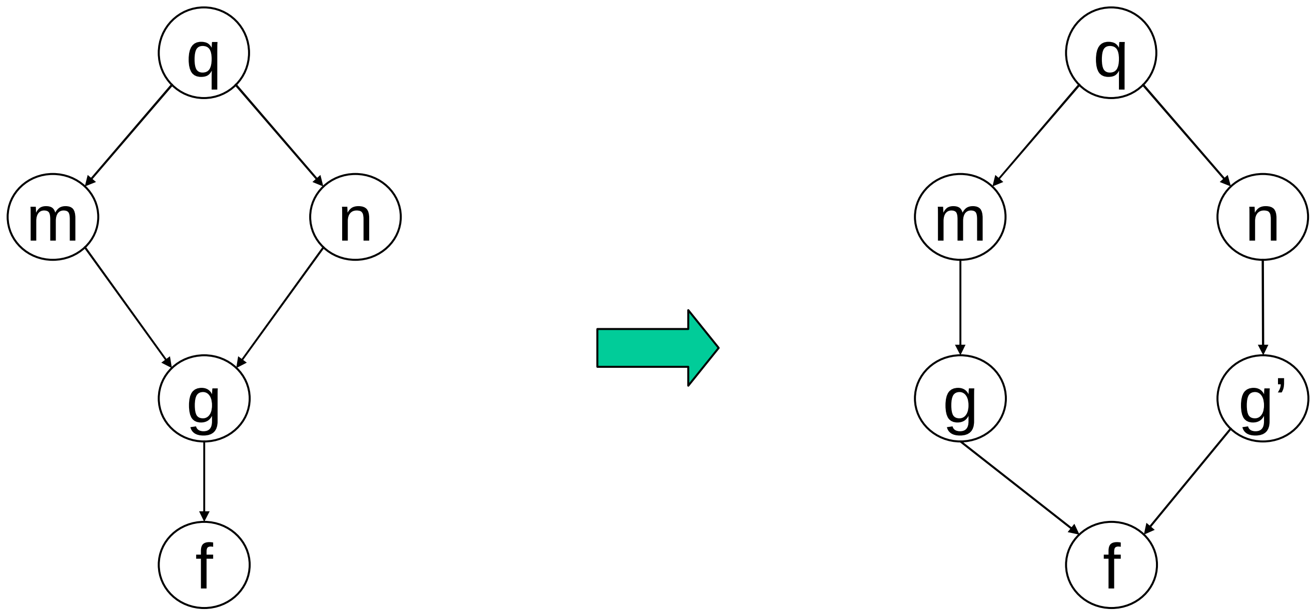


Extension to SNACK-pop (post-pass optimization)



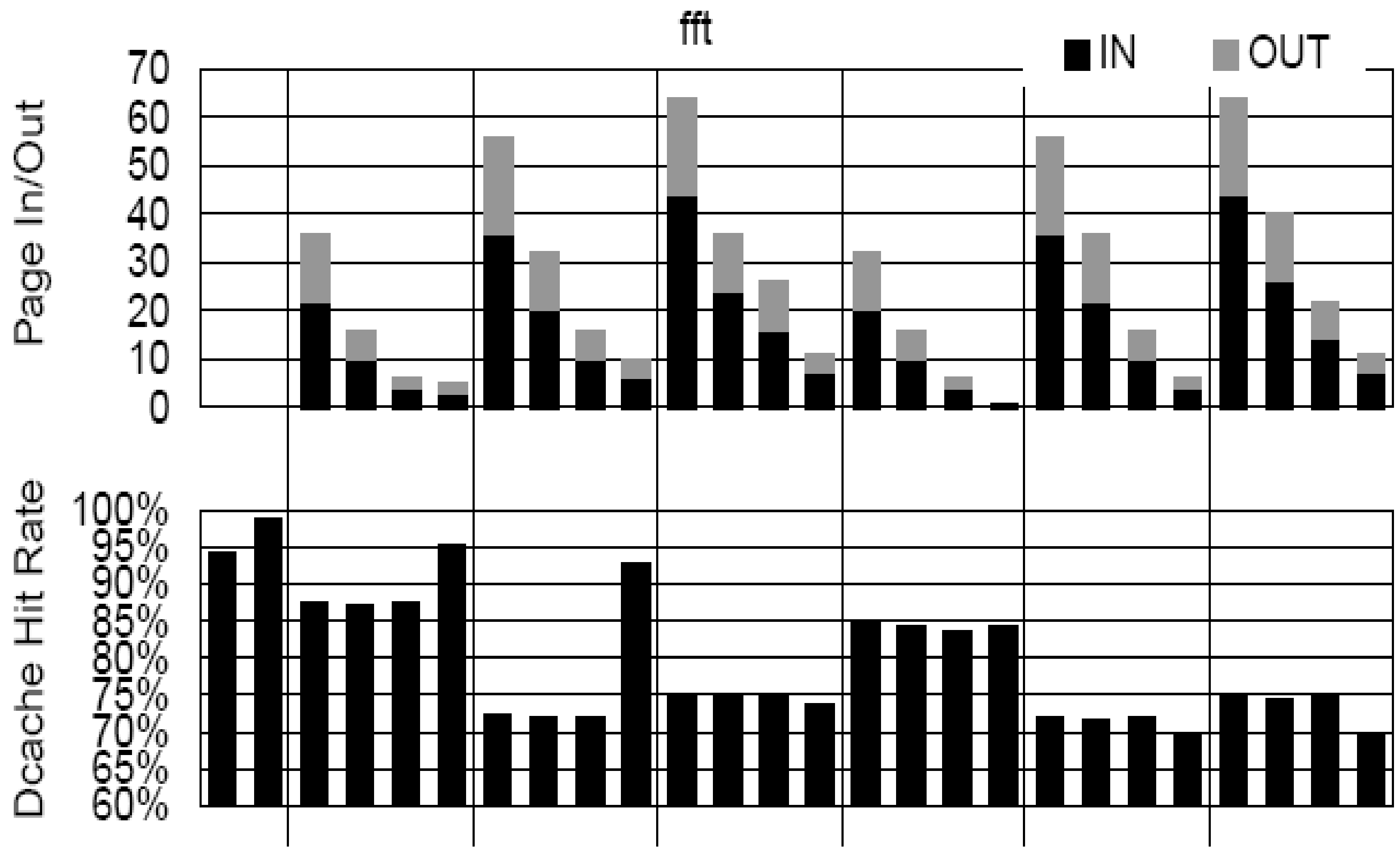
H. Cho, B. Egger, J. Lee, H. Shin: Dynamic Data Scratchpad Memory Management for a Memory Subsystem with an MMU, LCTES, 2007

Cloning of functions

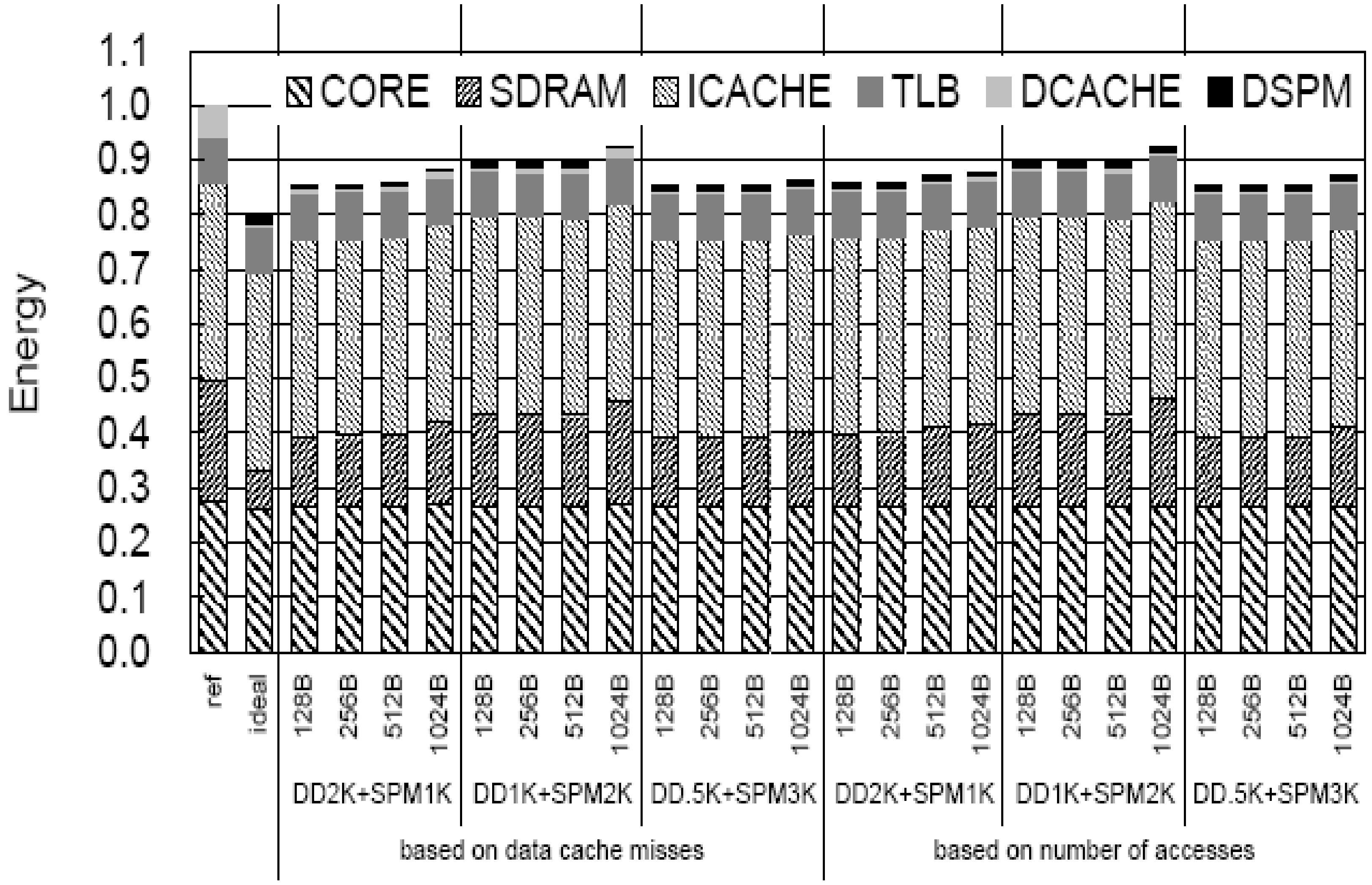


- Computation of which block should be moved in and out for a certain edge
- Generation of an ILP
- Decision about copy operations at compile time.

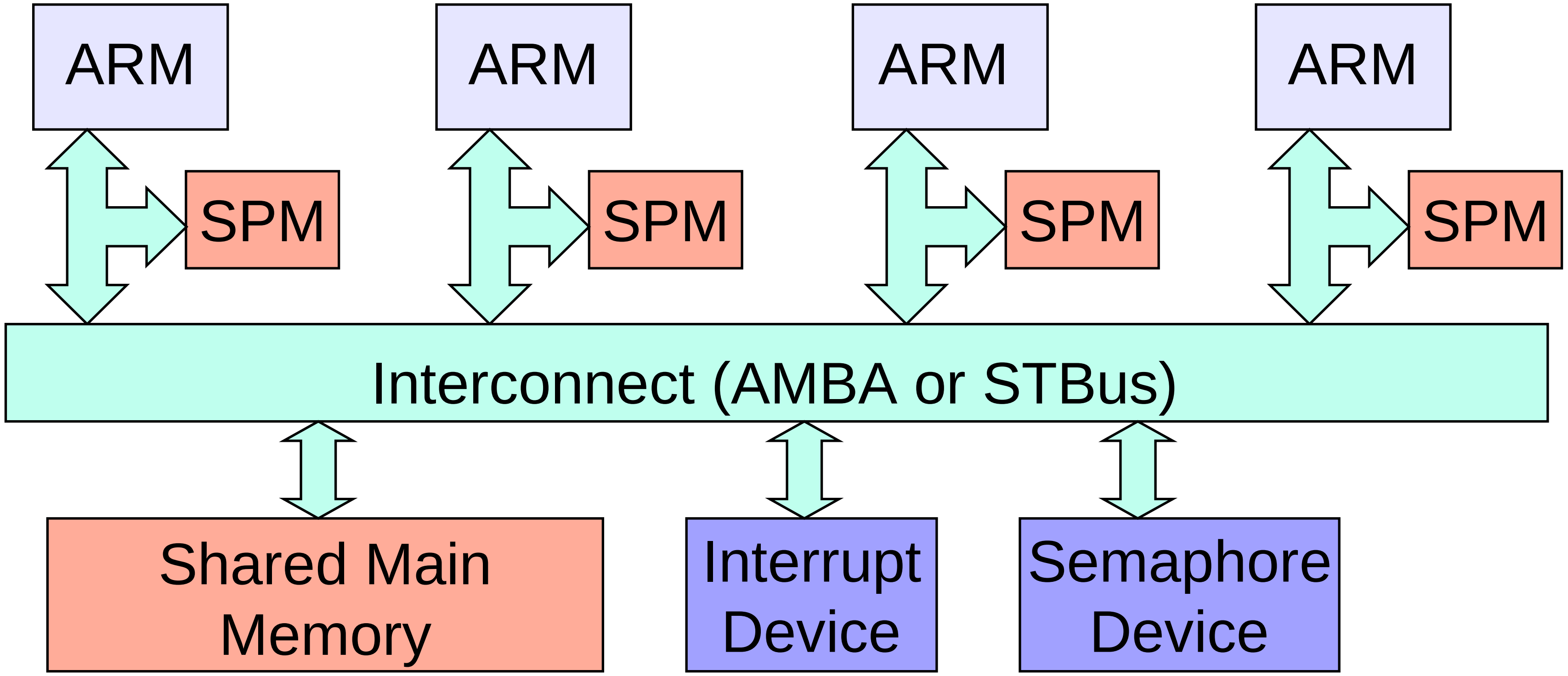
Results for SNACK-pop (1)



Results for SNACK-pop (2)

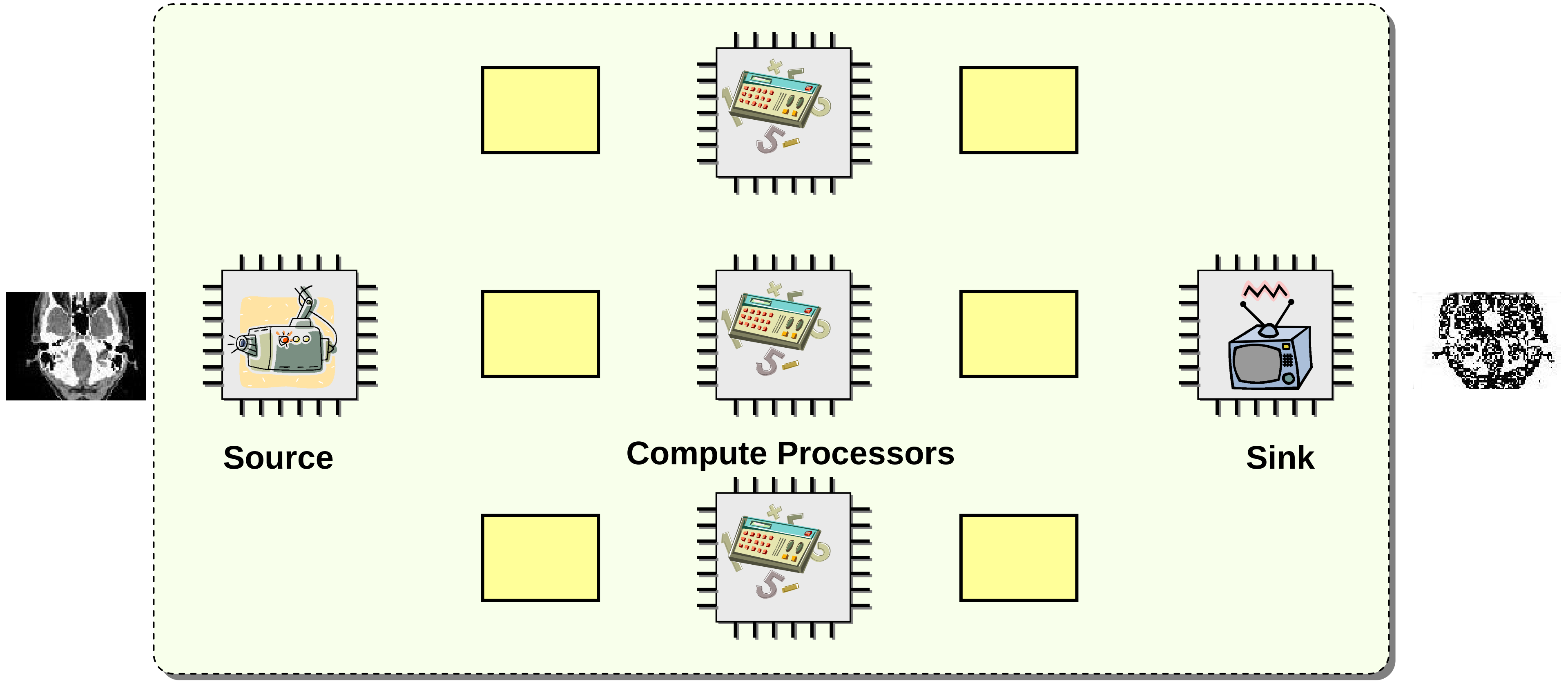


Multi-processor ARM (MPARM) Framework



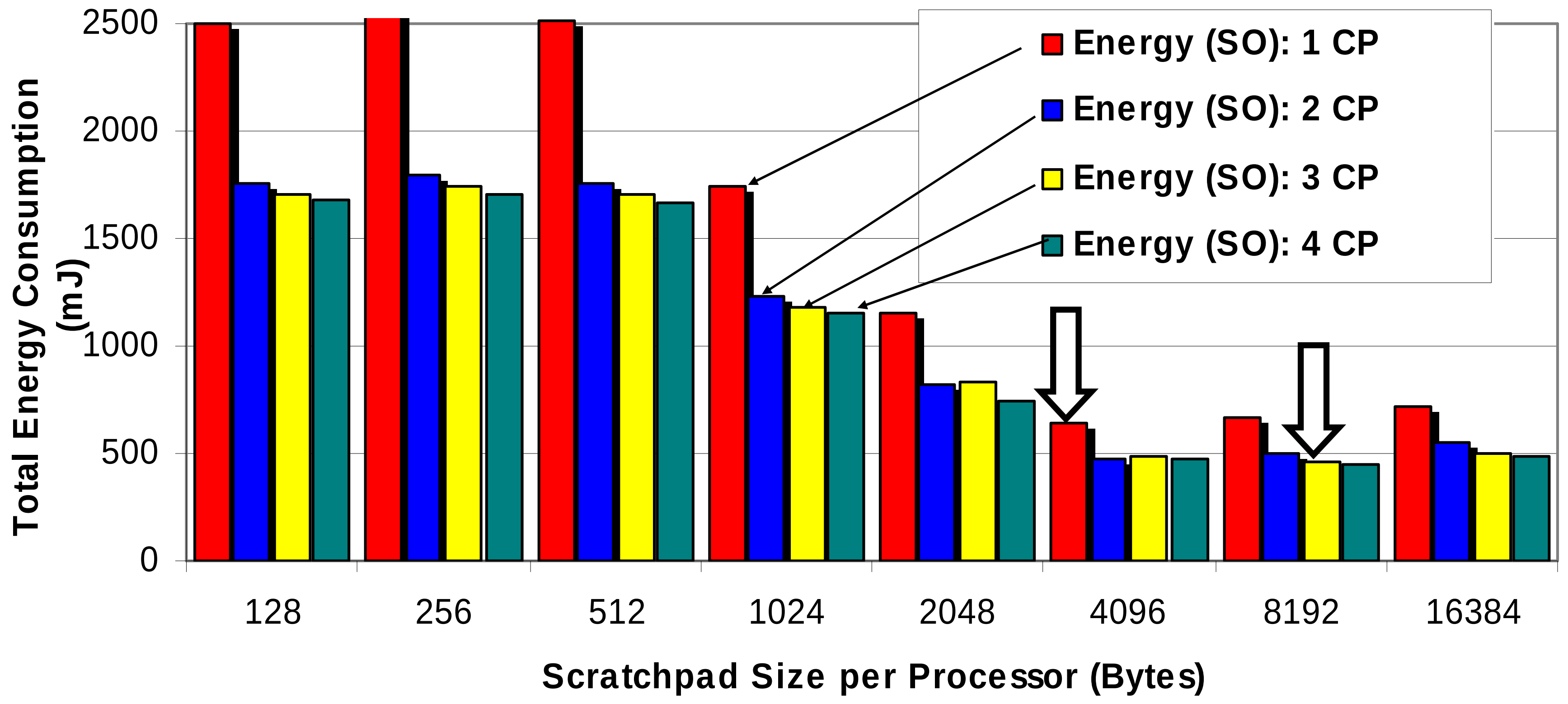
- Homogenous SMP ~ CELL processor
- Processing Unit : ARM7T processor
- Shared Coherent Main Memory
- Private Memory: Scratchpad Memory

Application Example:
Multi-Processor Edge Detection



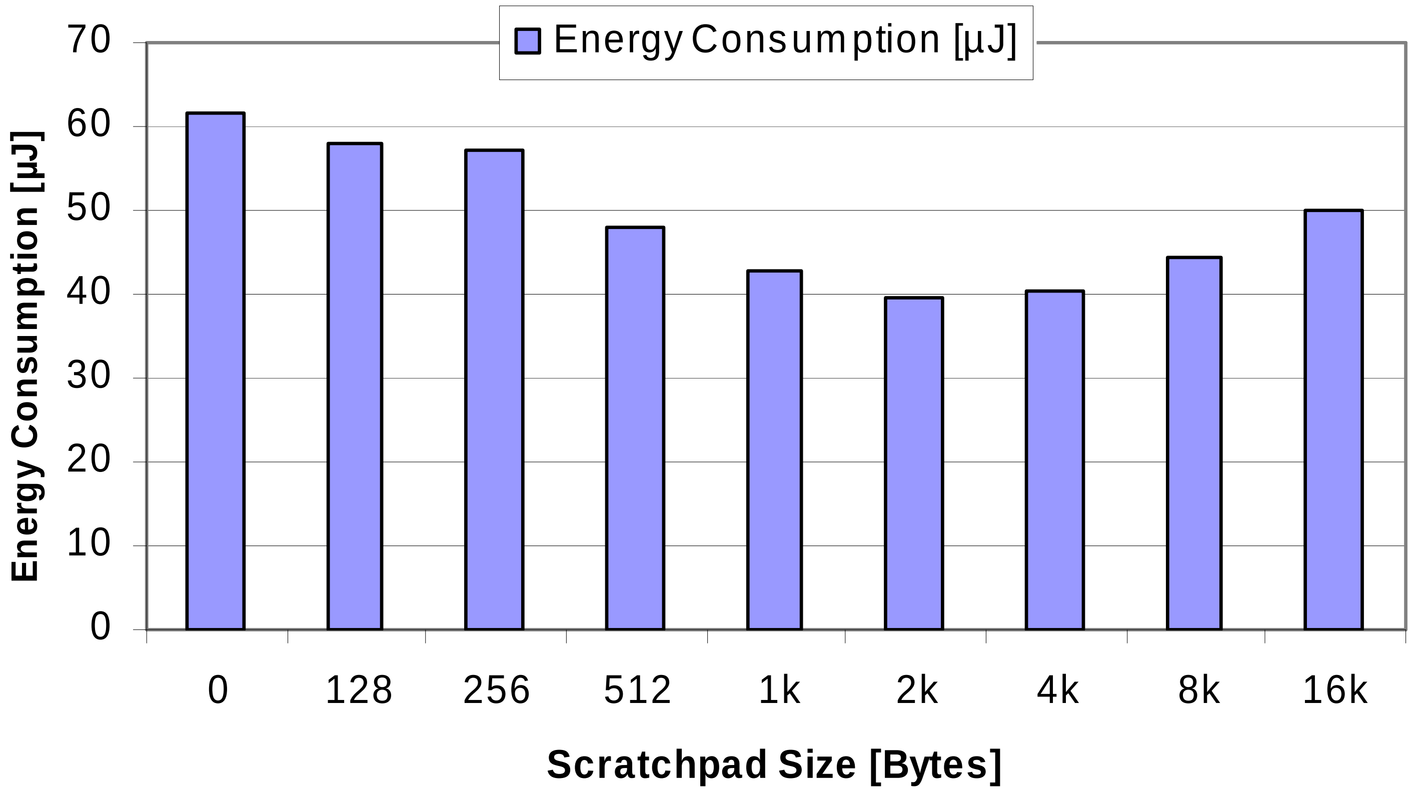
- Source, sink and *n* compute processors
- Each image is processed by an independent compute processor
    - Communication overhead is minimized.

Results:
Scratchpad Overlay for Edge Detection



- 2 CPs are better than 1 CP, then energy consumption stabilizes
- Best scratchpad size: 4kB (1CP& 2CP) 8kB (3CP & 4CP)

Results
DES-Encryption

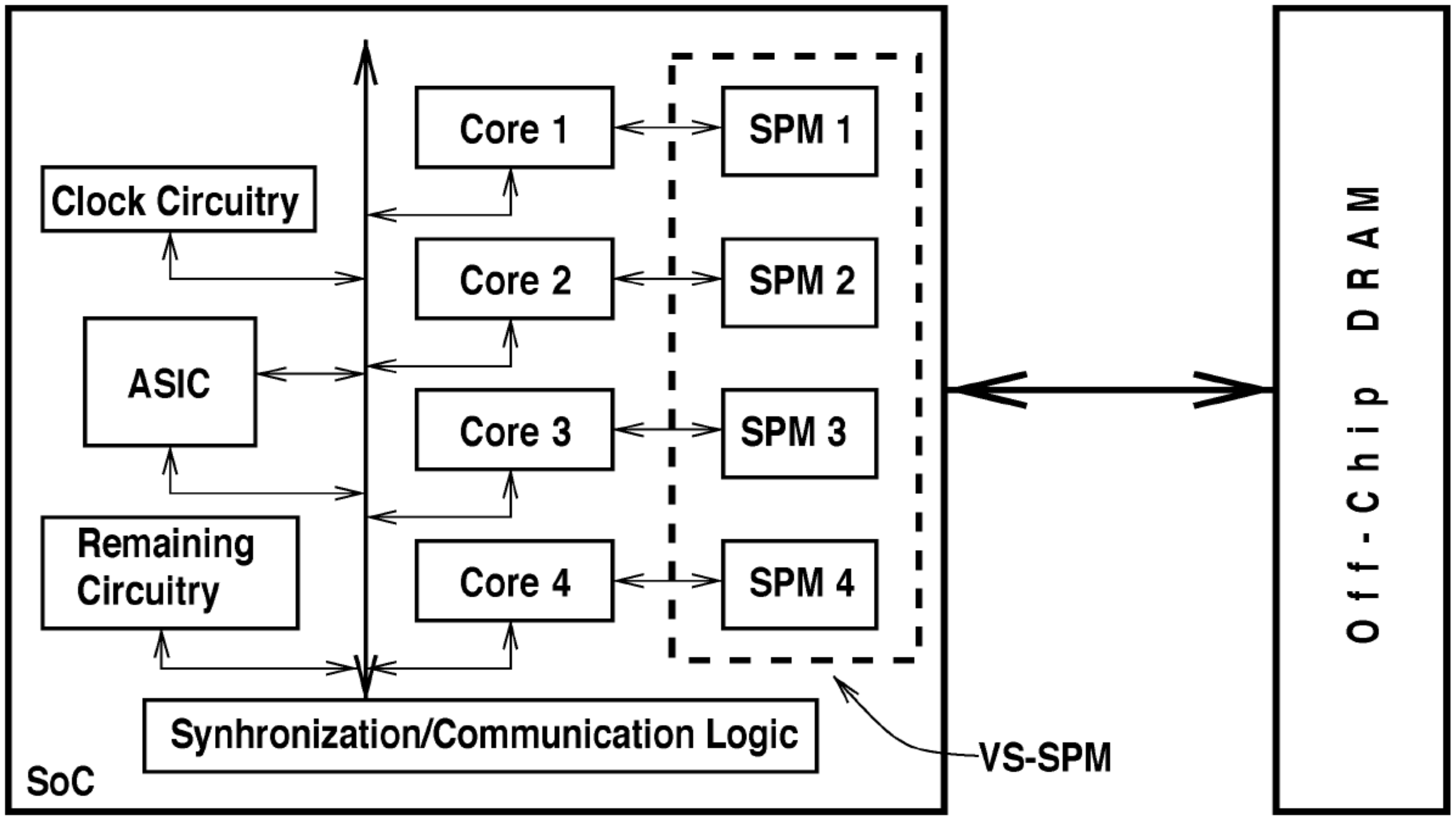


DES-Encryption: 4 processors: 2 Controllers+2 Compute Engines

Energy values from ST Microelectronics

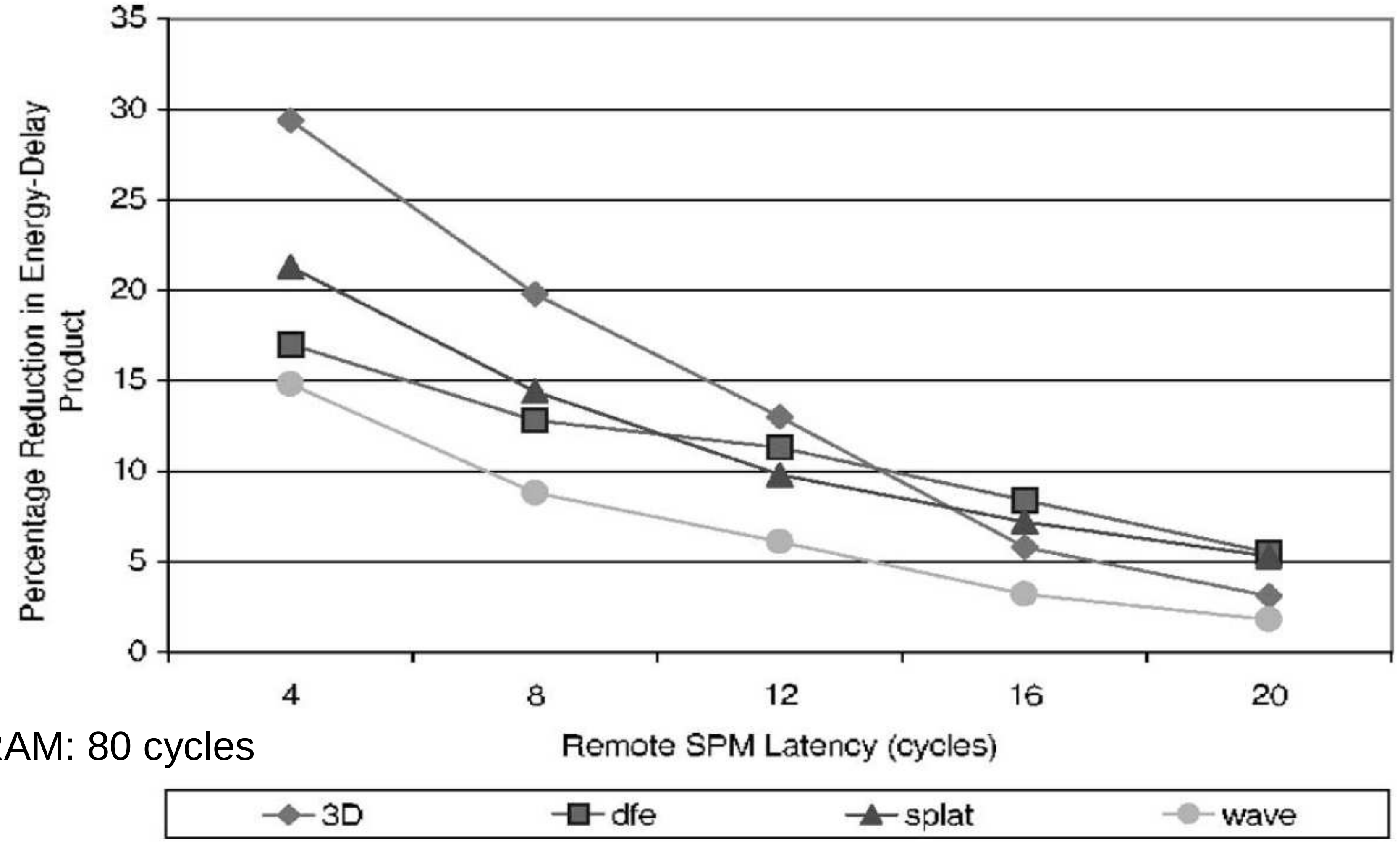
Result of ongoing cooperation between U. Bologna and U. Dortmund supported by ARTIST2 network of excellence.

MPSoC with shared SPMs



[M. Kandemir, I. Kadayif, A. Choudhary, J. Ramanujam, I. Kolcu: Compiler-Directed Scratch Pad Memory Optimization for Embedded Multiprocessors, *IEEE Trans. on VLSI*, Vol. 12, 2004, pp. 281-286]

Energy benefits despite large latencies for remote SPMs



DRAM: 80 cycles

Extensions

- Using DRAM
- Applications to Flash memory (copy code or execute in place): according to own experiments: very much parameter dependent
- Trying to imitate advantages of SPM with caches: partitioned caches, etc.

$\left.\begin{array}{l}\text{Using DRAM}\\ \text{Applications to Flash memory}\end{array}\right\}$ PhD thesis of Lars Wehmeyer

Improving predictability for caches

- Loop caches
- Mapping code to less used part(s) of the index space
- Cache locking/freezing
- Changing the memory allocation for code or data
- Mapping pieces of software to specific ways
  Methods:
  - Generating appropriate way in software
  - Allocation of certain parts of the address space to a specific way
  - Including way-identifiers in virtual to real-address translation

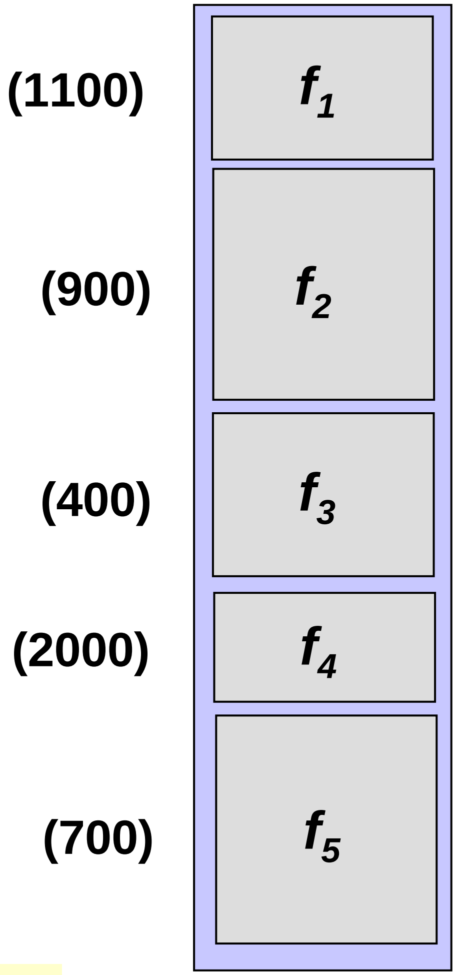
☞ “Caches behave almost like a scratch pad”

Code Layout Transformations (1)

Execution counts based approach:

- Sort the functions according to execution counts
  $f_4 > f_1 > f_2 > f_5 > f_3$

- Place functions in decreasing order of execution counts



[S. McFarling: Program optimization for instruction caches, 3*rd International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS),* 1989]

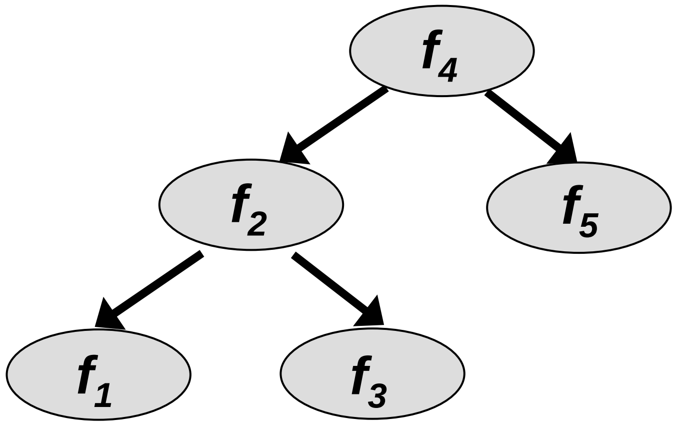
Code Layout Transformations (2)

Execution counts based approach:

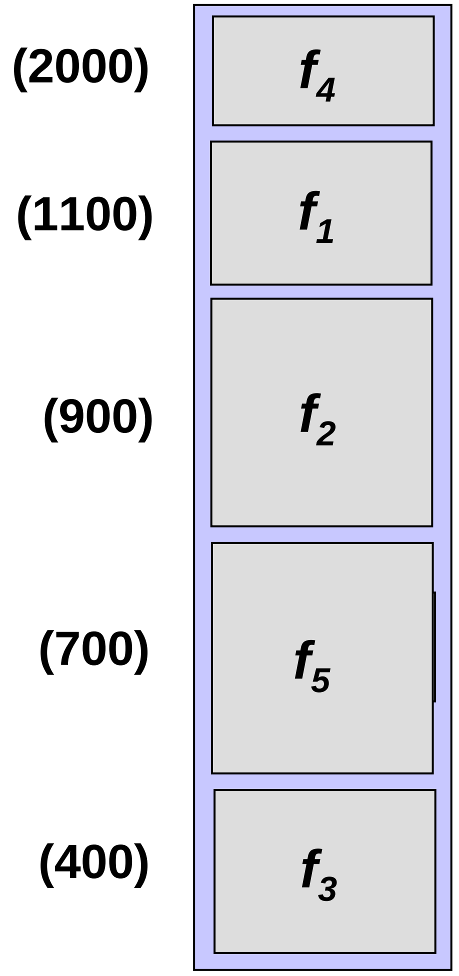
- Sort the functions according to execution counts
  $f_4 > f_1 > f_2 > f_5 > f_3$
- Place functions in decreasing order of execution counts

Transformation increases spatial locality.

Does not take in account calling order



$$\begin{tikzcd} & f_4 \arrow[dl] \arrow[dr] & \\ f_2 \arrow[d] \arrow[dr] & & f_5 \\ f_1 & f_3 & \end{tikzcd}$$



Code Layout Transformations (3)

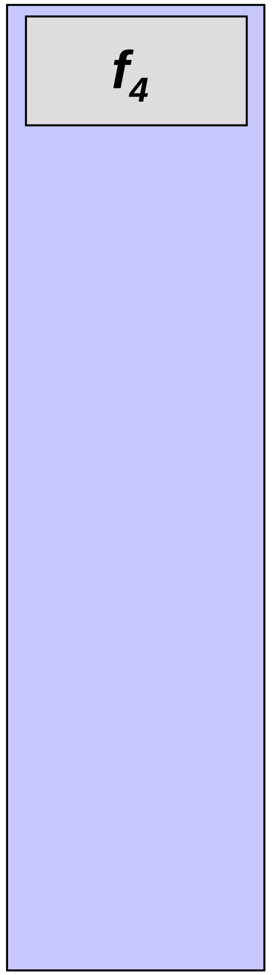
Call-Graph Based Algorithm:

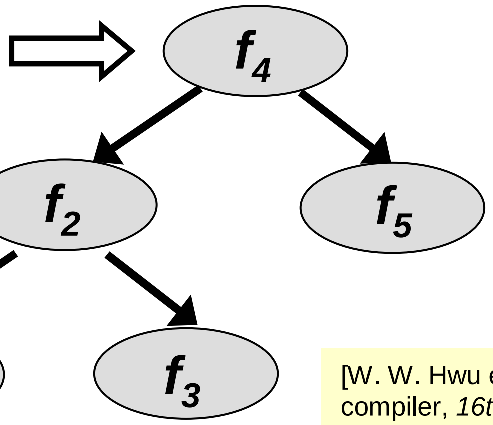
- Create weighted call-graph.
- Place functions according to weighted depth-first traversal.

  $f_4 > f_2 > f_1 > f_3 > f_5$

Increases spatial locality.

(2000)





[W. W. Hwu et al.: Achieving high instruction cache performance with an optimizing compiler, *16th Annual International Symposium on Computer Architecture,* 1989]

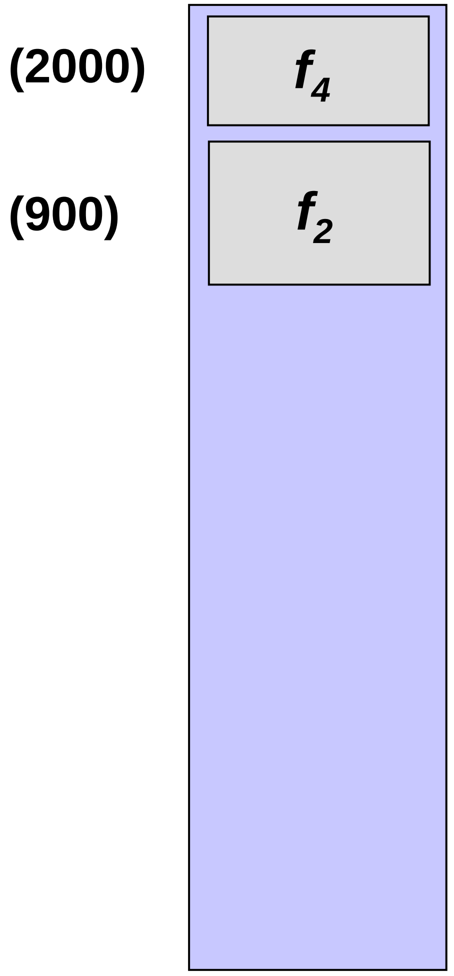
Code Layout Transformations (3)

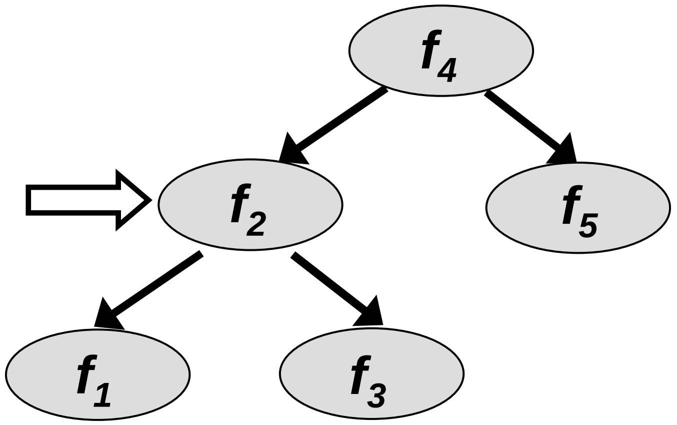
Call-Graph Based Algorithm:

- Create weighted call-graph.
- Place functions according to weighted depth-first traversal.

  $f_4 > f_2 > f_1 > f_3 > f_5$

Increases spatial locality.





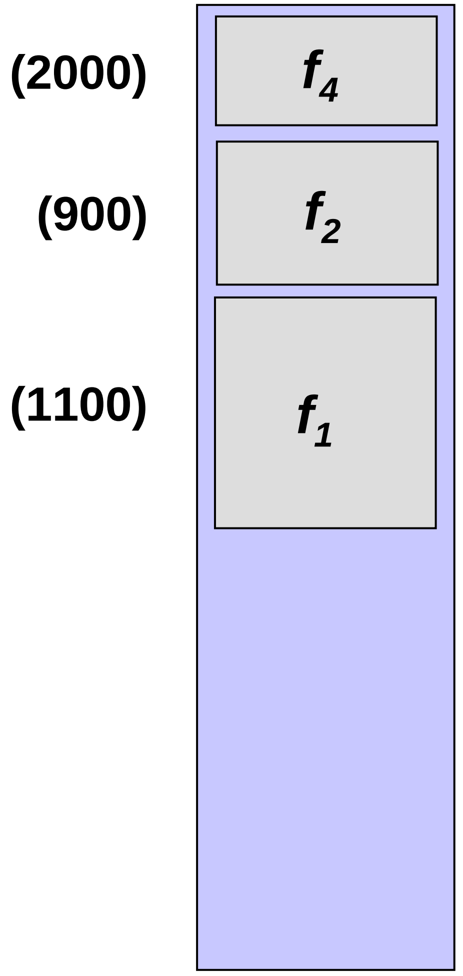
Code Layout Transformations (4)

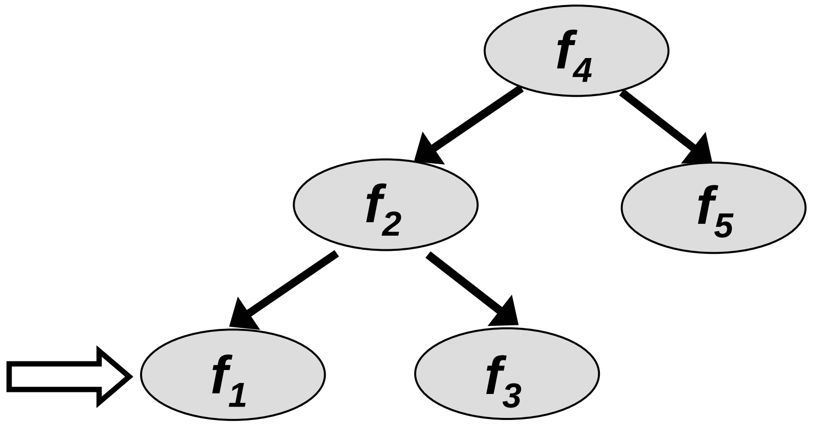
Call-Graph Based Algorithm:

- Create weighted call-graph.
- Place functions according to weighted depth-first traversal.

  $f_4 > f_2 > f_1 > f_3 > f_5$

Increases spatial locality.





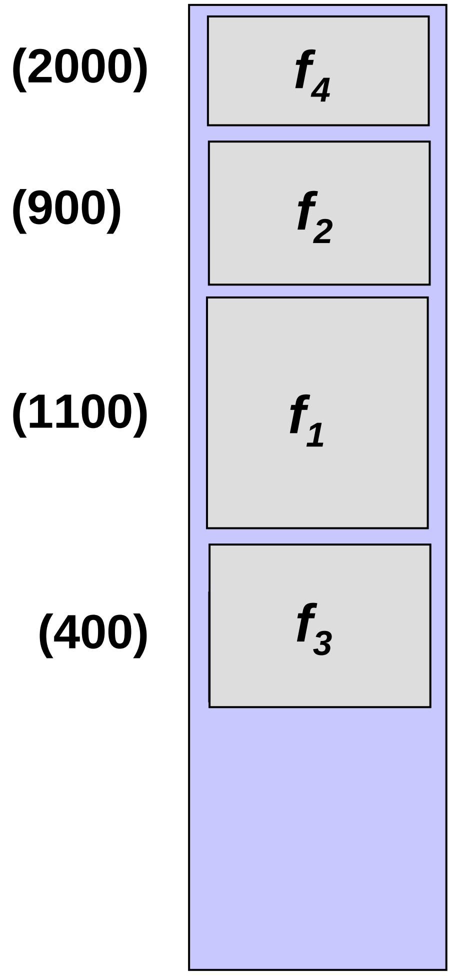
Code Layout Transformations (5)

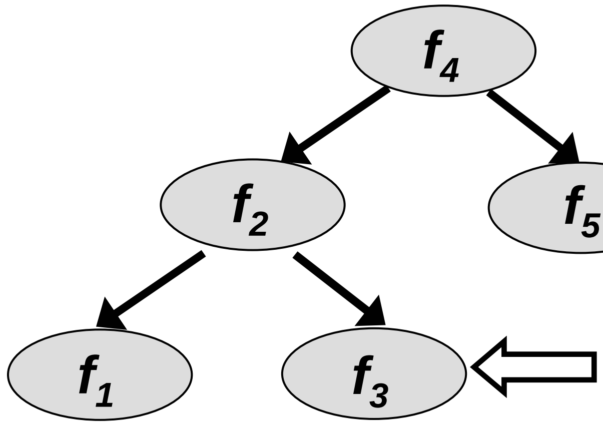
Call-Graph Based Algorithm:

- Create weighted call-graph.
- Place functions according to weighted depth-first traversal.

  $f_4 > f_2 > f_1 > f_3 > f_5$

Increases spatial locality.





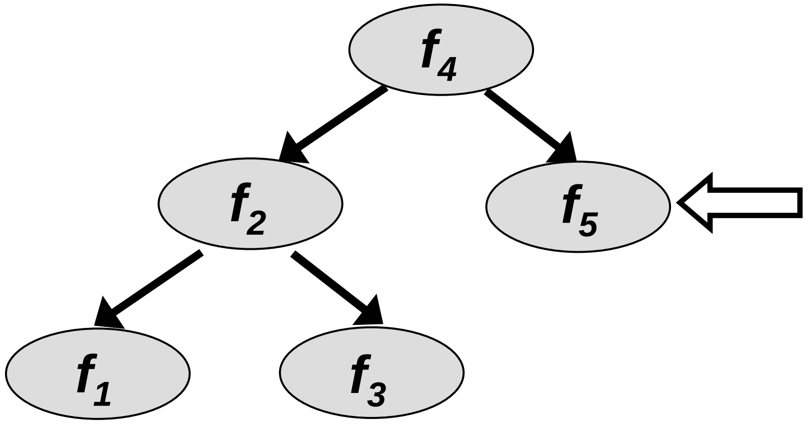
Code Layout Transformations (6)

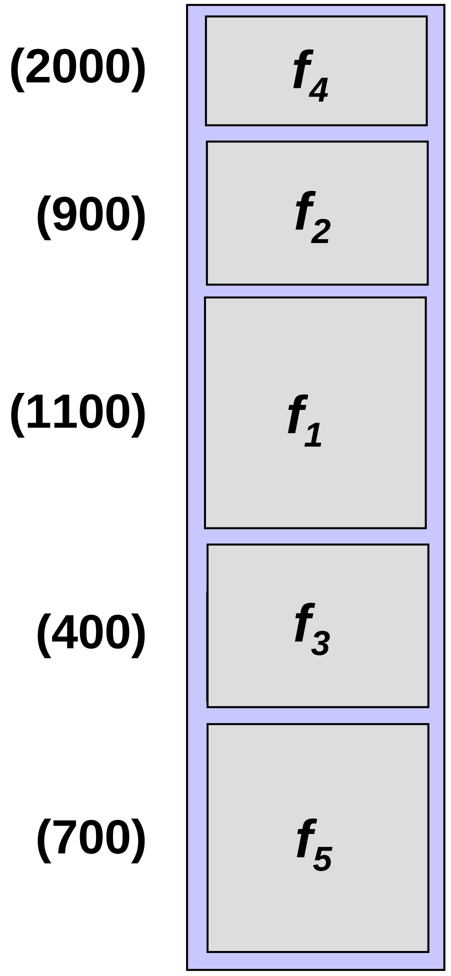
Call-Graph Based Algorithm:

- Create weighted call-graph.
- Place functions according to weighted depth-first traversal.

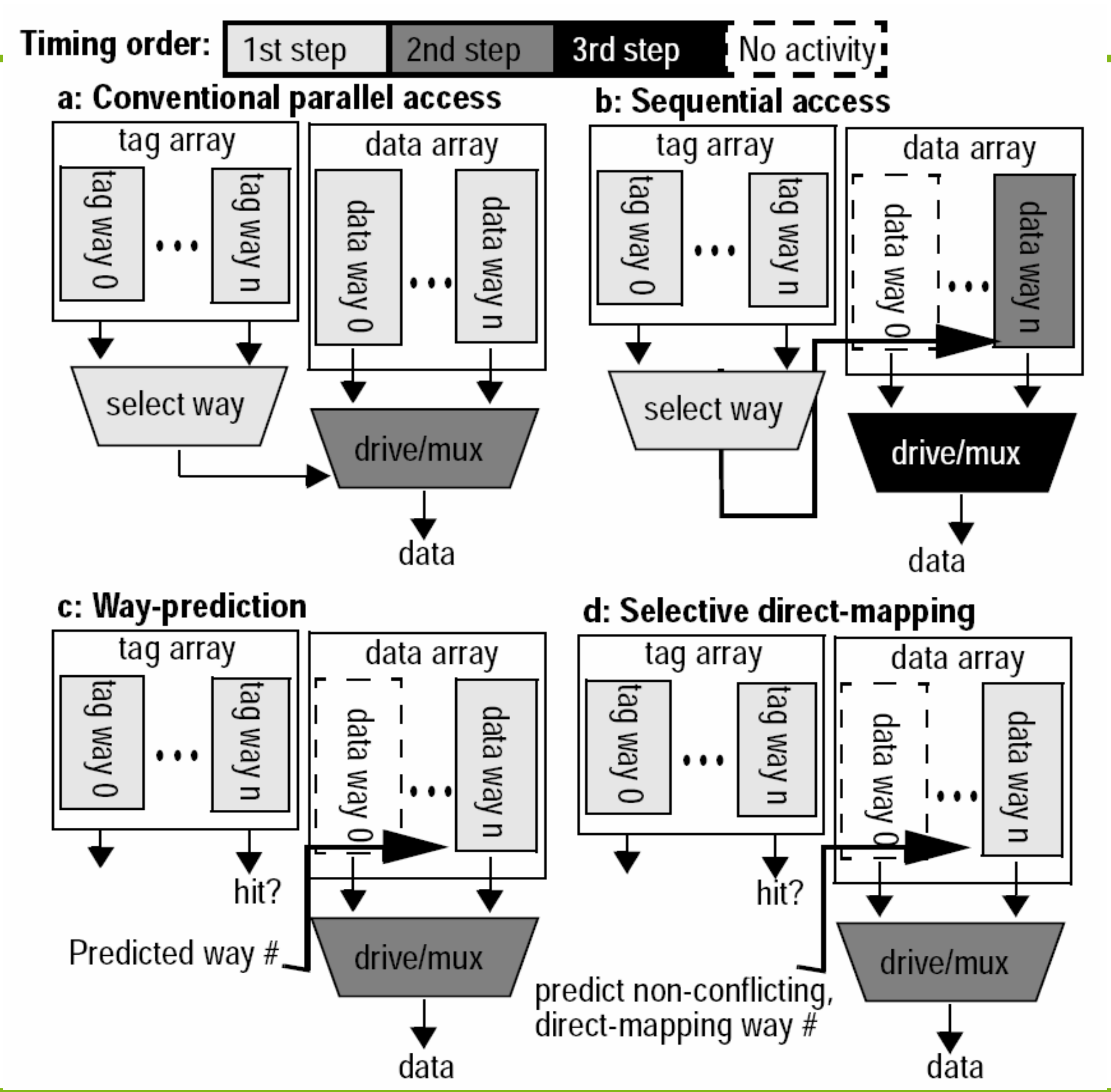
  $f_4 > f_2 > f_1 > f_3 > f_5$
- Combined with placing frequently executed traces at the top of the code space for functions.

Increases spatial locality.



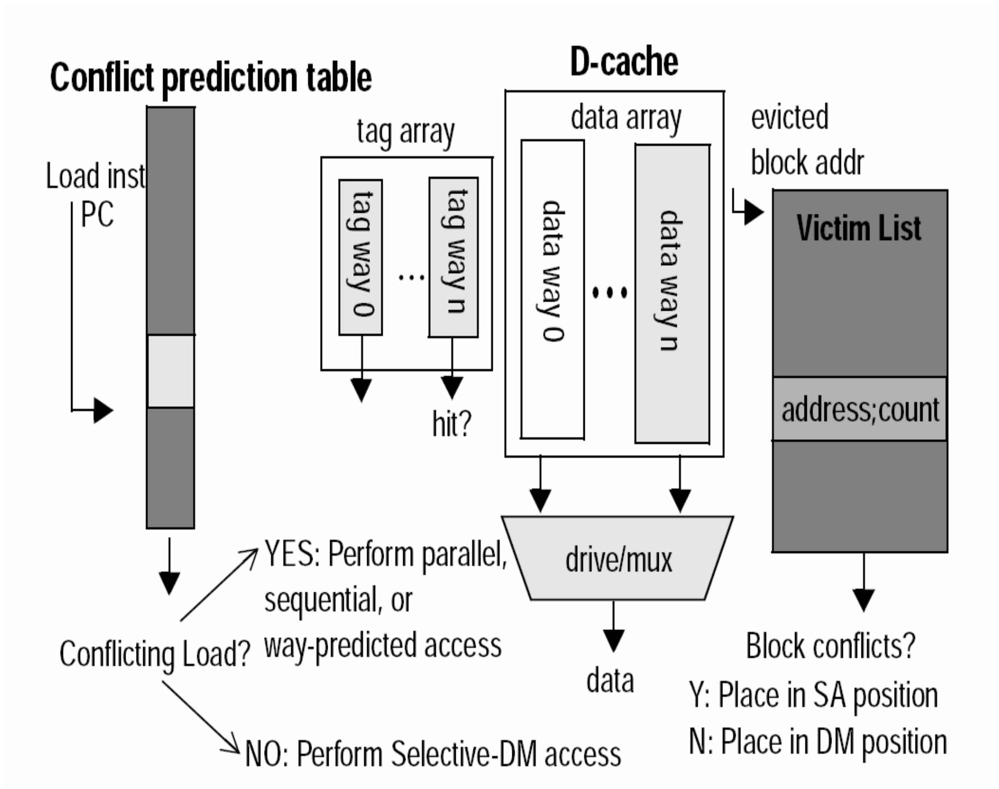


Way prediction/selective direct mapping



[M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, K. Roy: Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping, *MICRO-34*, 2001]

Hardware organization for way prediction



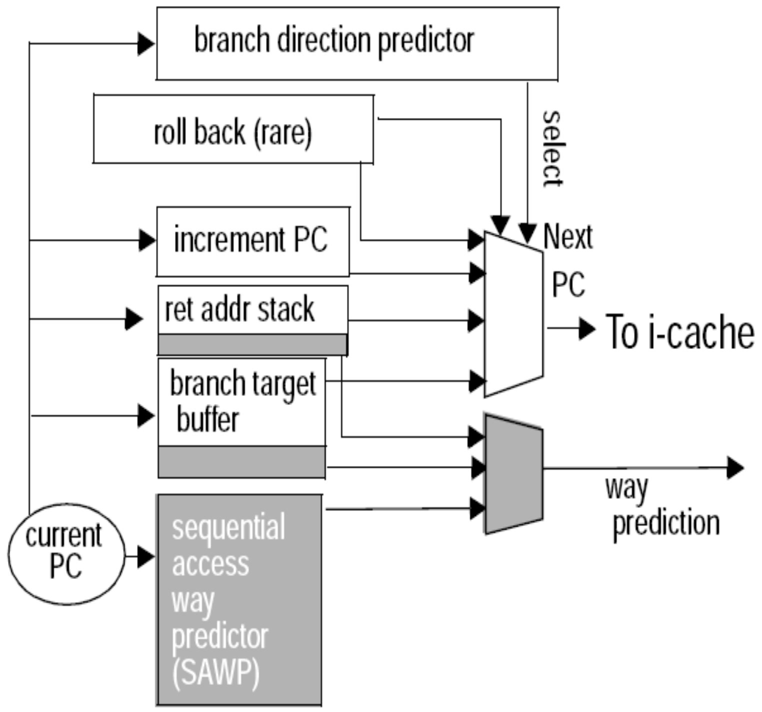


FIGURE 3: Fetch and i-cache access mechanism.

Results for the paper on way prediction (1)

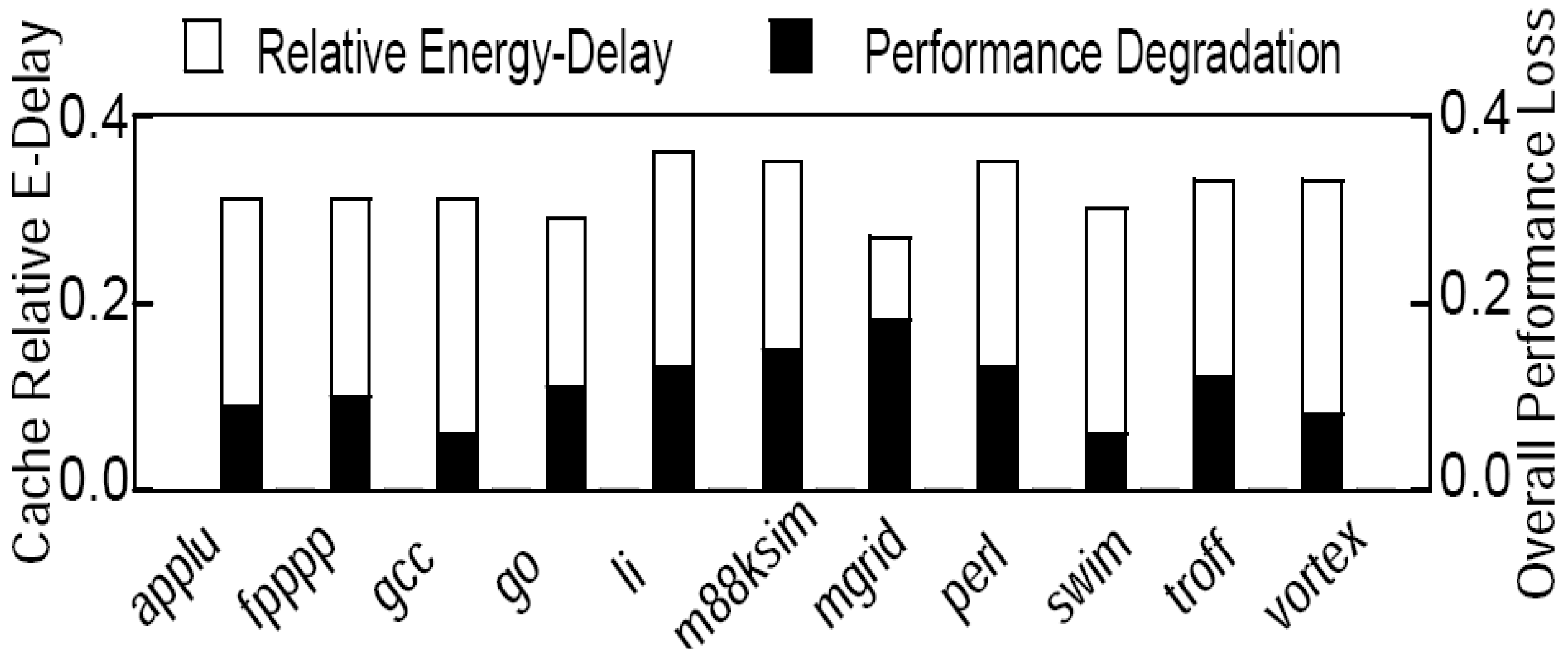
System configuration parameters

Instruction issue & decode bandwidth	8 issues per cycle
L1 I-Cache	16K, 4-way, 1 cycle
Base L1 D-Cache	16K, 4-way, 1 or 2 cycles, 2ports
L2 cache	1M, 8-way, 12 cycle latency
Memory access latency	80 cycles+4 cycles per 8 bytes
Reorder buffer size	64
LSQ size	32
Branch predictor	2-level hybrid

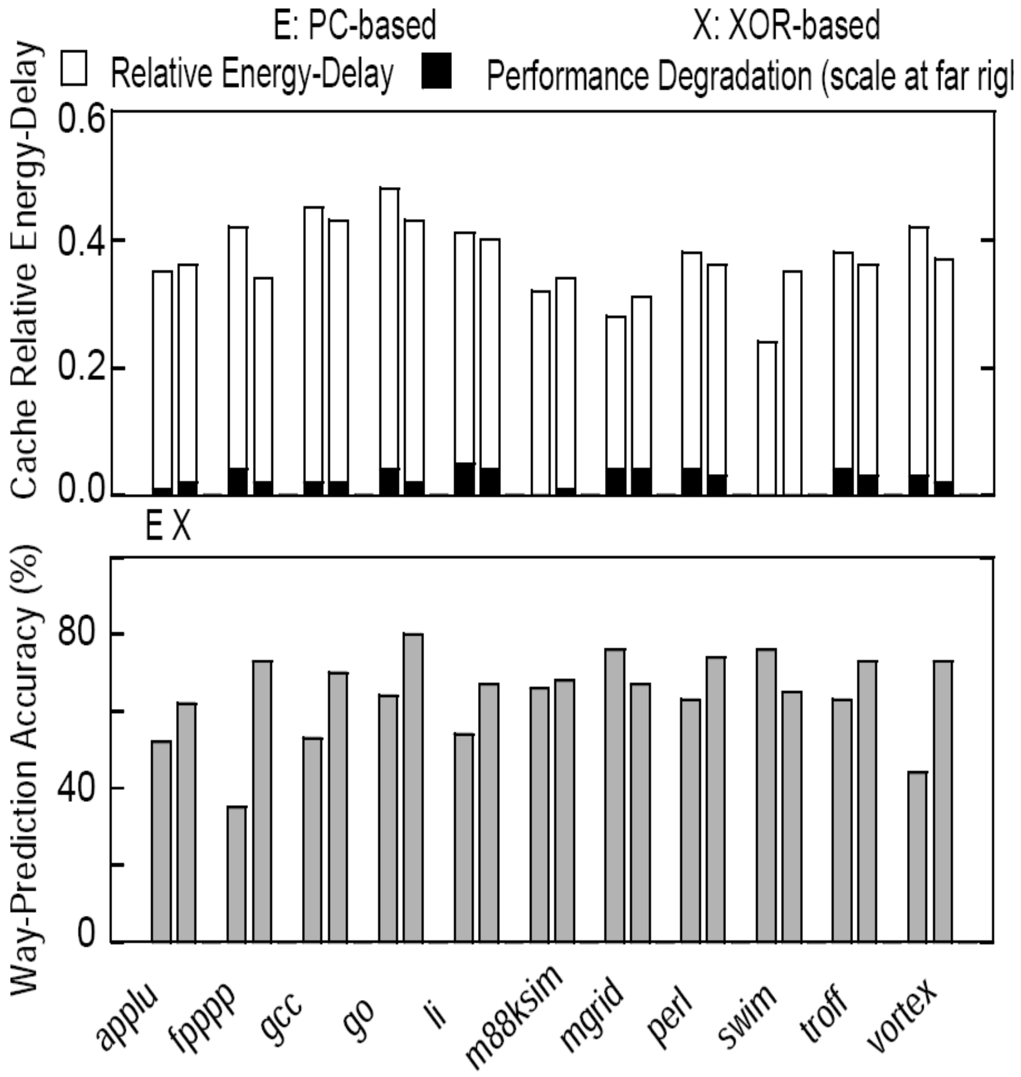
Cache energy and prediction overhead

Energy component	**Relative energy**
Parallel access cache read (4 ways read)	1.00
1 way read	**0.21**
Cache write	0.24
Tag array energy (incl. in the above numbers)	0.06
1024x4bit prediction table read/write	0.007

Results for the paper on way prediction (2)



Results for the paper on way prediction (2)

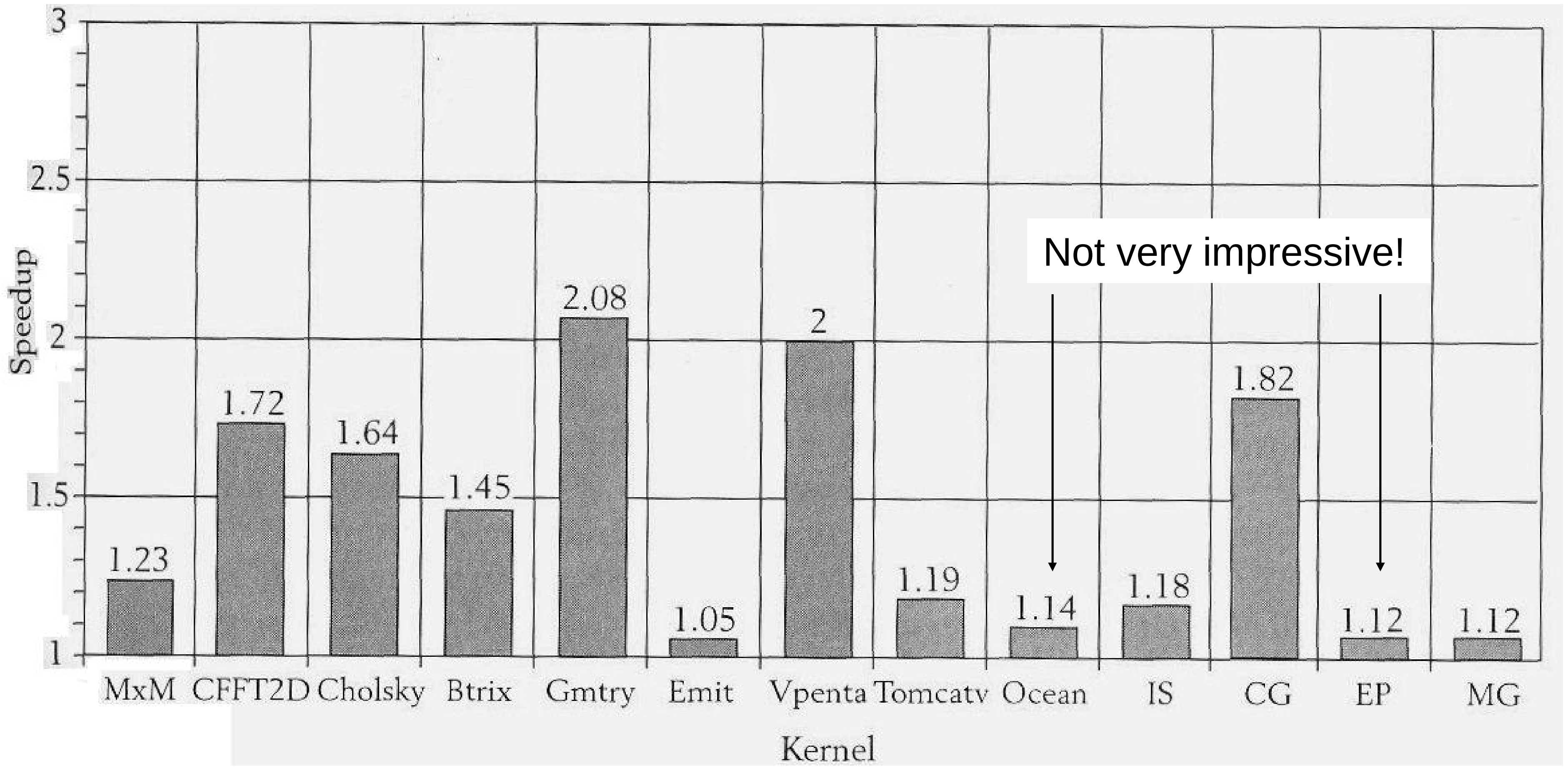


Prefetching

- Prefetch instructions load values into the cache
  Pipeline not stalled for prefetching
- Prefetching instructions introduced in ~1985-1995
- Potentially, all miss latencies can be avoided
- Disadvantages:
  - Increased # of instructions
  - Potential premature eviction of cache line
  - Potentially pre-loads lines that are never used
- Steps
  - Determination of references requiring prefetches
  - Insertion of prefetches (early enough!)

[R. Allen, K. Kennedy: Optimizing Compilers for Modern Architectures, *Morgan-Kaufman*, 2002]

Results for prefetching



[Mowry, as cited by R. Allen & K. Kennedy]

Optimization for exploiting processor-memory interface: Problem Definition (1)

XScale is stalled for 30% of time, but each stall duration is small

- Average stall duration = 4 cycles
- Longest stall duration < 100 cycles

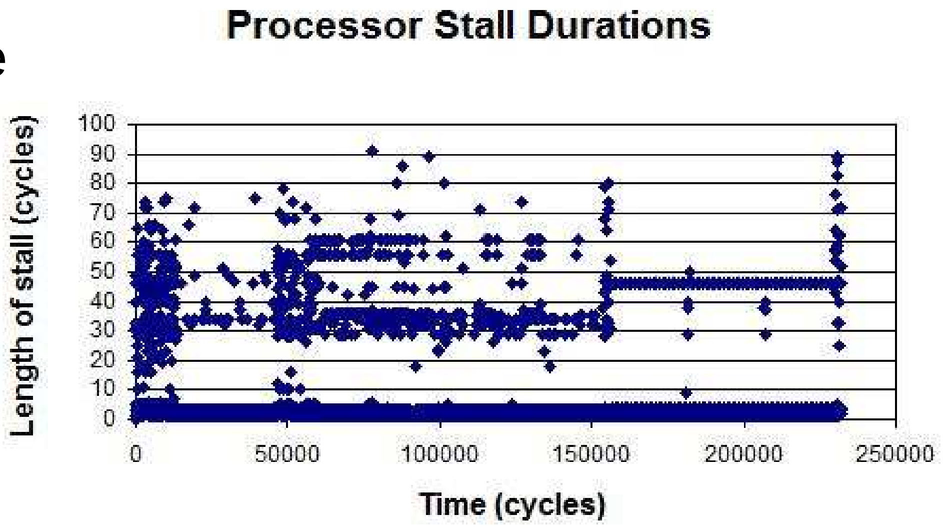
Break-even stall duration for profitable switching

- 360 cycles

Maximum processor stall

- < 100 cycles

NOT possible to switch the processor to IDLE mode



[A. Shrivastava, E. Earlie, N. Dutt, A. Nicolau: Aggregating processor free time for energy reduction, *Intern. Conf. on Hardware/Software Codesign and System Synthesis* (CODES/ISSS), 2005, pp. 154-159]

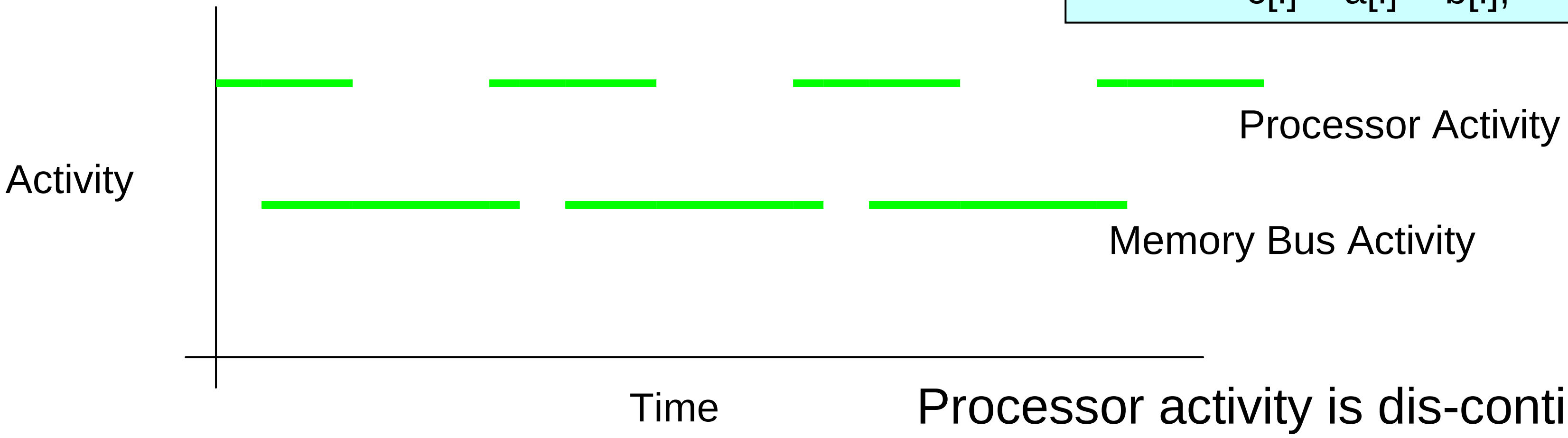
Optimization for exploiting processor-memory interface: Problem Definition (2)

- CT (Computation Time): Time to execute an iteration of the loop, assuming all data is present in the cache
- DT (Data Transfer Time): Time to transfer data required by an iteration of a loop between cache and memory

Consider the execution of a memory-bound loop (DT > CT)

- Processor has to stall

```
for (int i=0; i<1000; i++)
        c[i] = a[i] + b[i];
```

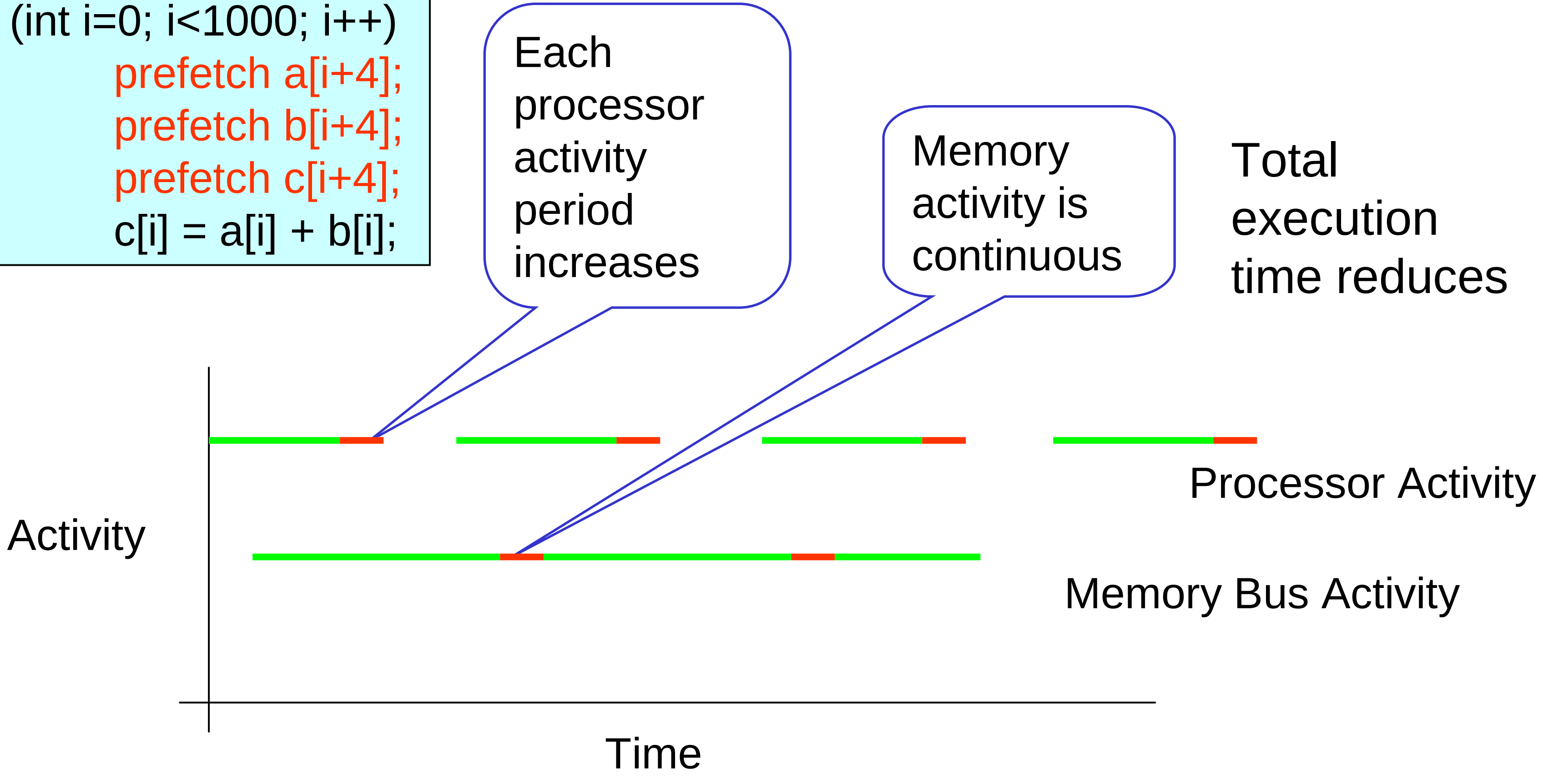


Processor activity is dis-continuous

Memory activity is dis-continuous

Optimization for exploiting processor-memory interface: Prefetching Solution

```
for (int i=0; i<1000; i++)
        prefetch a[i+4];
        prefetch b[i+4];
        prefetch c[i+4];
        c[i] = a[i] + b[i];
```


Each processor activity period increases
Memory activity is continuous
Total execution time reduces
Processor Activity
Activity
Memory Bus Activity
Time

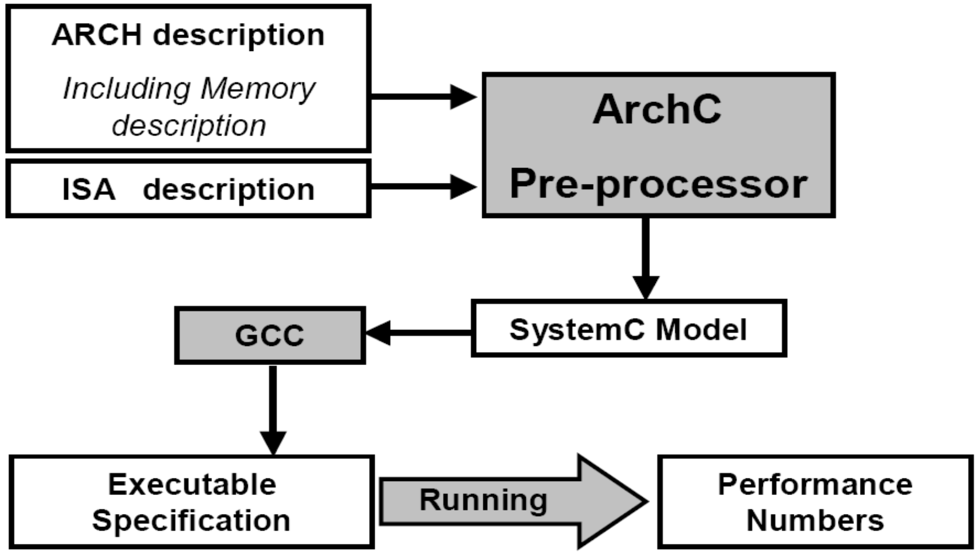
Processor activity is dis-continuous

Memory activity is continuous

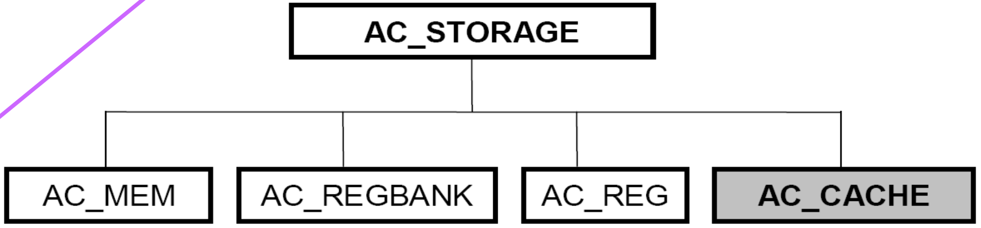
Memory hierarchy description languages: ArchC

Consists of description of ISA and HW architecture

Extension of SystemC (can be generated from ArchC):

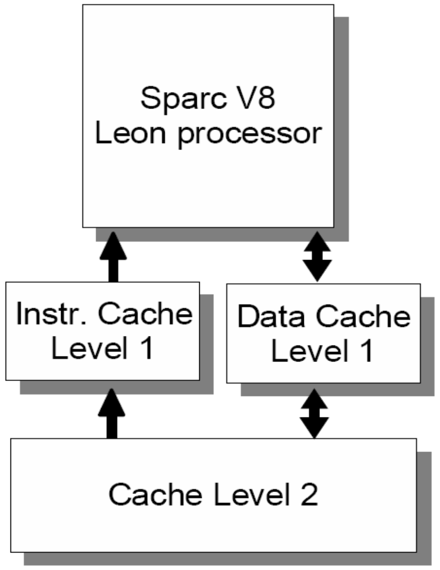


Storage class structure



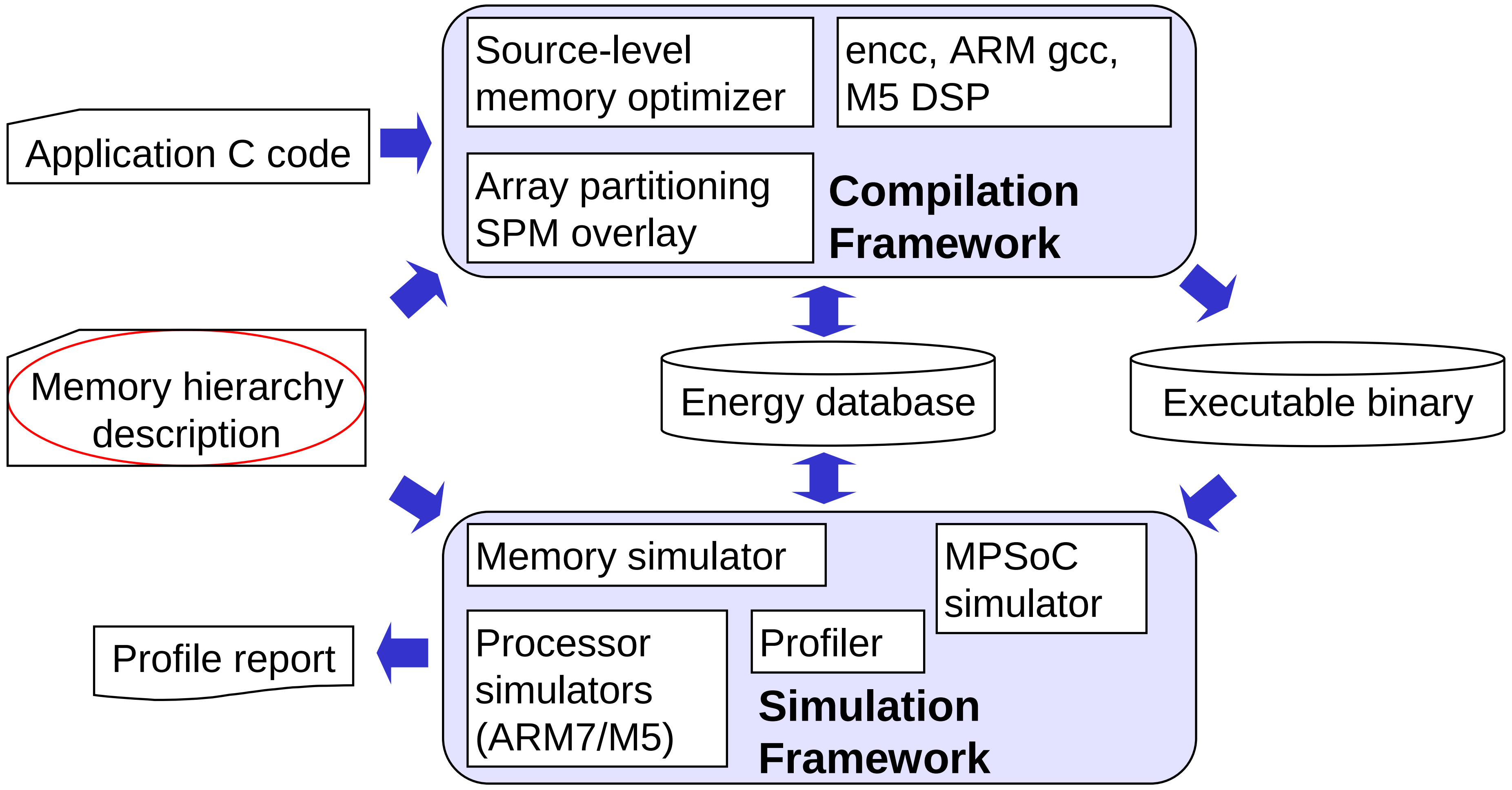
[P. Viana, E. Barros, S. Rigo, R. Azevedo, G. Araújo: Exploring Memory Hierarchy with ArchC, *15th Symposium on Computer Architecture and High Performance Computing*, 2003, pp. 2 – 9]

Example: Description of a simple cache-based architecture



```
AC_ARCH(leon){

  ac_cache   icache("dm", 128, "wt")
  ac_cache   dcache("2w", 64, 4, "wt", "lru")
  ac_cache   ul2cache("dm", 4k, "wt")

  ac_regbank RB:520;
  ac_reg PRS, Y, WIM;

  ac_pipe   pipe = {IF, ID, EX, MEM, WB};

  ARCH_CTOR(leon){
    ac_isa("leon_isa.ac");

    icache.bindTo( ul2cache );   //Memory hierarchy
    dcache.bindTo( ul2cache );   //construction
  };
};
```

Memory Aware Compilation and Simulation Framework (for C) MACC



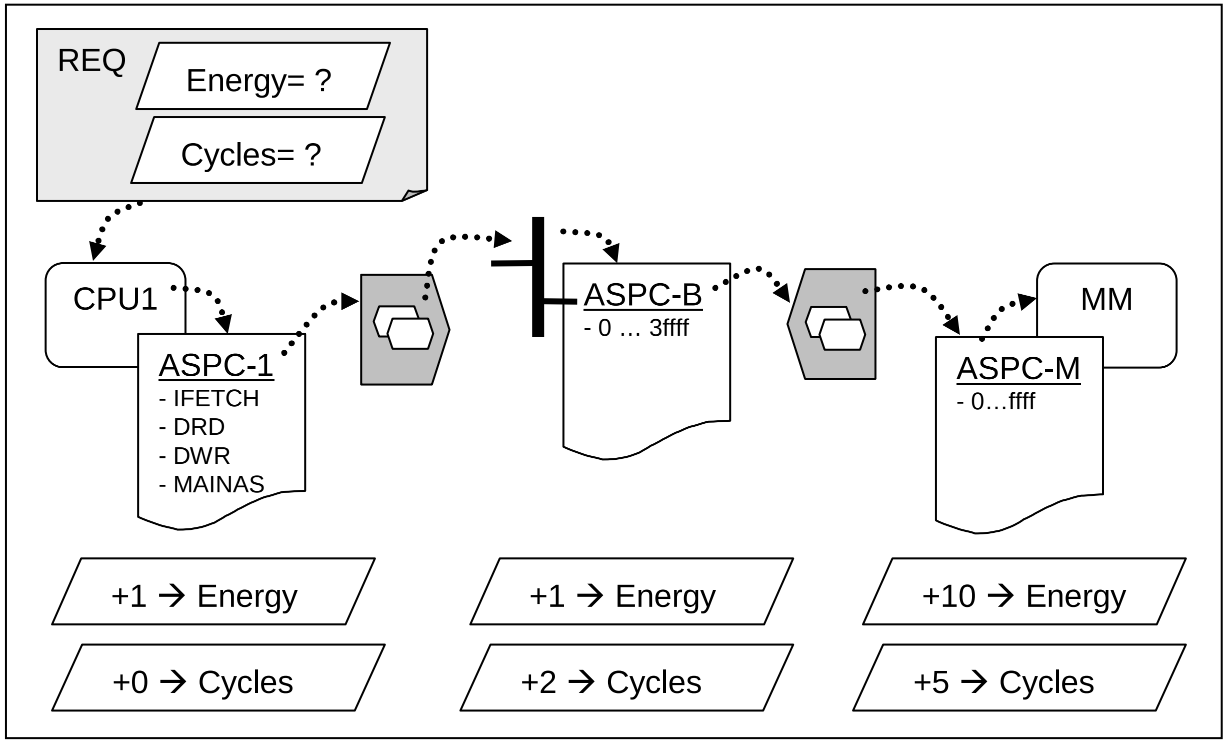
[M. Verma, L. Wehmeyer, R. Pyka, P. Marwedel, L. Benini: Compilation and Simulation Tool Chain for Memory Aware Energy Optimizations, *Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS VI)*, 2006].

Memory architecture description @ MACCv2

- Query can include address, time stamp, value, …
- Query can request energy, delay, stored values
- Query processed along a chain of HW components, incl. busses, ports, address translations etc., each adding delay & energy

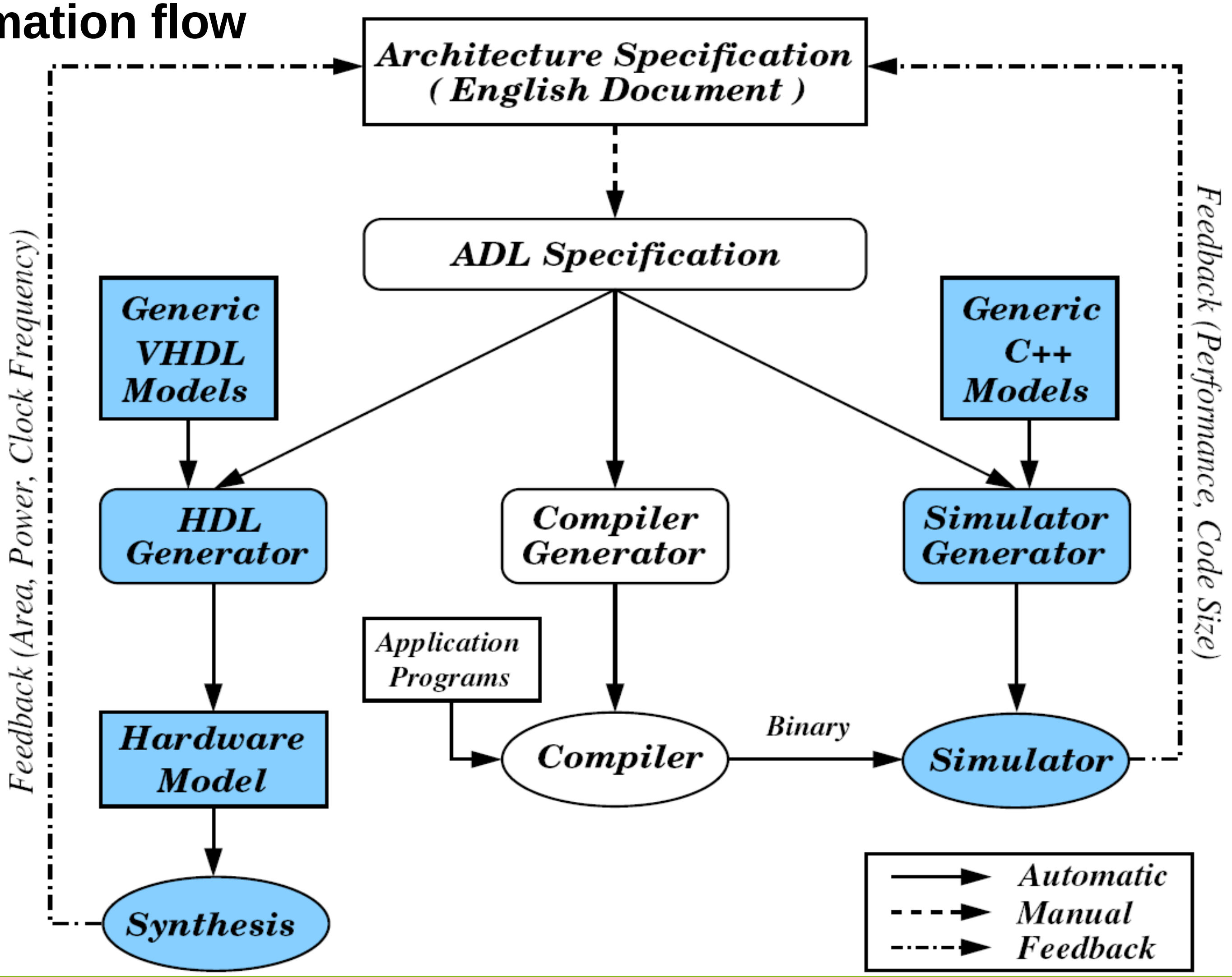
- API query to model simplifies integration into compiler
- External XML representation

[R. Pyka et al.: Versatile System level Memory Description Approach for embedded MPSoCs, *University of Dortmund, Informatik 12*, 2007]



Controlling tool chain generation through an architecture description language (ADL): EXPRESSION

Overall information flow



[P. Mishra, A. Shrivastava, N. Dutt: Architecture description language (ADL)-driven software toolkit generation for architectural exploration of programmable SOCs, *ACM Trans. Des. Autom. Electron. Syst. (TODAES)*, 2006, pp. 626-658]

Description of Memories in EXPRESSION

Generic approach, based on the analysis of a wide range of systems;

Used for verification.

```
(STORAGE_SECTION
  (DataL1
      (TYPE DCACHE) (WORDSIZE 64)
      (LINESIZE 8) (NUM_LINES 1024)
      (ASSOCIATIVITY 2) (READ_LATENCY 1)  ...
      (REPLACEMENT_POLICY LRU)
      (WRITE_POLICY WRITE_BACK)
  )
  (ScratchPad
      (TYPE SRAM) (ADDRESS_RANGE 0 4095)  ....
  )
  (SB
      (TYPE STREAM_BUFFER) .....
  (InstL1
      (TYPE ICACHE) .........
  )
  (L2
      (TYPE DCACHE) .......
  )
  (MainMemory
      (TYPE DRAM)
  )
  (Connect
      (TYPE CONNECTIVITY)
      (CONNECTIONS
          (InstL1, L2) (DataL1, SB) (SB, L2)
          (L2, MainMemory)
)))
```

EXPRESSION: results

0

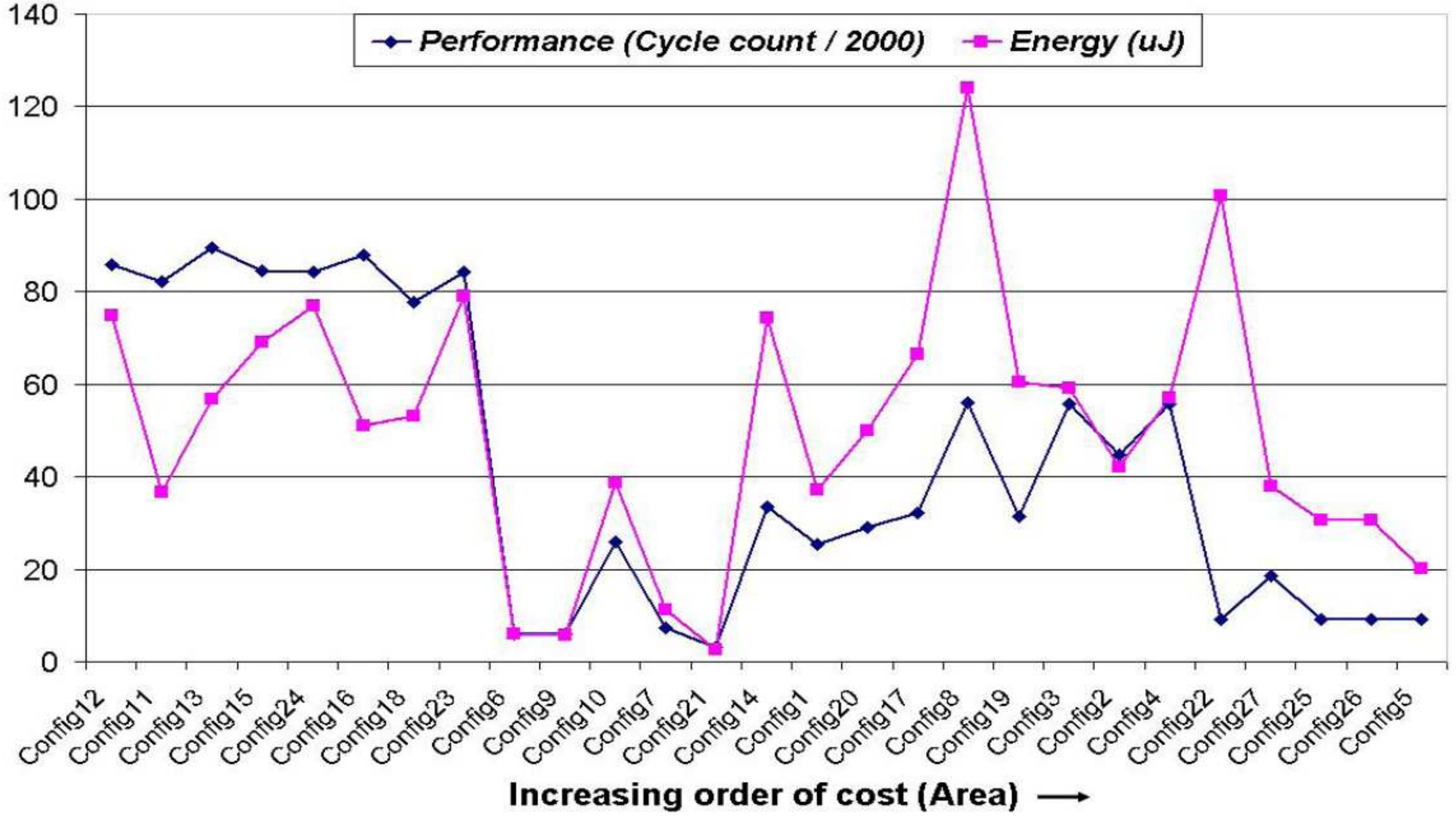
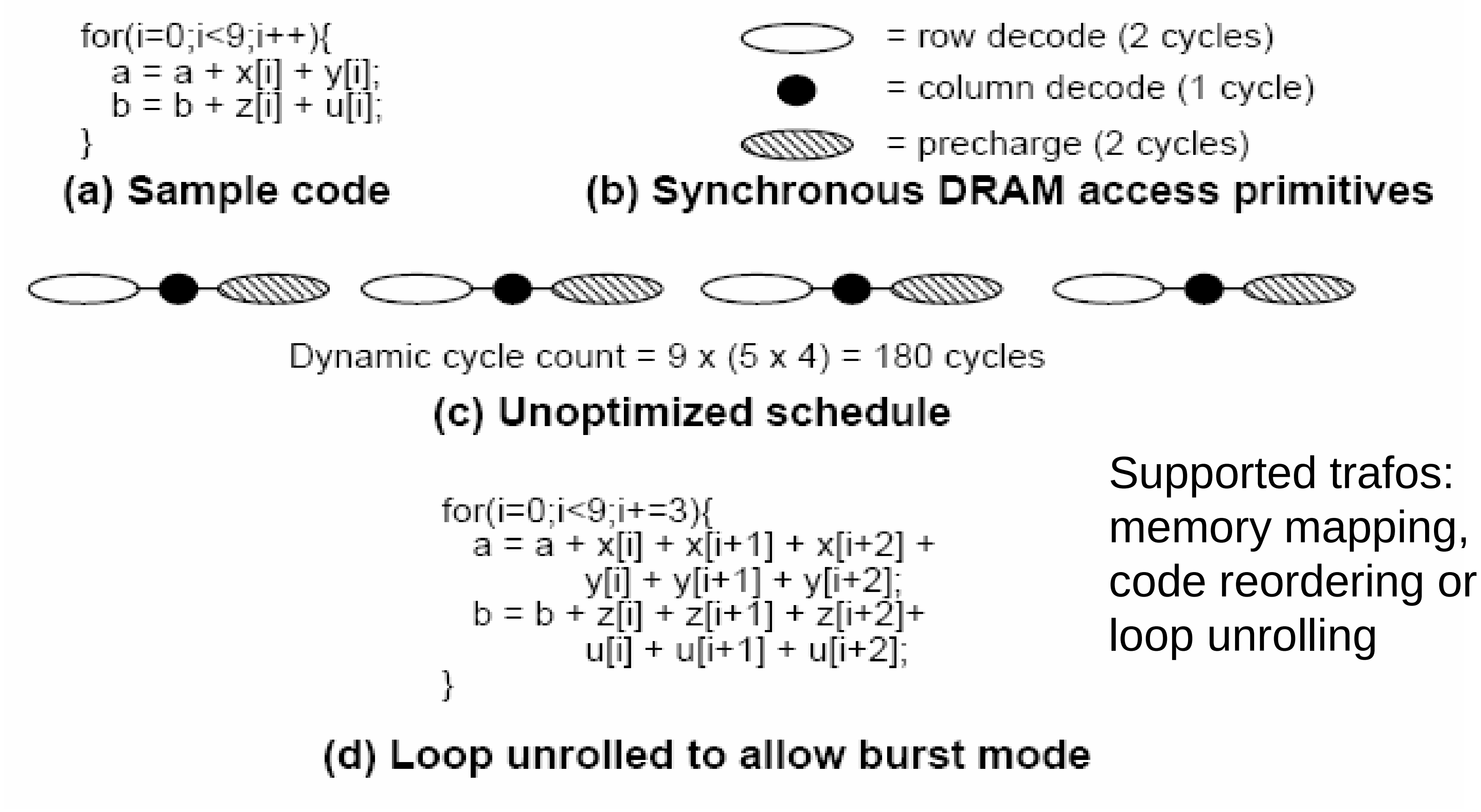


Figure 4.11: Memory exploration results for GSR

Optimization for main memory Exploiting burst mode of DRAM (1)



```
for(i=0;i<9;i++){
   a = a + x[i] + y[i];
   b = b + z[i] + u[i];
}
```

(a) Sample code

○ = row decode (2 cycles)

● = column decode (1 cycle)

▨ = precharge (2 cycles)

(b) Synchronous DRAM access primitives

Dynamic cycle count = 9 x (5 x 4) = 180 cycles

(c) Unoptimized schedule

```
for(i=0;i<9;i+=3){
   a = a + x[i] + x[i+1] + x[i+2] +
           y[i] + y[i+1] + y[i+2];
   b = b + z[i] + z[i+1] + z[i+2]+
           u[i] + u[i+1] + u[i+2];
}
```

(d) Loop unrolled to allow burst mode

Supported trafos: memory mapping, code reordering or loop unrolling

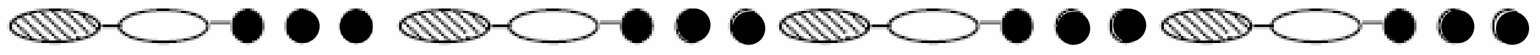
[P. Grun, N. Dutt, A. Nicolau: Memory aware compilation through accurate timing extraction, *DAC*, 2000, pp. 316 – 321]

Optimization for main memory Exploiting burst mode of DRAM (2)

Timing extracted from EXPRESSION model

```
for(i=0; i<9;i+=3){
a=a+x[i]+x[i+1]+x[i+2]+
      y[i]+y[i+1]+y[i+2];
b=b+z[i]+z[i+1]+z[i+2]+
      u[i]+u[i+1]+u[i+2];}
```

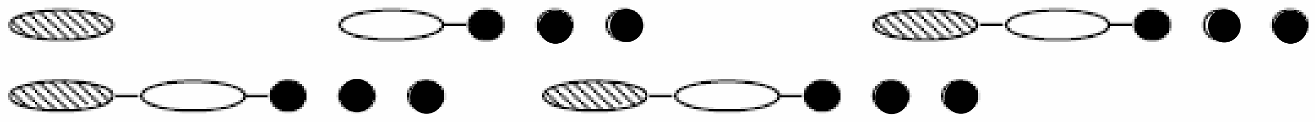
(d) Loop unrolled to allow burst mode



Dynamic behavior (dynamic cycle count = 3 x 28 = 84 cycles)

(e) Optimized code without accurate timing

2 banks



Dynamic cycle count = 3 x 20 = 60 cycles

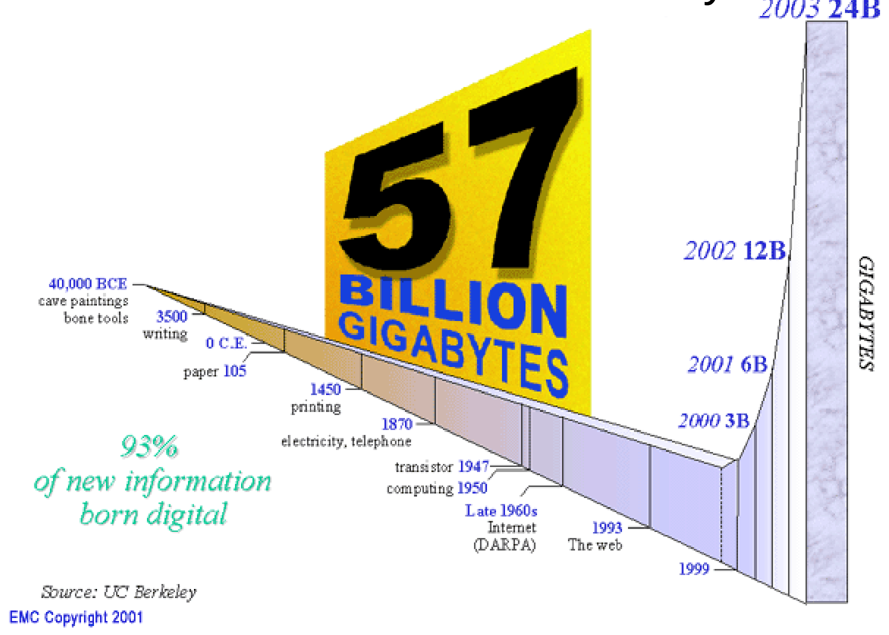
(f) Optimized code with accurate timing

Open circles of original paper changed into closed circles (column decodes).

Memory hierarchies beyond main memory

- Massive datasets are being collected everywhere
- Storage management software is billion-$ industry

More New Information Over Next 2 Years Than in All Previous History



Examples (2002):

Phone: AT&T 20TB phone call database, wireless tracking

Consumer: WalMart 70TB database, buying patterns

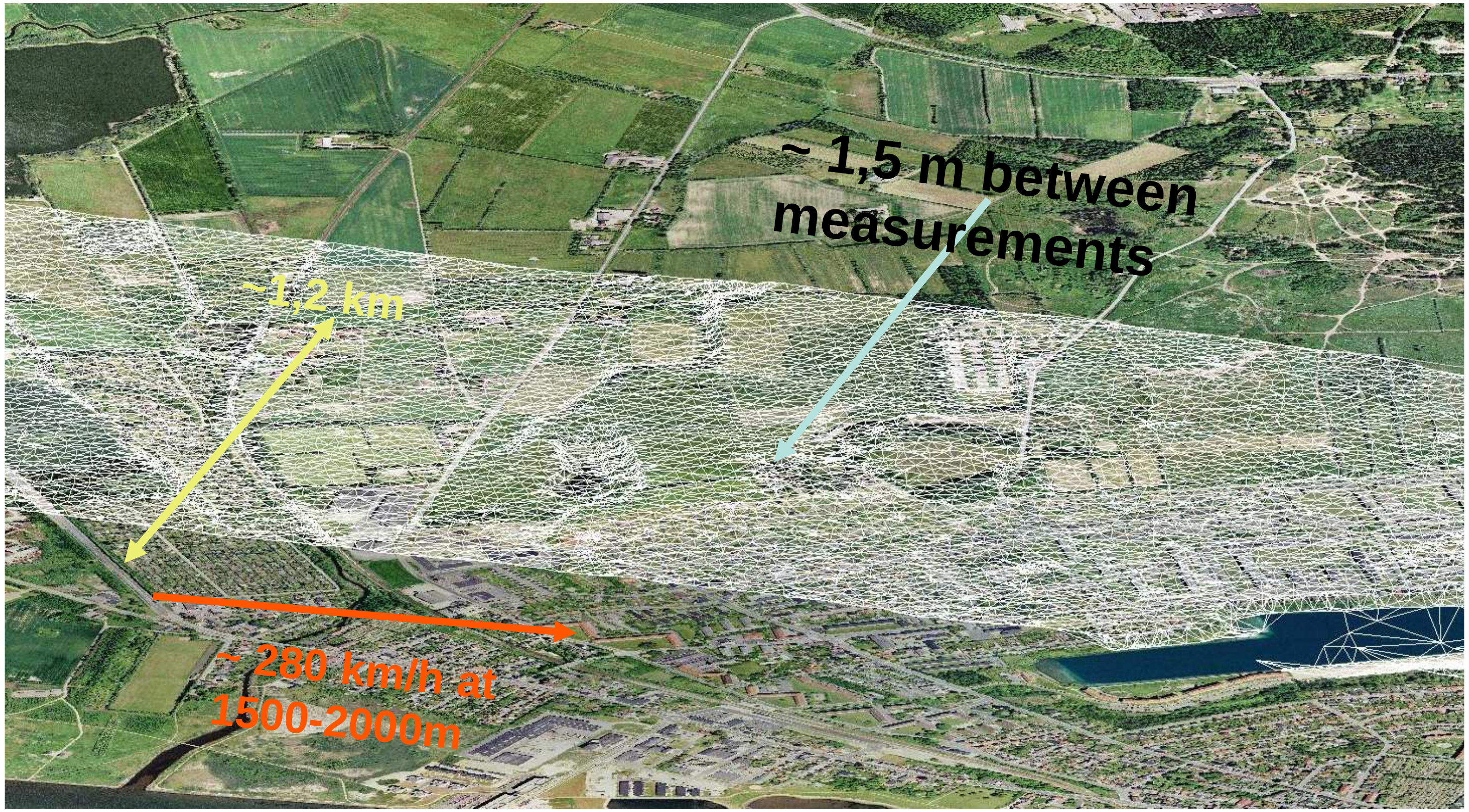
WEB: Web crawl of 200M pages and 2000M links, Akamai stores 7 billion clicks per day

Geography: NASA satellites generate 1.2TB per day

[© Larse Arge, I/O-Algorithms, http://www.daimi.au.dk/~large/ioS07/]

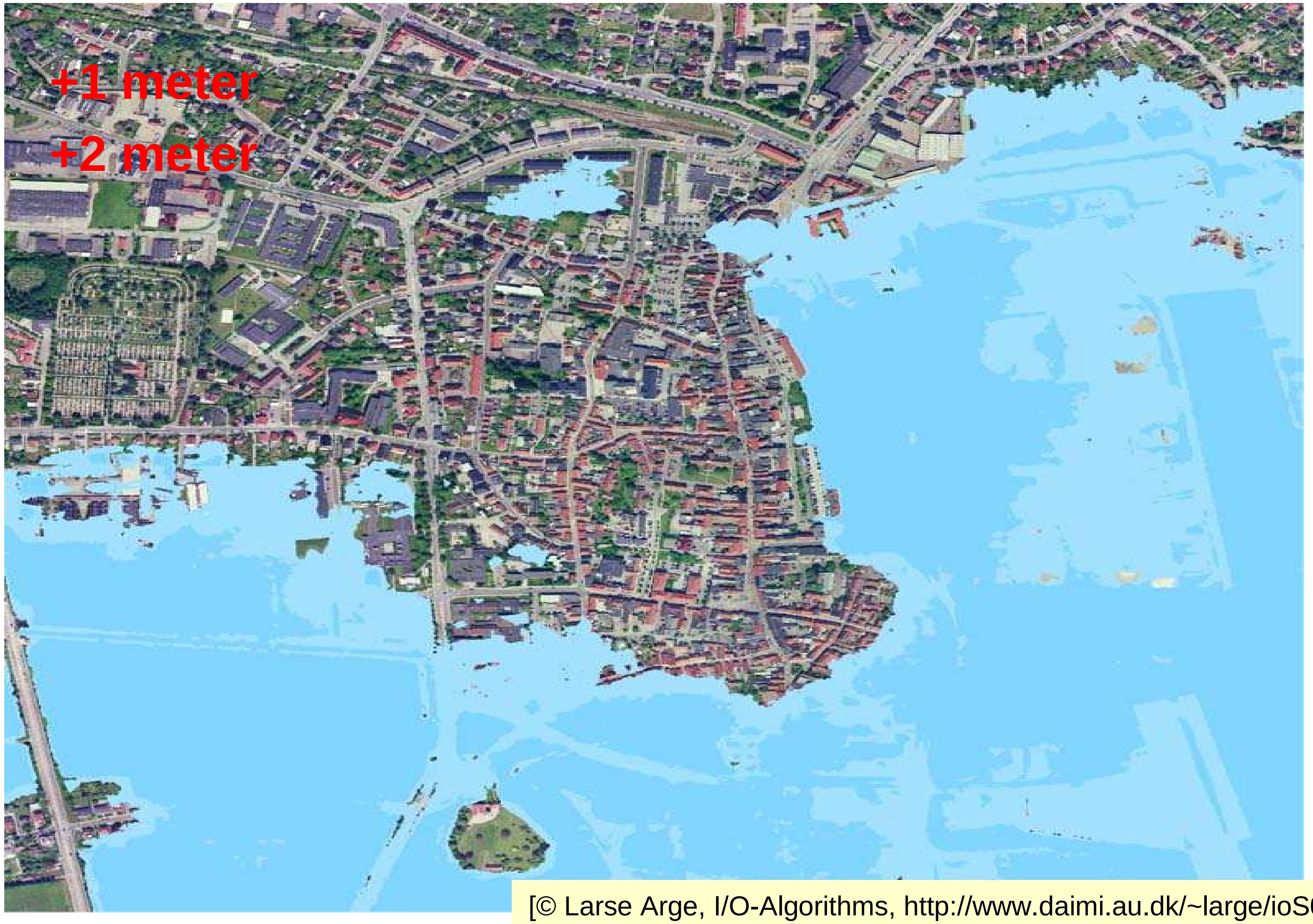
Example: LIDAR Terrain Data

COWI A/S (and others) is currently scanning Denmark



[© Larse Arge, I/O-Algorithms, http://www.daimi.au.dk/~large/ioS07/]

Application Example: Flooding Prediction



[© Larse Arge, I/O-Algorithms, http://www.daimi.au.dk/~large/ioS07/]