technische universität
dortmund

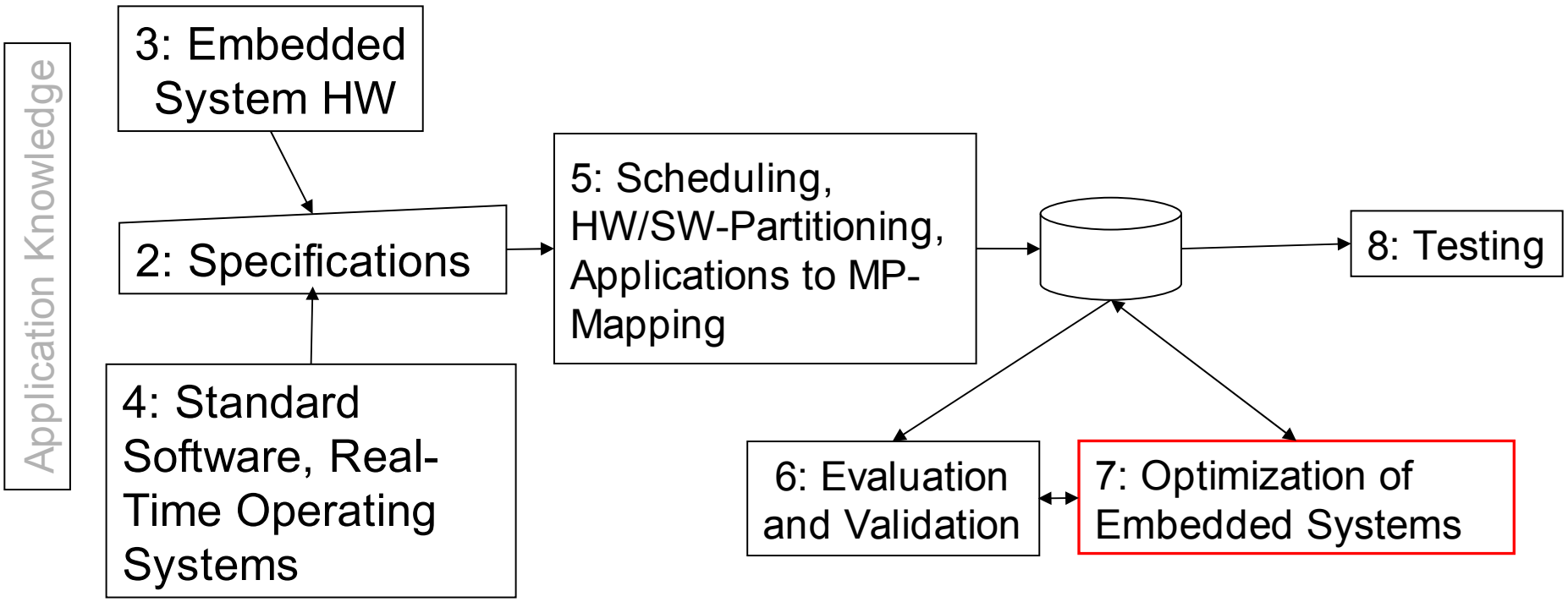
fakultät für informatik
informatik 12

Optimizations

Peter Marwedel
TU Dortmund
Informatik 12
Germany

2009/01/17

Structure of this course

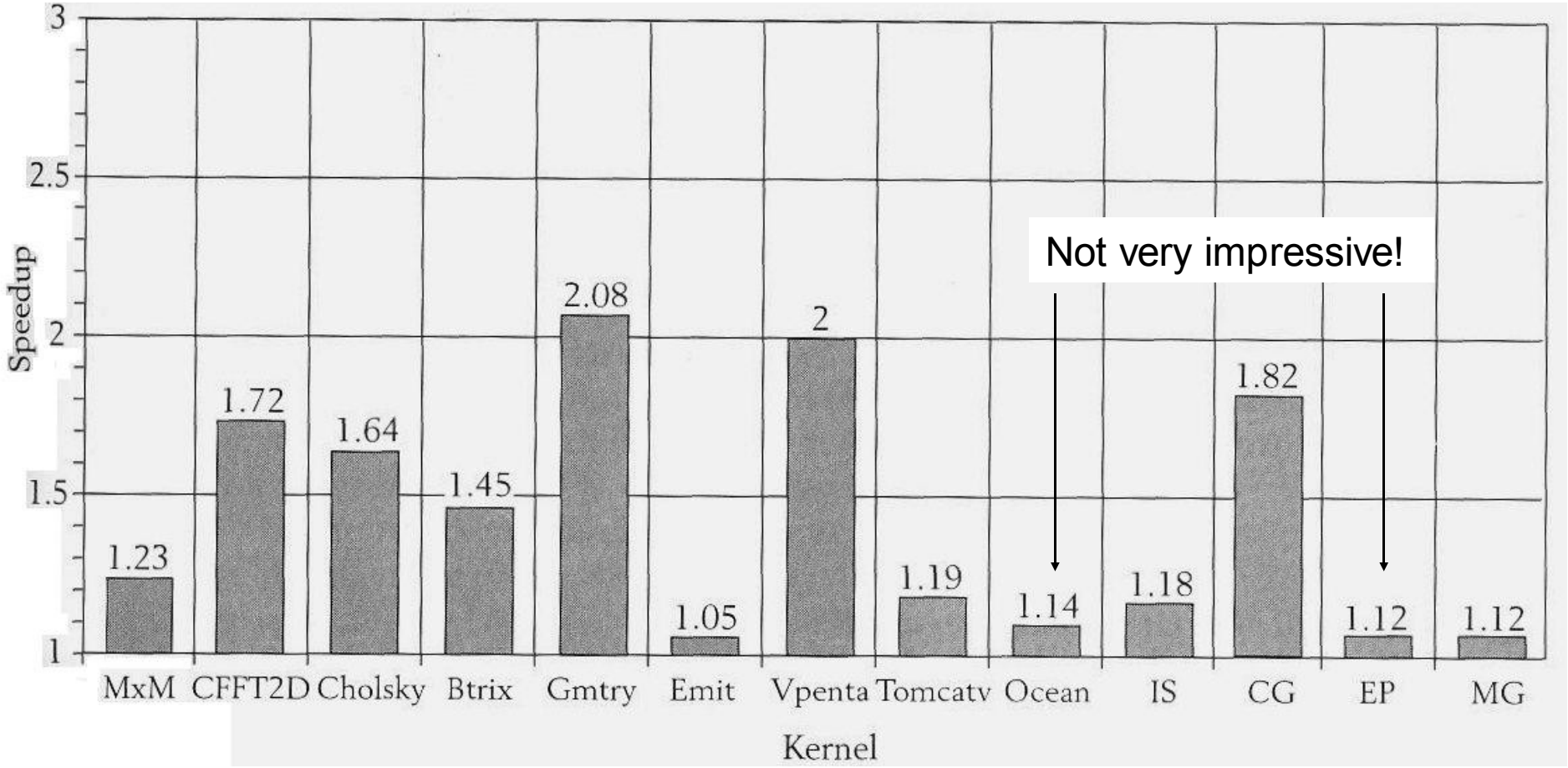


Prefetching

- 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

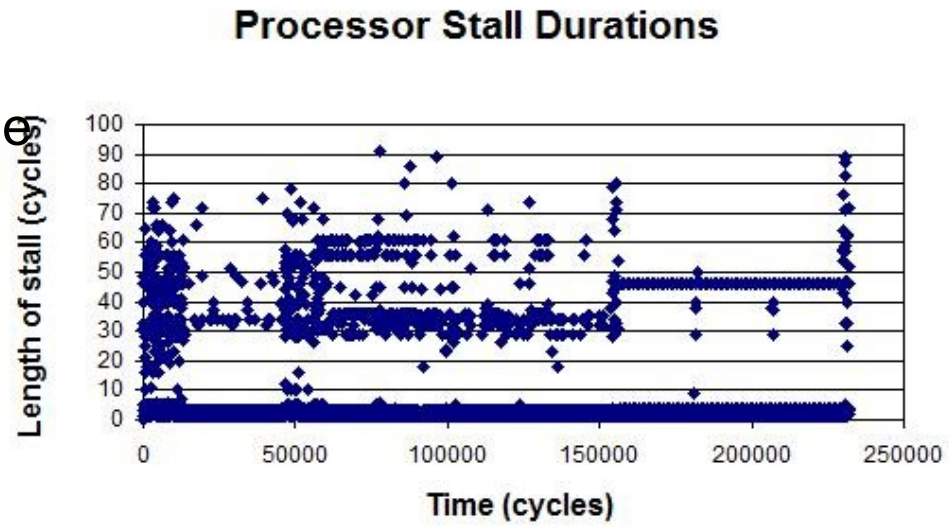
Break-even stall duration for profitable switching

- 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]

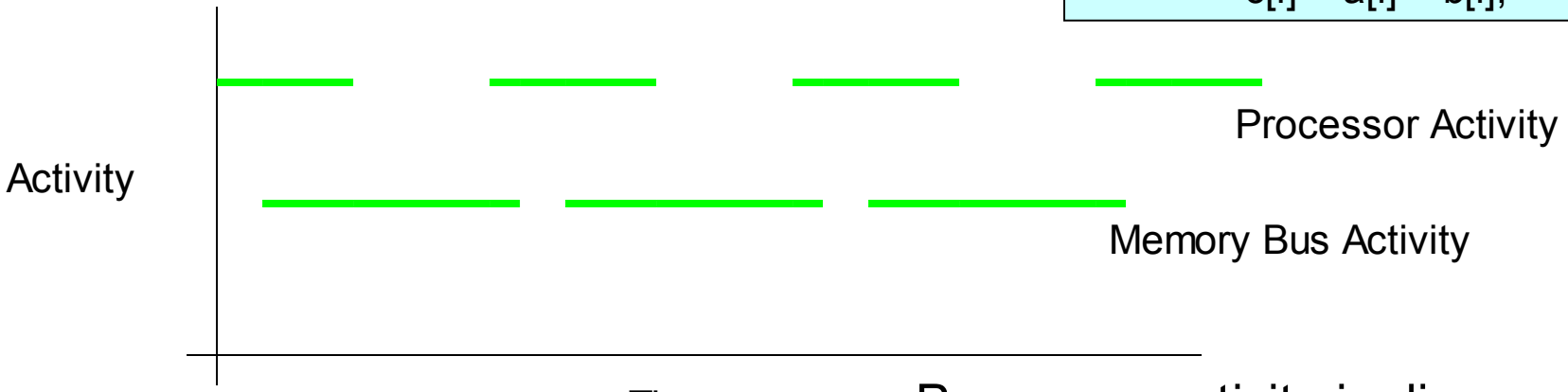
Optimization for exploiting processor-memory interface: Problem Definition (2)

- 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];
```

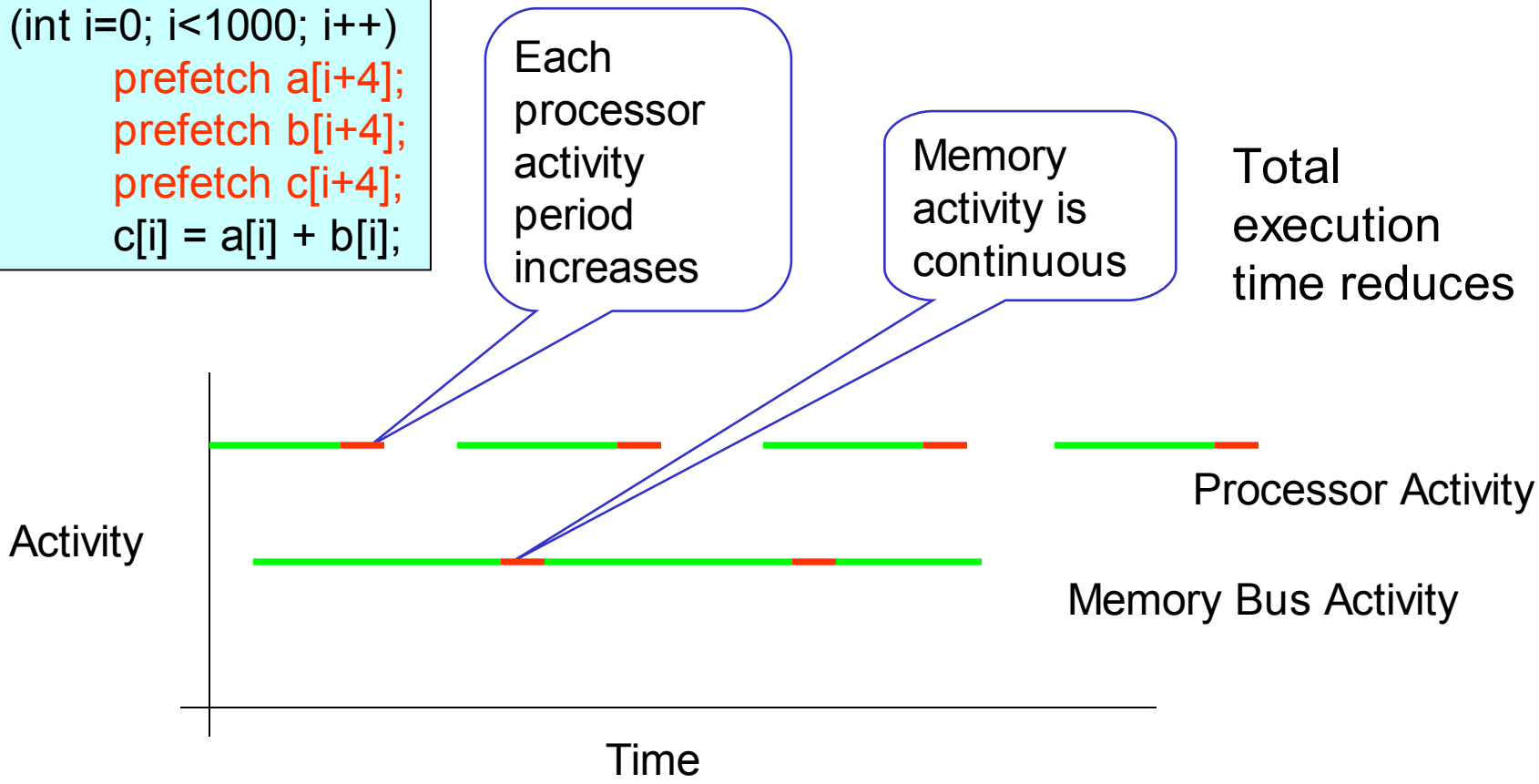


Processor activity is dis-continuous

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];
```

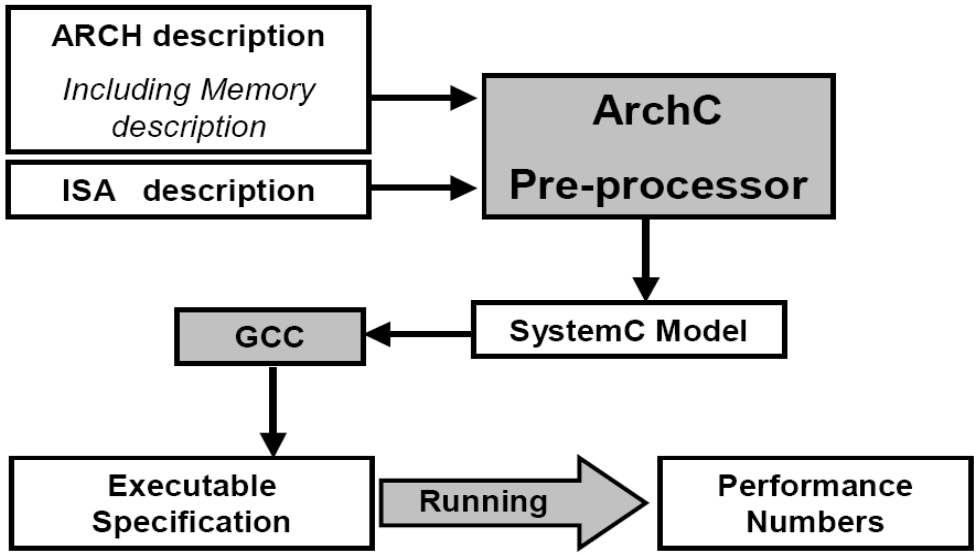


Processor activity is dis-continuous

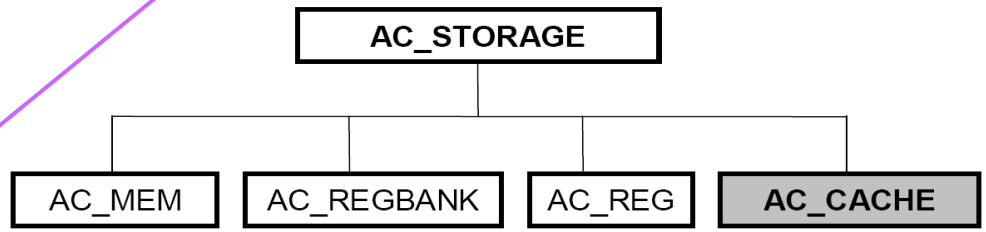
Memory activity is continuous

Memory hierarchy description languages: ArchC

Consists of description of ISA and HW architecture
Extension of SystemC (can be generated from ArchC):

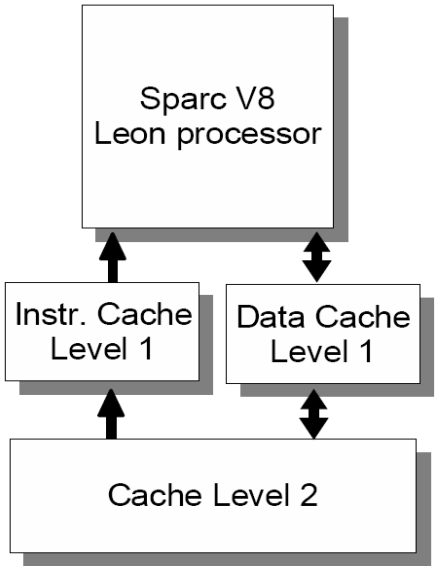


Storage class structure



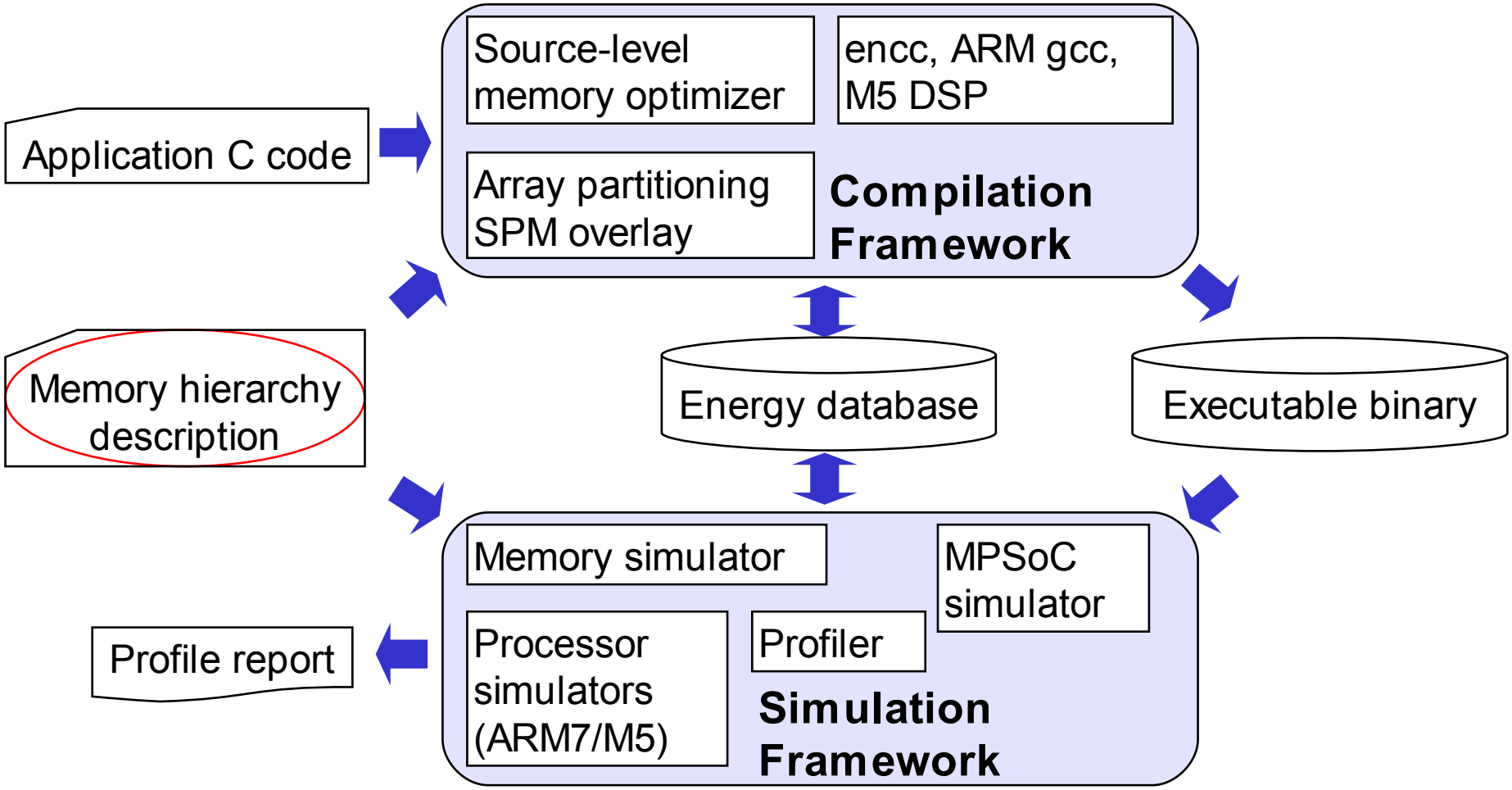
[P. Viana, E. Barros, S. Rigo, R. Azevedo, G. Araújo: Exploring Memory Hierarchy with ArchC, *15th Symposium on Computer Architecture and High Performance Computing*, 2003, pp. 2 – 9]

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

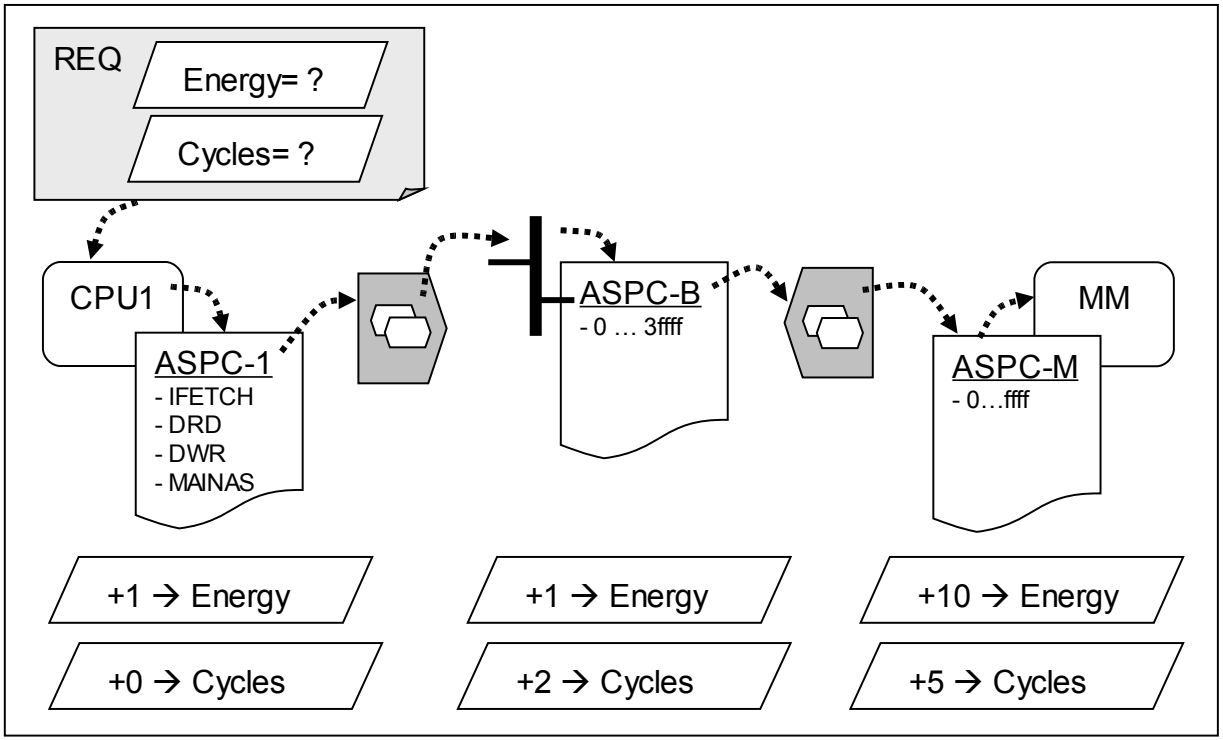


[M. Verma, L. Wehmeyer, R. Pyka, P. Marwedel, L. Benini: Compilation and Simulation Tool Chain for Memory Aware Energy Optimizations, *Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS VI)*, 2006].

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

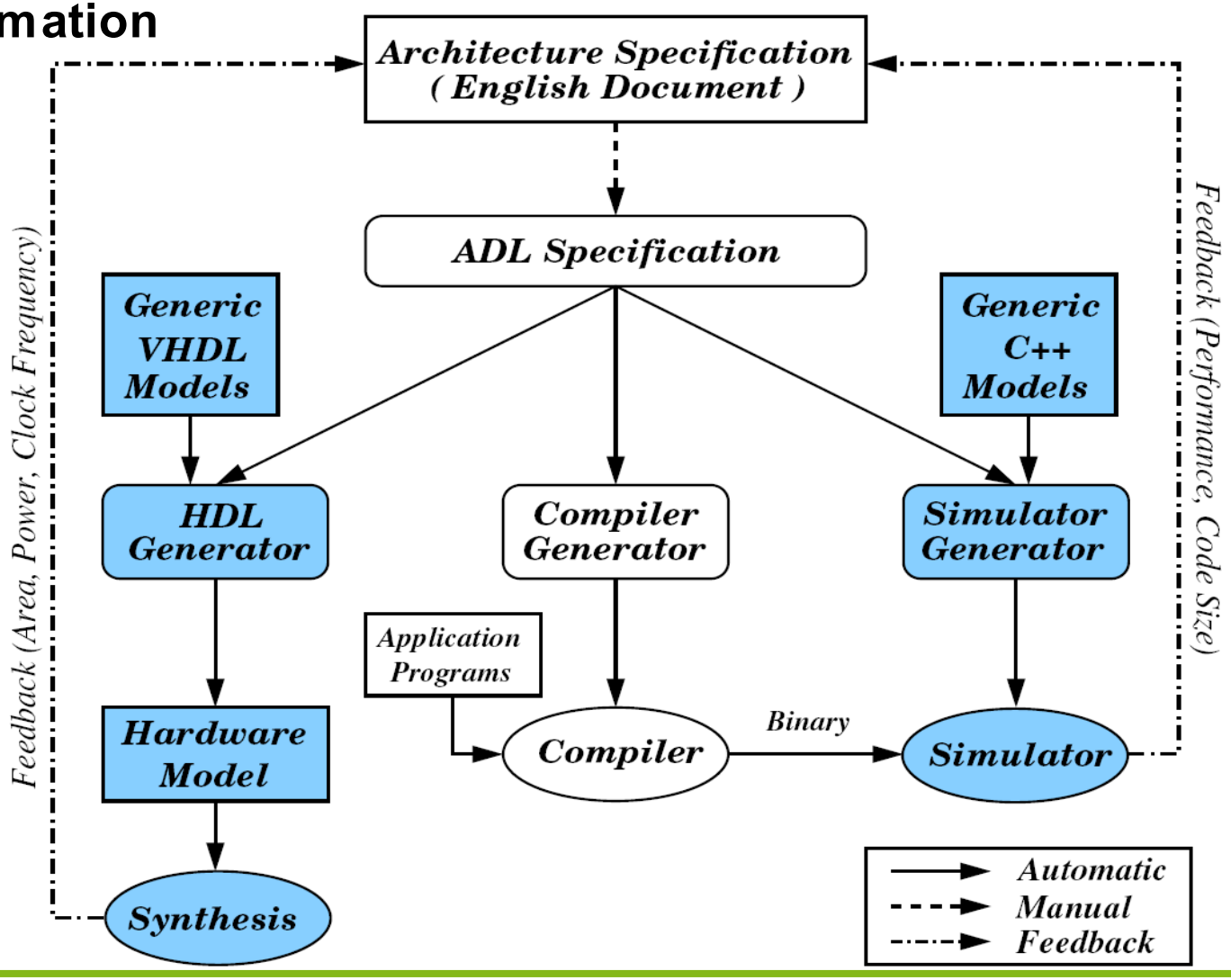
- API query to model simplifies integration into compiler
- External XML representation



[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

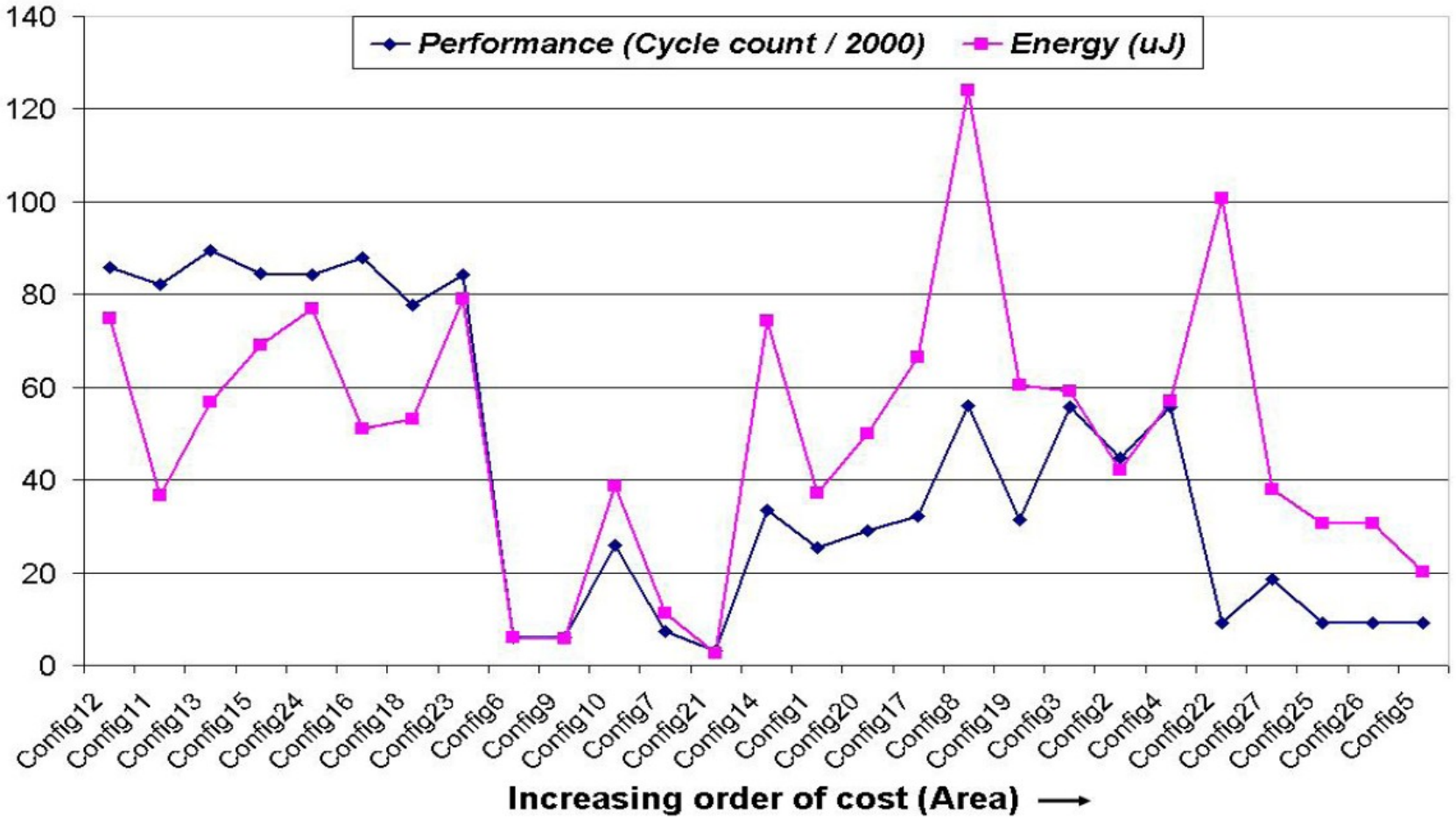
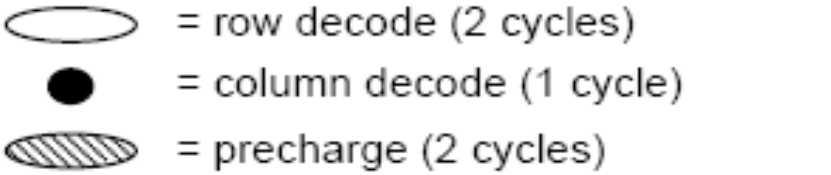


Figure 4.11: Memory exploration results for GSR

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



(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

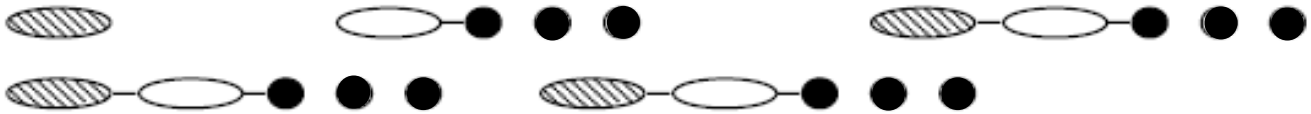
```
for(i=0; i<9;i+=3){
a=a+x[i]+x[i+1]+x[i+2]+
      y[i]+y[i+1]+y[i+2];
b=b+z[i]+z[i+1]+z[i+2]+
      u[i]+u[i+1]+u[i+2];}
```

(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).

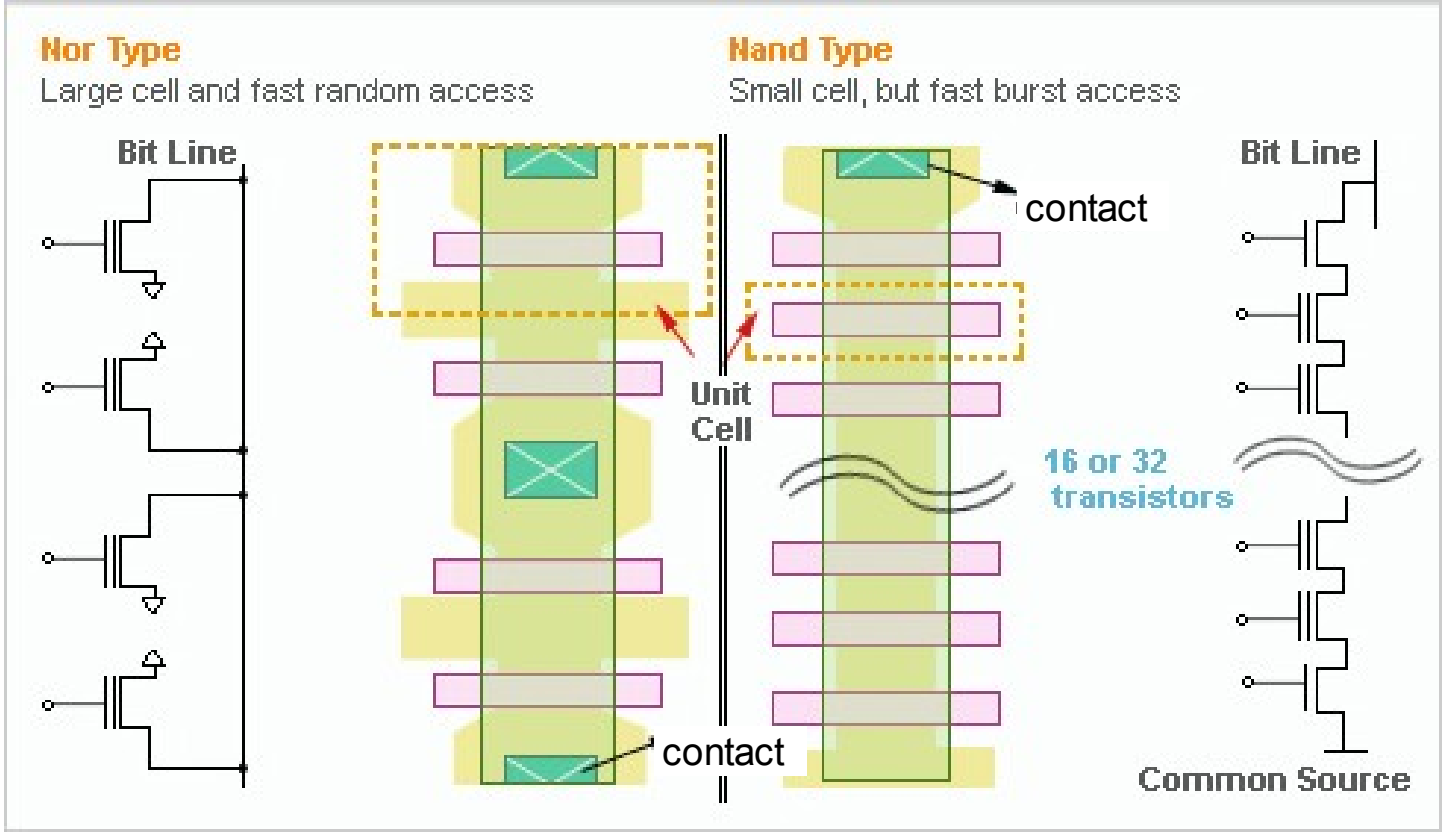
Comparison Flash/Microdrive

	Sandisk Type I Flash	Sandisk Type II Flash	IBM Microdrive DSCM-10340
Capacity [MB]	64	300	340
Power [W] (standby/operating)	0,15/0.66	0,15/0,66	0,07/0.83
Write cycles	300.000	300.000	unlimited
Mean-time between failures [h]	>1.000.000	>1.000.000	service-life=min(5J, 8800 h operating)
Error rates, uncorrectable	< 1 per 10^{14}	<1 per 10^{14}	<1 per 10^{13}
Max. power ons	unlimited	unlimited	300.000
Shock tolerance	2000 G; 2000 G	2000 G;	175 G; 1500 G

Source: Hennessy/Patterson, Computer Architecture, 2002

NOR- and NAND-Flash

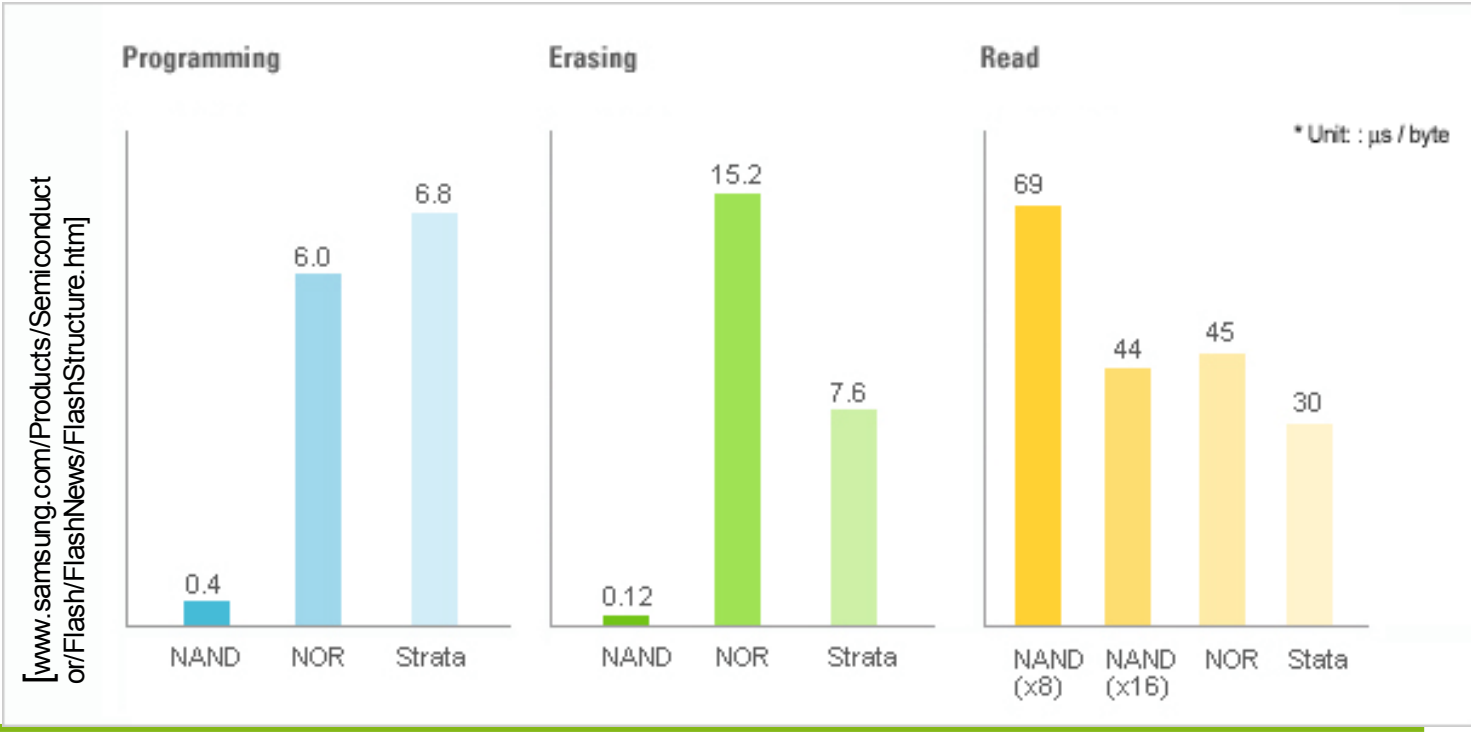
NOR: Transistor between bit line and ground
NAND: Several transistor between bit line and ground



was at [www.samsung.com/Products/ Semiconductor/ Flash/FlashNews/ FlashStructure. htm] (2007)

Properties of NOR- and NAND-Flash memories

Type/Property	NOR	NAND
Random access	Yes ☺	No ☹
Erase block	Slow ☹	Fast ☺
Size of cell	Larger ☺	Small ☺
Reliability	Larger ☺	Smaller ☹
Execute in place	Yes ☺	No ☹
Applications	Code storage, boot flash, set top box	Data storage, USB sticks, memory cards



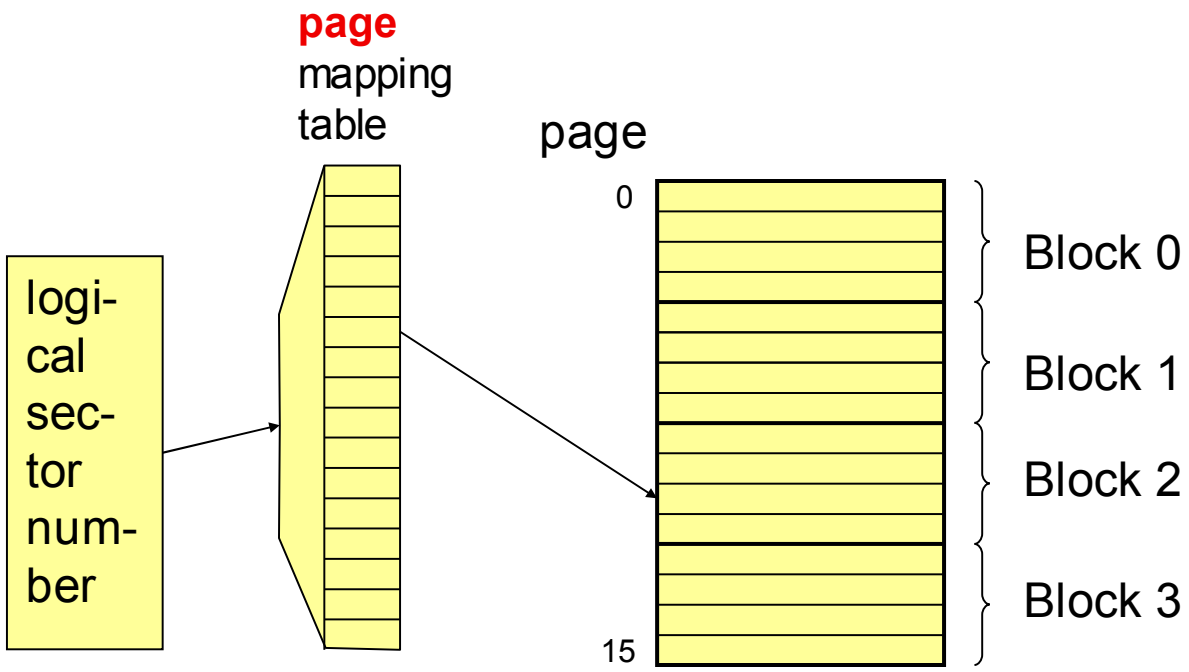
Characteristics of NAND Flash memory

Memory partitioned into blocks (typ. 16-256 KB), blocks partitioned into pages (typ. 0.5-5 KB). Read/write operations performed in page units.

	Single Level Cell (SLC)	Multi Level Cell (MLC)
Read (page)	25 µs	≫ 25 µs
Write (page)	300 µs	≫ 300 µs
Erase (block)	2 ms	1.5 ms

J. Lee, S. Kim, H. Kwin, C. Hyun, S, Ahn, J. Choi, D. Lee, S.Noh: Block Recycling Schemes and Their Cost-based Optimization in NAND Flash Memory Based Storage System, EMSOFT'07, Sept. 2007

Page/sector mapping flash transaction layer (FTL)

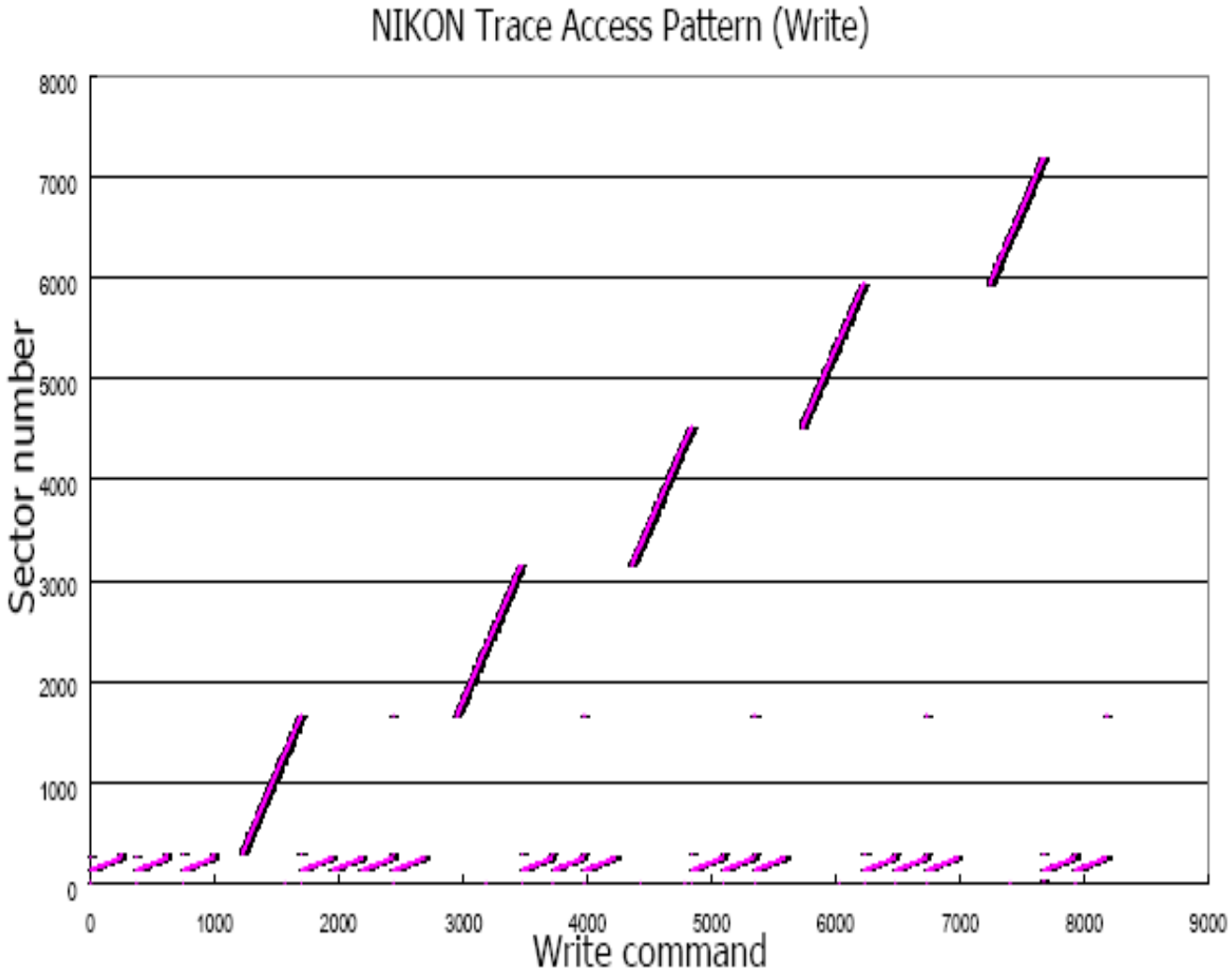


Inverted page table stored in flash memory (extra bits);
"normal page" table constructed during initialization.
Page table may become large
Used in low capacity NOR Flash memories

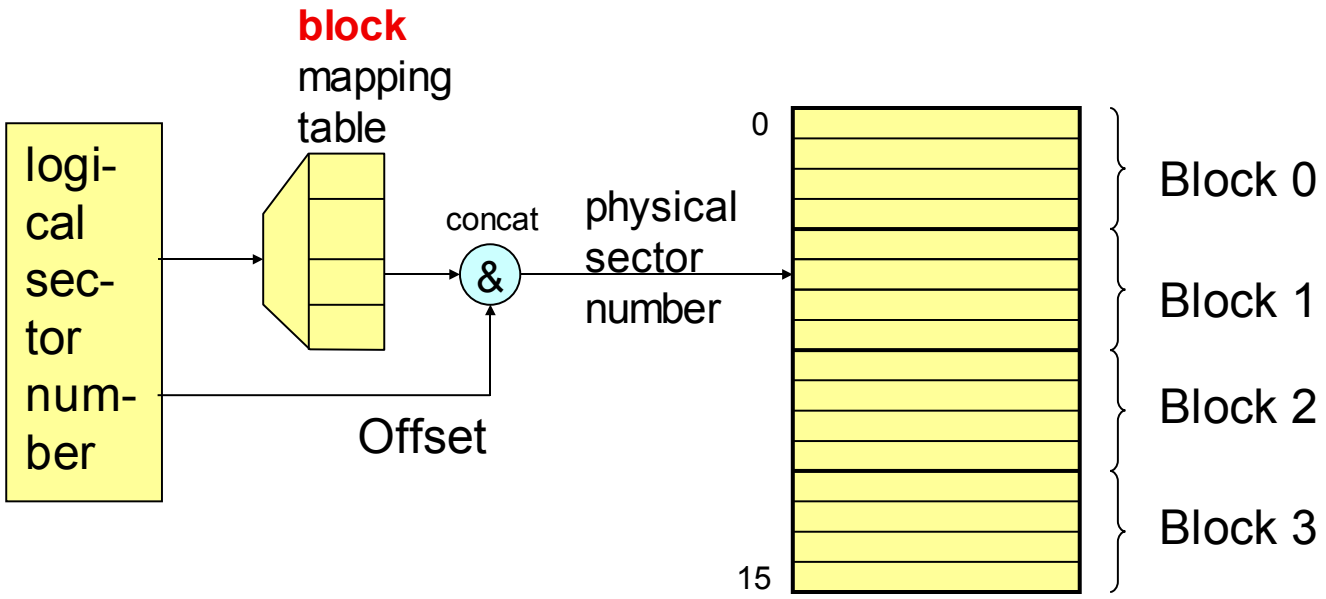
sector ≈ page
+ extra bits

Exploiting regularity

Usually, long sequence of sequential writes



Block mapping flash transaction layer (FTL)



- Mapping tables smaller than for page-based FTLs.

☞ used in high capacity NAND Flash memories

- Overall operation is simple,
- but successive writes require copying into a new block
- Degraded performance for random and repeated writes.

☞ Hybrid schemes

Wear-leveling

- Example (Lofgren et al., 2000, 2003):
  - Each erase unit carries erase counter
  - One erase unit set aside as a spare
  - When one of the most worn out units is reclaimed, its counter is compared to least-worn out unit. If Δ is large:
    - content of least-worn-out ($\approx$ constants) $\rightarrow$ spare
    - content of most worn-out $\rightarrow$ least worn-out
    - most worn-out unit becomes the new spare

  Counter increment may be lost if power is lost between erase and counter update

☞ Attempts to avoid erase counter in the same erase unit

Source: Gal, Toledo, *ACM Computing Surveys*, June 2005

Flash-specific file systems

- Two-layer approach can be inefficient:
  - FTL emulates flash as a magnetic disc
  - Standard file system assumes magnetic disc

  Example: deleted sectors not marked ☞ not reclaimed
- Log-structured file systems just append new information
  - For disc-based file system:
    - Fast writes
    - Slow reads (head movement for gather operations)
  - Ideal for flash-based file system:
    - Writes done in new sectors
    - Reads not slow: no head movement

☞ Specific log-based flash file systems
- JFFS2 (NOR)
- YAFFS (NAND)

Source: Gal, Toledo, *ACM Computing Surveys*, June 2005

Flash-aware application data structures

- Direct use of flash-specific properties in applications
  - Typically requires partitioning of the flash memory and possibly wasted space within partitions
- Execute-in-place
  - Used with NOR-flash, directly addressable by processor
  - Problematic in systems without MMU (no FTL feasible!):
    - instructions must be stored contiguously in flash
    - instructions cannot move
  - Code needed during erase cannot be stored in flash, unless suspended writing or erasing feasible

Source: Gal, Toledo, *ACM Computing Surveys*, June 2005

Flash memory as main memory

One approach published (Wu, Zwaenepoel, 1994):

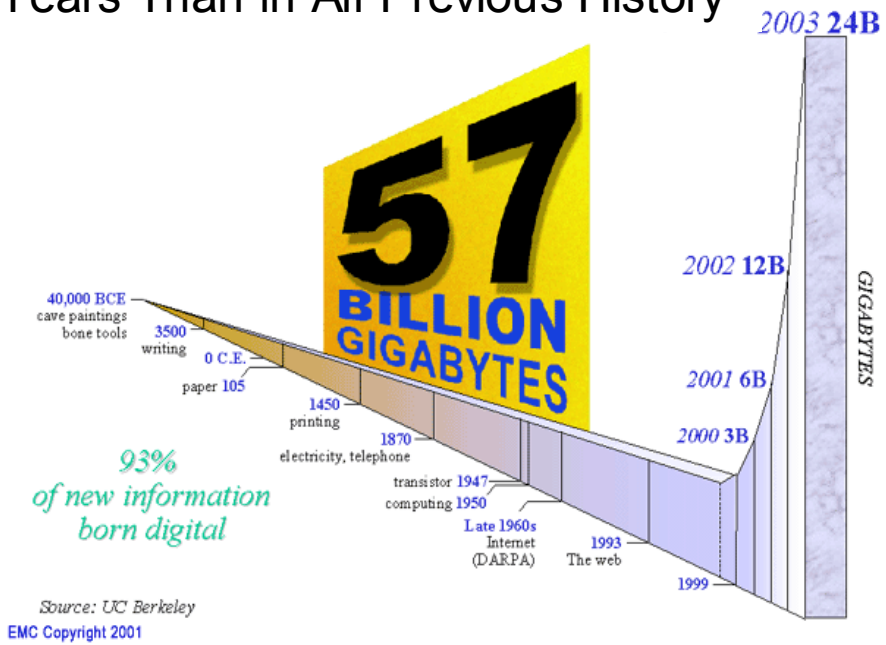
- Uses MMU
- RAM + Flash mapped to memory map
- Reads from Flash read single words from Flash
- Writes copy block of data into RAM,
  all updates done in RAM
- If the RAM is full, a block is copied back to Flash
- Crucial issue: Speed of writes.
  Proposal based on wide bus between Flash and RAM, so that writes are sufficiently fast
  ☞ Larger erase units, increased wear-out feasible.

M. Wu, W. Zwaenepoel: eNVy: A nonvolatile, main memory storage system. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*. 1994, p. 86–97.

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

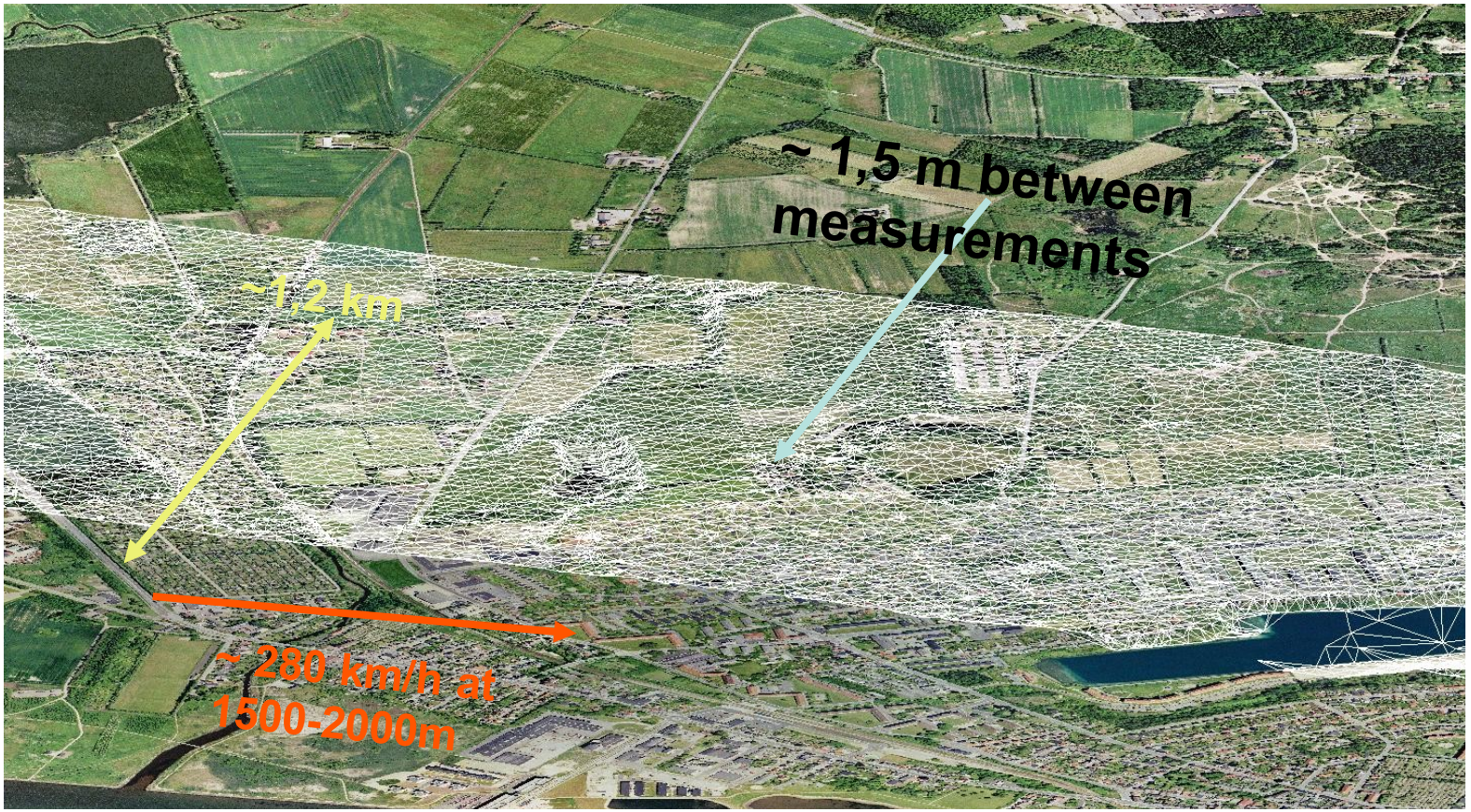
WEB: Web crawl of 200M pages and 2000M links, Akamai stores 7 billion clicks per day

Geography: NASA satellites generate 1.2TB per day

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

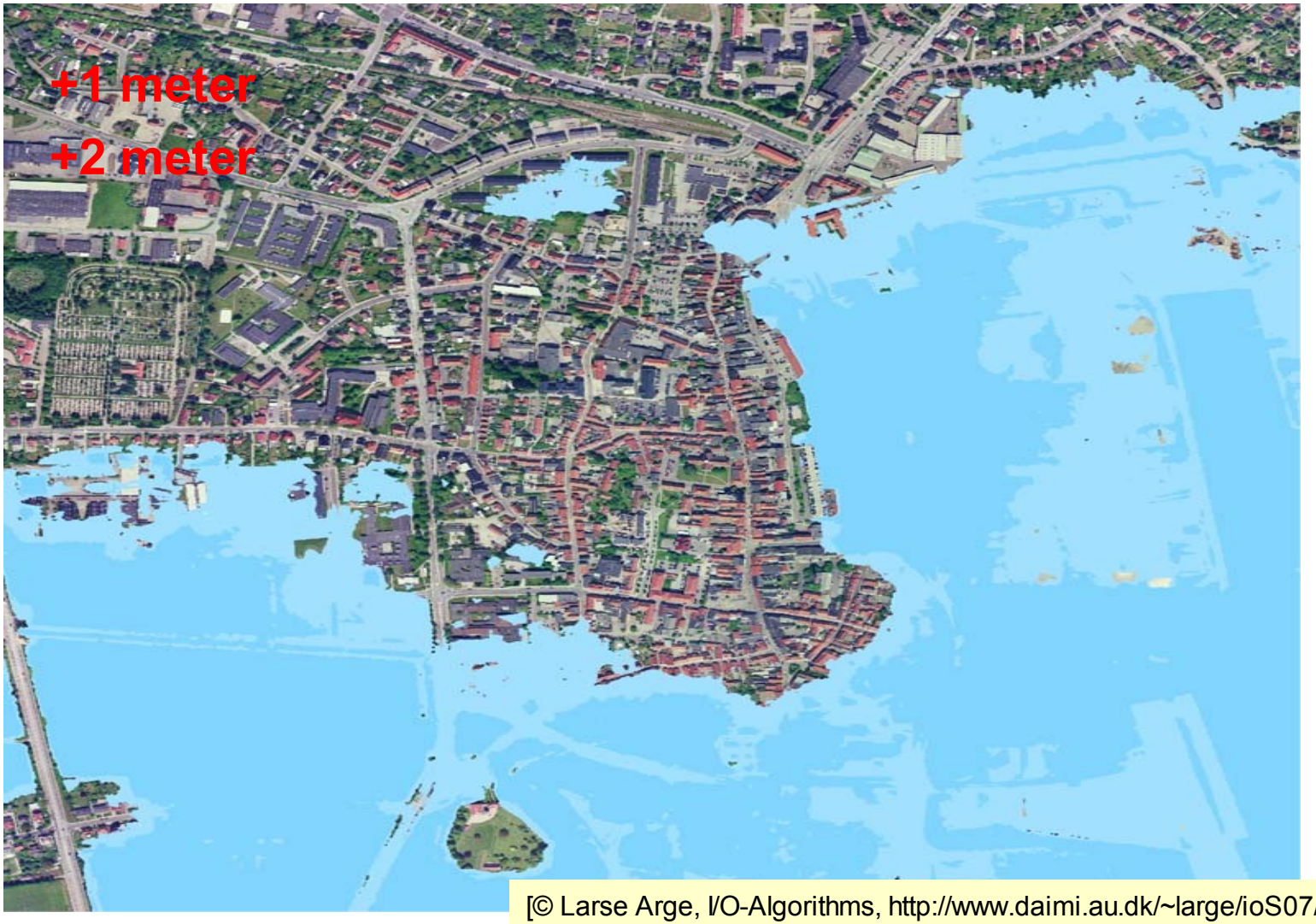
Example: LIDAR Terrain Data

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



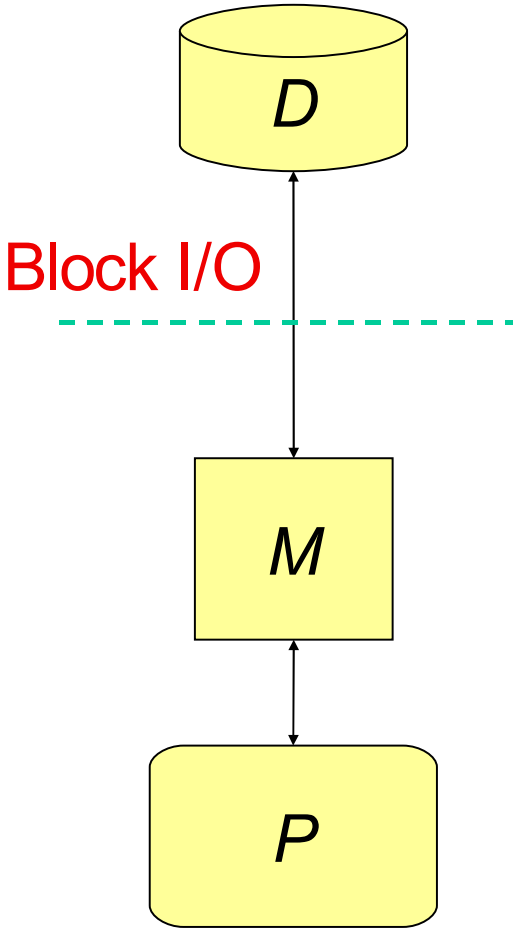
[© Larse Arge, I/O-Algorithms, http://www.daimi.au.dk/~large/ioS07/]

Application Example: Flooding Prediction



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

External Memory Model



N= # of items in the problem instance
B = # of items per disk block
M = # of items that fit in main memory

T = # of items in output

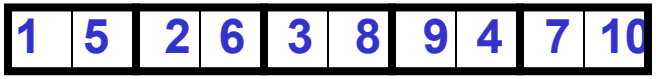
I/O: Move block between memory and disk

We assume (for convenience) that $M > B^2$

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

Scalability Problems: Block Access Matters

- Example: Reading an array from disk
  - Array size N = 10 elements
  - Disk block size B = 2 elements
  - Main memory size M = 4 elements (2 blocks)



Algorithm 1: N=10 I/Os

Algorithm 2: N/B=5 I/Os

- Difference between N and N/B large since block size is large
  - Example: $N = 256 \times 10^6$, $B = 8000$, $1ms$ disk access time

    $\Rightarrow$ ***N*** I/Os take 256×10^3 *sec* = *4266 min* = *71 hr*

    $\Rightarrow$ ***N/B*** I/Os take *256/8 sec* = *32 sec*

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

Re-writing algorithms for memory hierarchies

Analysis of algorithm complexity mostly using the *RAM* (random access machine; const. mem. acc. times) model outdated

☞ take memory hierarchies explicitly into account.

Example:

- Usually, divide-&-conquer algorithms are good.
- "Cache"-oblivious algorithms (are good for any size of the faster memory and any block size). Assuming
  - Optimal replacement (Belady's algorithm)
  - 2 Memory levels considered (there can be more)
  - Full associativity
  - Automatic replacement

Unlikely to be ever automatic

[Piyush Kumar: Cache Oblivious Algorithms, in: U. Meyer et al. (eds.): Algorithms for Memory Hierarchies, *Lecture Notes in Computer Science, Volume 2625*, 2003, pp. 193-212]

[Naila Rahman: Algorithms for Hardware Caches and TLB, in: U. Meyer et al. (eds.): Algorithms for Memory Hierarchies, *Lecture Notes in Computer Science, Volume 2625*, 2003, pp. 171-192]

Fundamental Bounds

	Internal	External
Scanning:	N	$\frac{N}{B}$
Sorting:	$N \log N$	$\frac{N}{B} \log_{M/B} \frac{N}{B}$
Permuting	N	$\min\{N, \frac{N}{B} \log_{M/B} \frac{N}{B}\}$
Searching:	$\log_2 N$	$\log_B N$

- Note:
  - Linear I/O: $O(N/B)$
  - Permuting not linear
  - Permuting and sorting bounds are equal in all practical cases
  - B factor VERY important: $\frac{N}{B} < \frac{N}{B} \log_{M/B} \frac{N}{B} << N$
- Which results apply to flash memory?

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

technische universität
dortmund

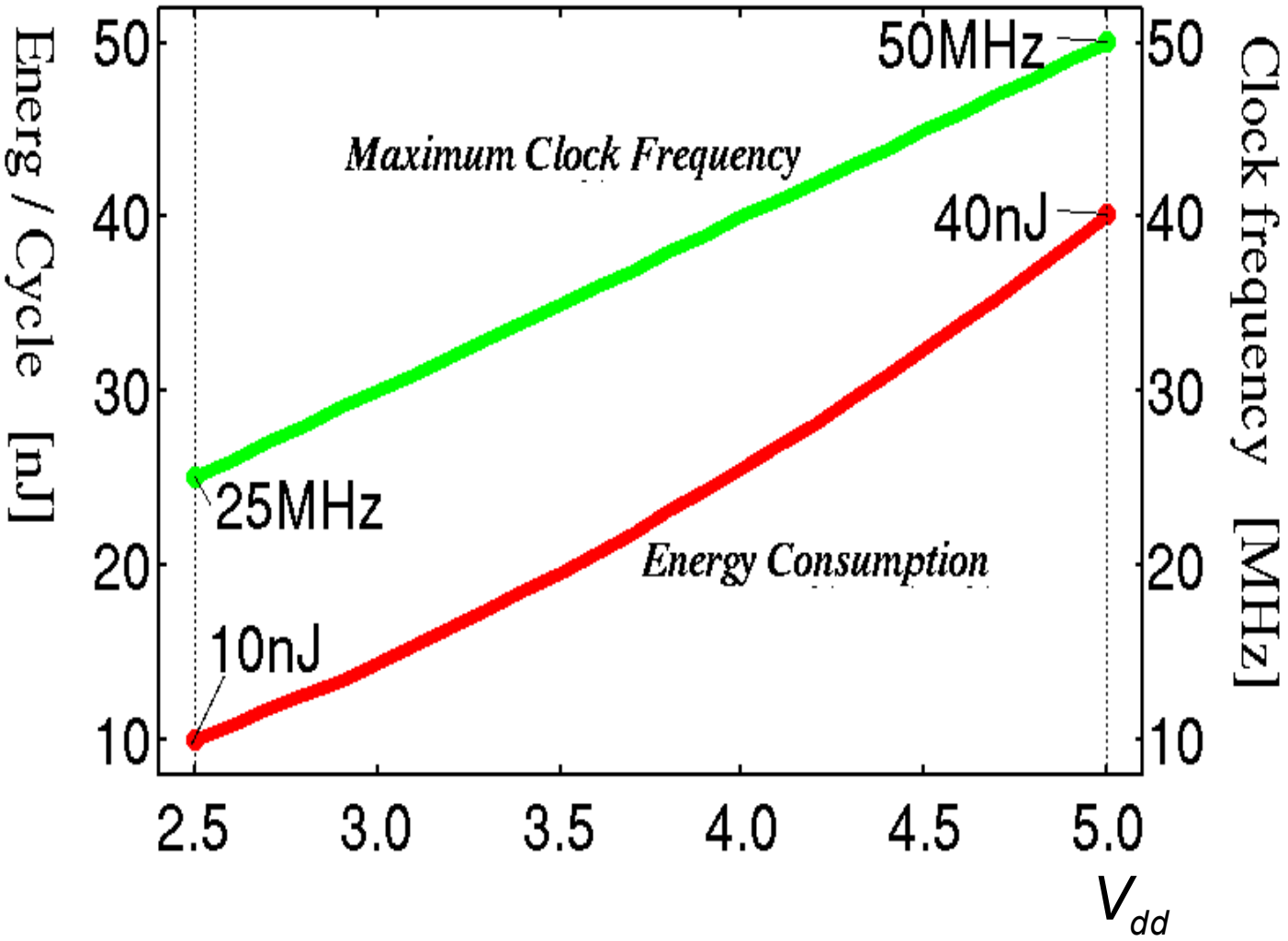
fakultät für informatik
informatik 12

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:

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

V_t : threshhold voltage

(V_t substancially < than V_{dd})

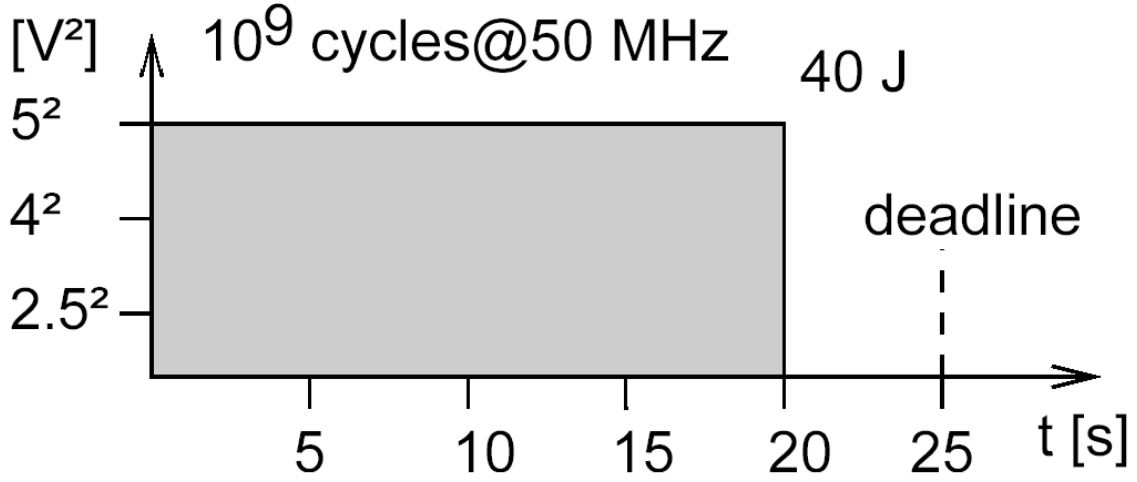
☞ Decreasing V_{dd} reduces P quadratically,
while the run-time of algorithms is only linearly increased
(ignoring the effects of the memory system).

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

a)

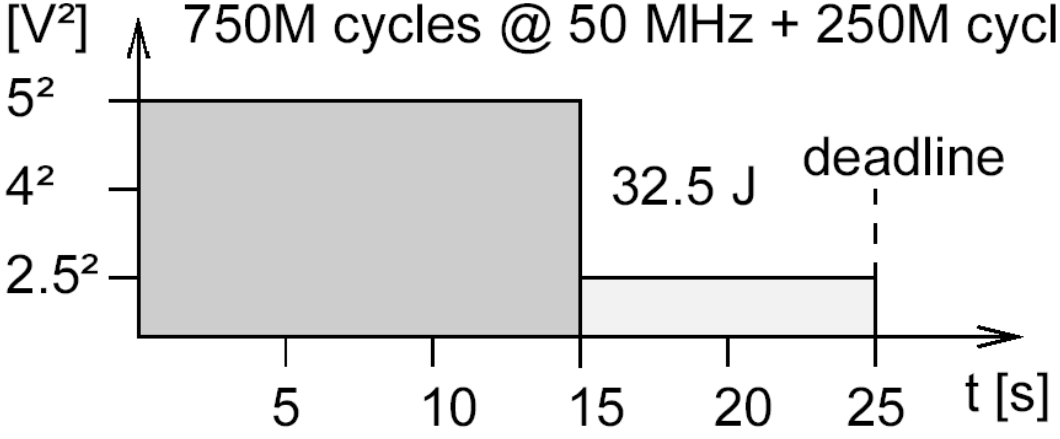


$E_a = 10^9 \times 40 \times 10^{-9}$
$= 40$ [J]

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) [V²] 750M cycles @ 50 MHz + 250M cycles @ 25

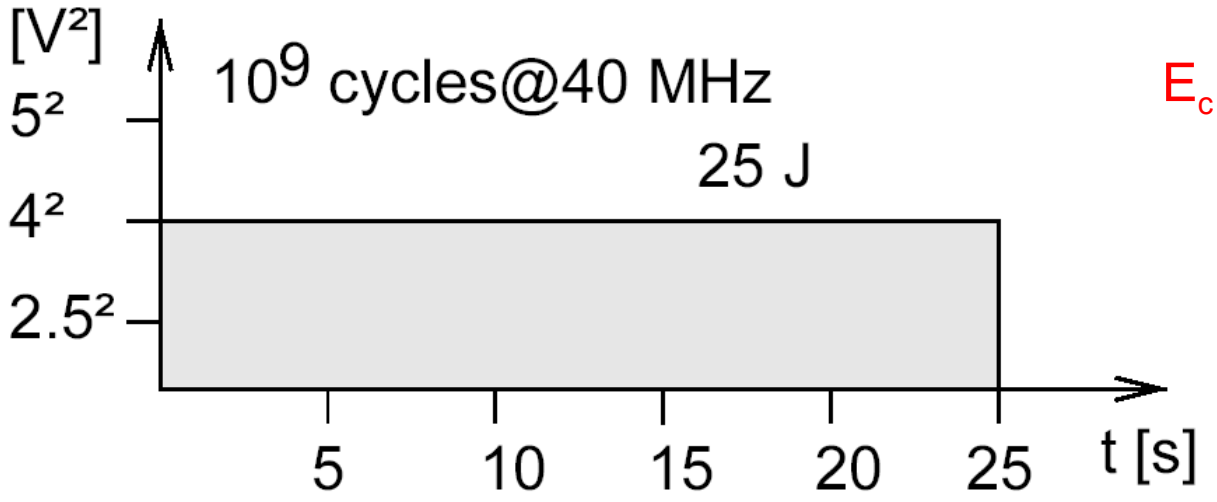


E_b= 750 10^6 x 40 10^{-9} +
250 10^6 x 10 10^{-9}
= 32.5 [J]

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)



$E_c = 10^9 \times 25 \times 10^{-9}$
$= 25$ [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 a voltage schedule with at most two voltages minimizes the energy consumption under any time constraint. If a processor can only use a number of discrete voltage levels, then the two voltages which minimize the energy consumption are the two immediate neighbors of the ideal voltage V_{ideal} possible for a variable voltage processor.

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_i comprises the actual capacitance CL and the switching activity α)

Designing an I(L)P model

Simplifying assumptions of the IP-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 Programming Model

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

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

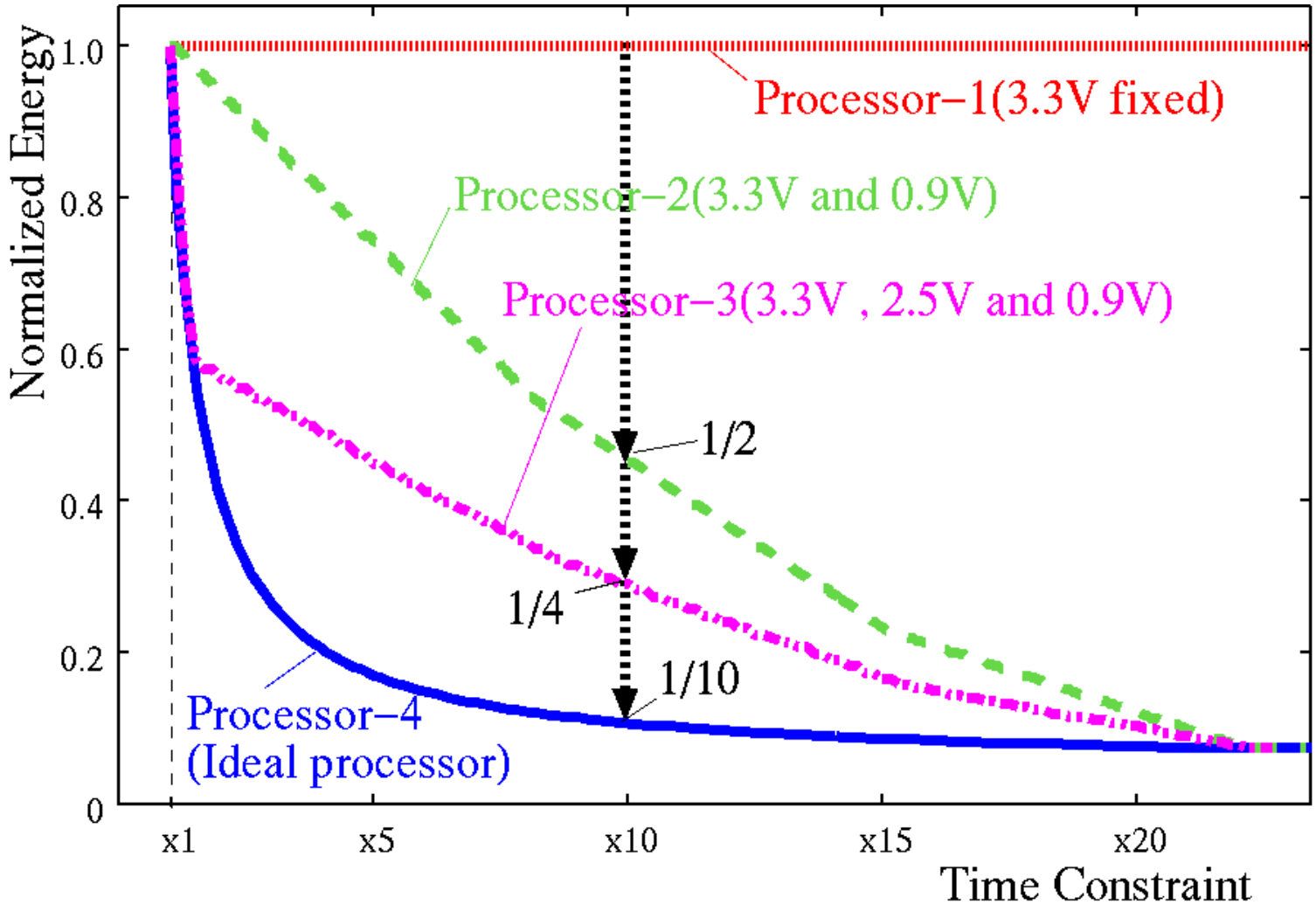
and $$\sum_{i=1}^{L} \sum_{j=1}^{N} \frac{x_{i,j}}{F_i} \leq T$$

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 , ….

Experimental Results



Summary

- Optimizations exploiting memory hierarchies
  - Prefetching
  - Memory-architecture aware compilation
    - Burst mode access exploited by EXPRESSION
  - Support for FLASH memory
  - Memory hierarchies beyond “main memory”
- Dynamic voltage scaling (DVS)
  - An ILP model for voltage assignment in a multi-tasking system
- Dynamic power management (DPM) (briefly)