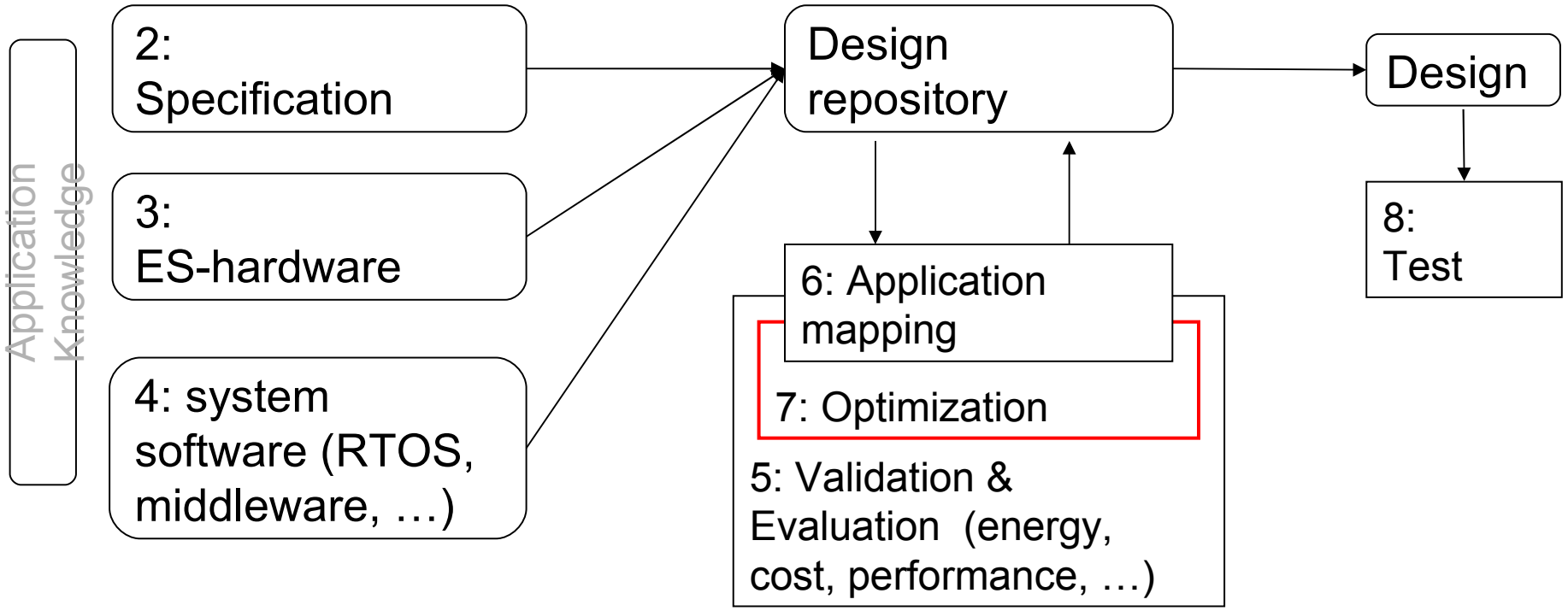
technische universität
dortmund

fakultät für informatik
informatik 12

Additional compiler optimizations

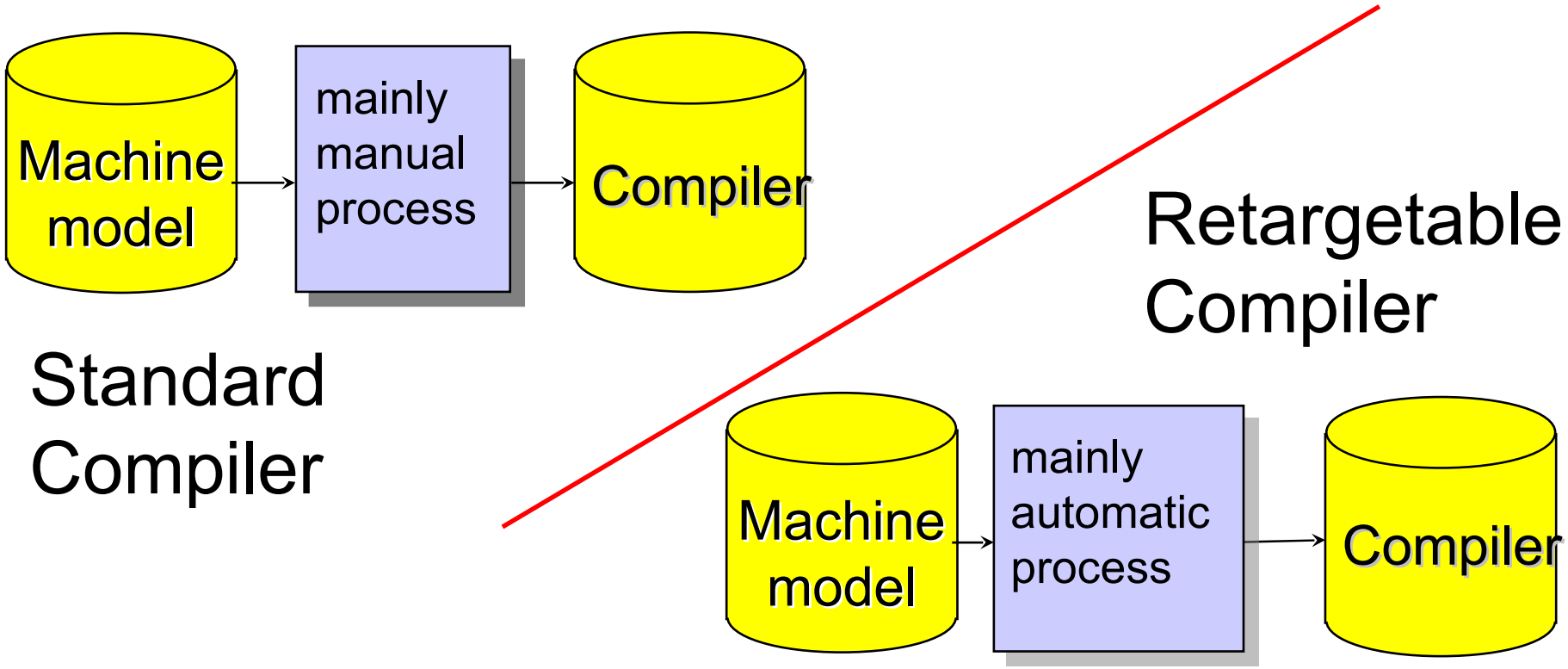
Peter Marwedel
TU Dortmund
Informatik 12
Germany

Structure of this course



Numbers denote sequence of chapters

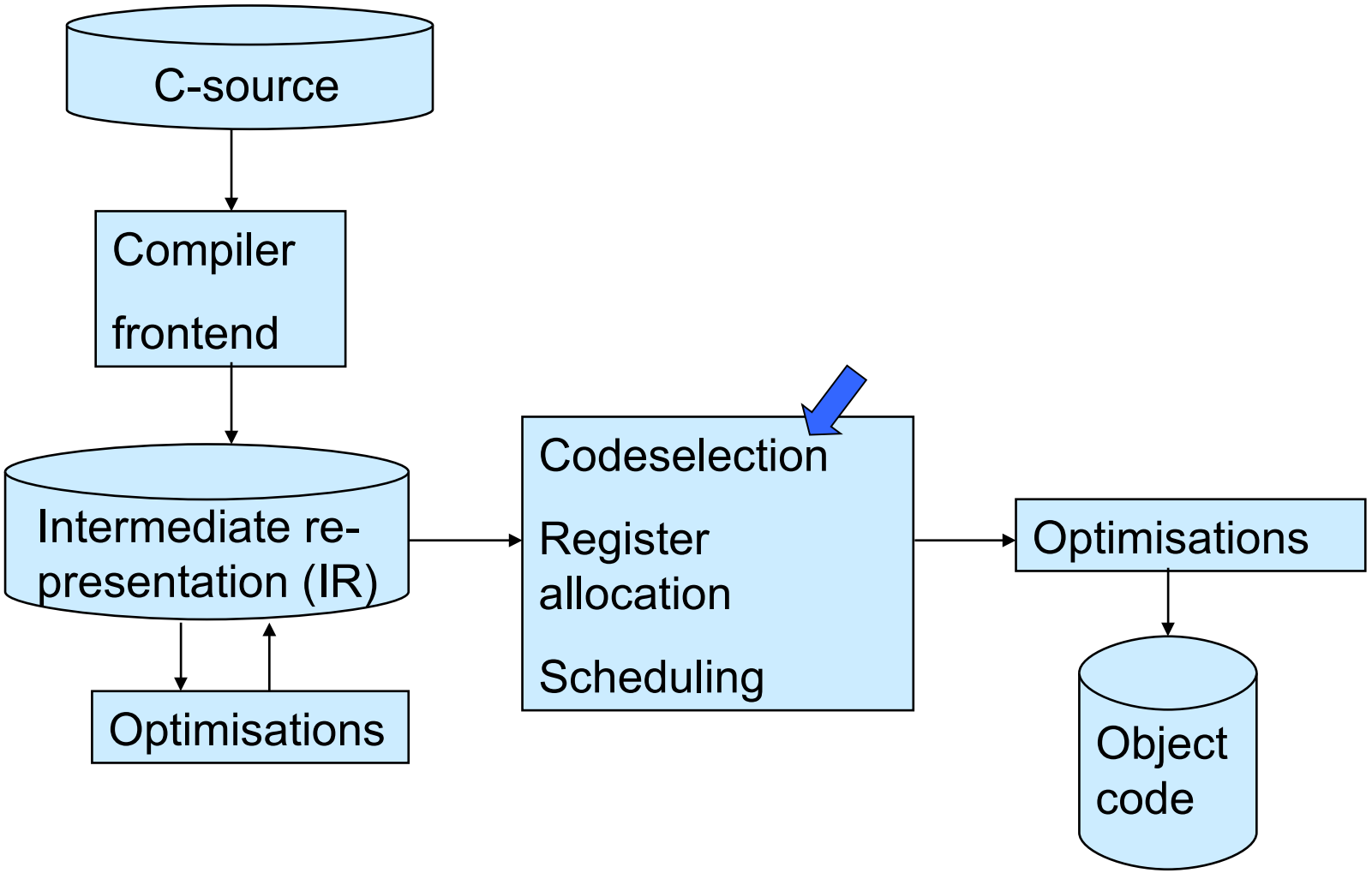
Retargetable Compilers vs. Standard Compilers



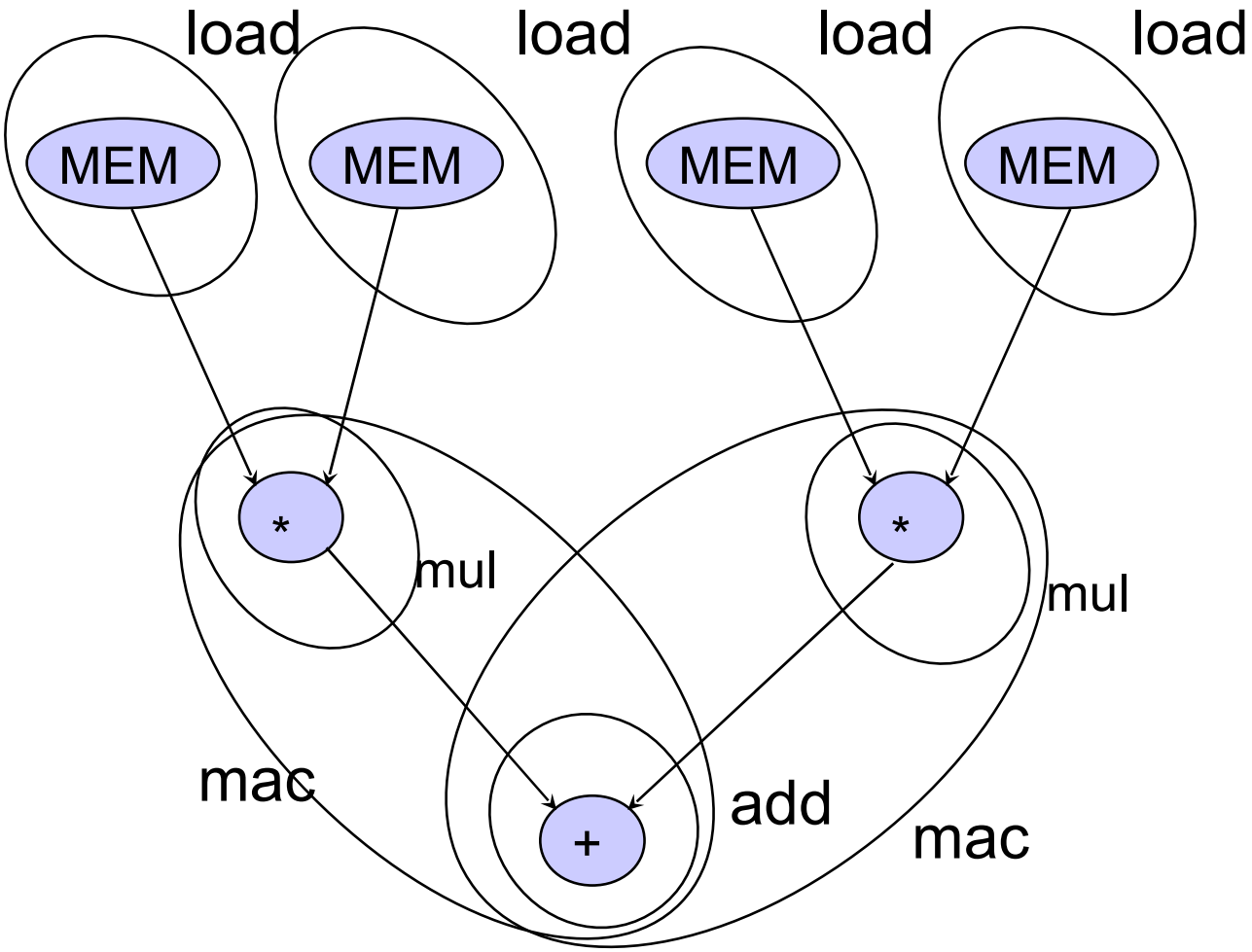
Developer retargetability: compiler specialists responsible for retargeting compilers.

User retargetability: users responsible for retargeting compiler.

Compiler structure



Code selection = covering DFGs



Does not yet consider data moves to input registers.

Code selection by tree parsing (1)

Specification of grammar for generating a iburg parser*:

terminals: {MEM, *, +}

non-terminals: {reg1,reg2,reg3}

start symbol: reg1

rules:

"add" (cost=2): reg1 -> + (reg1, reg2)

"mul" (cost=2): reg1 -> * (reg1,reg2)

"mac" (cost=3): reg1 -> + (*(reg1,reg2), reg3)

"load" (cost=1): reg1 -> MEM

"mov2"(cost=1): reg2 -> reg1

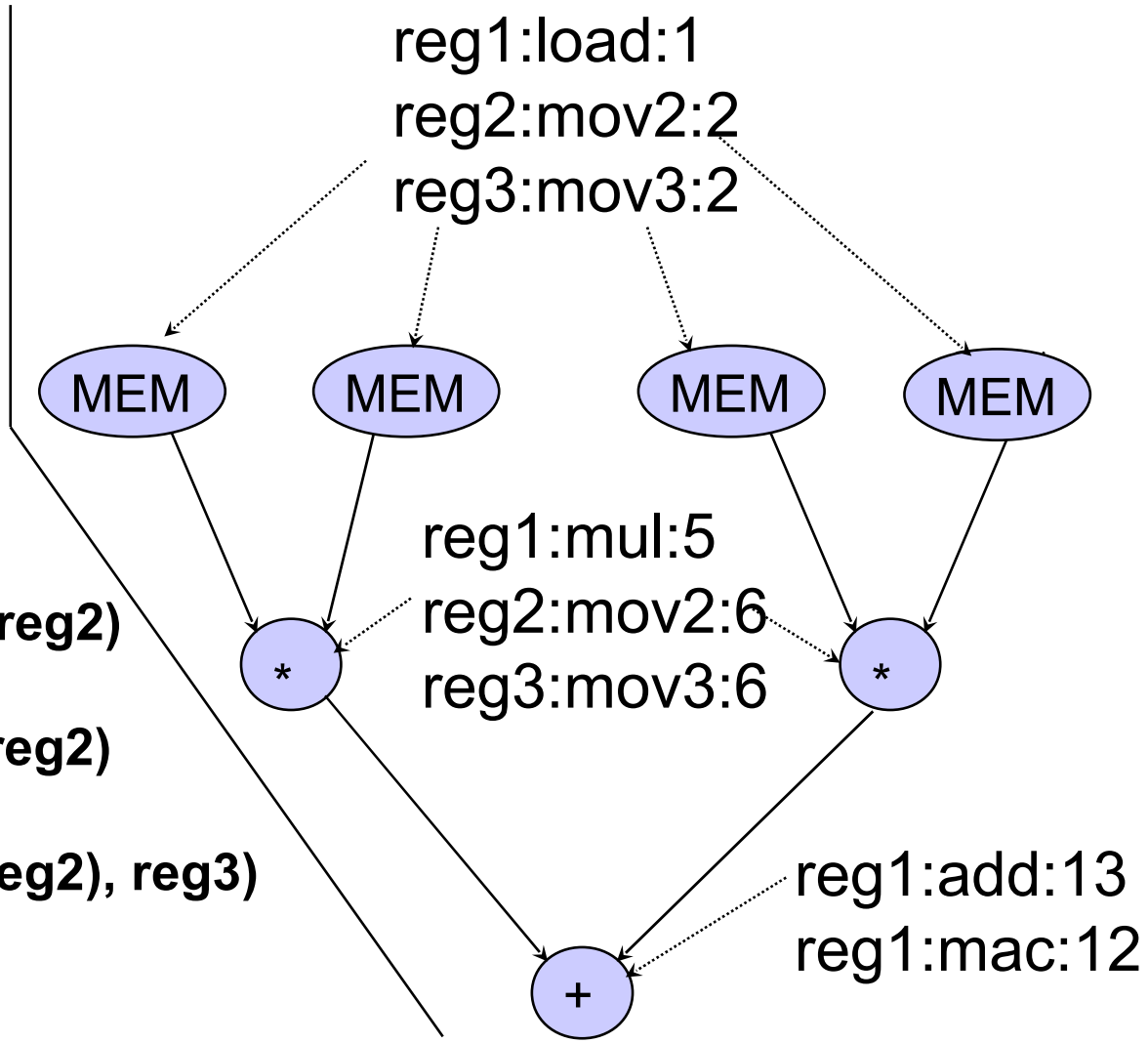
"mov3"(cost=1): reg3 -> reg1

Code selection by tree parsing (2)

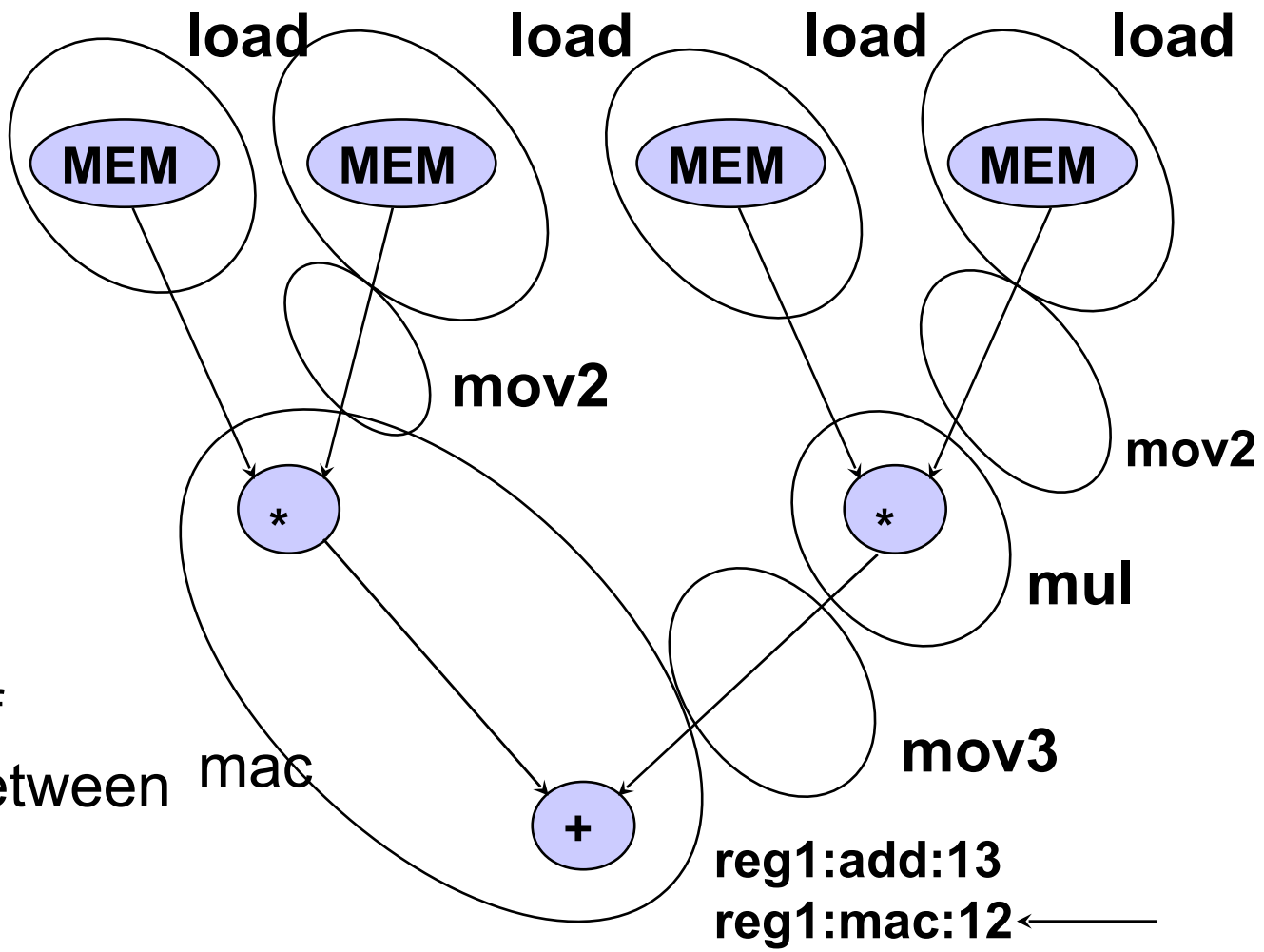
- nodes annotated with (register/pattern/cost)-triples -

“load”(cost=1):
reg1 -> MEM
“mov2”(cost=1):
reg2 -> reg1
“mov3”(cost=1):
reg3 -> reg1

“add” (cost=2):
reg1 -> +(reg1, reg2)
“mul” (cost=2):
reg1 -> *(reg1,reg2)
“mac” (cost=3):
reg1->+(*(reg1,reg2), reg3)



Code selection by tree parsing (3)
- final selection of cheapest set of instructions -



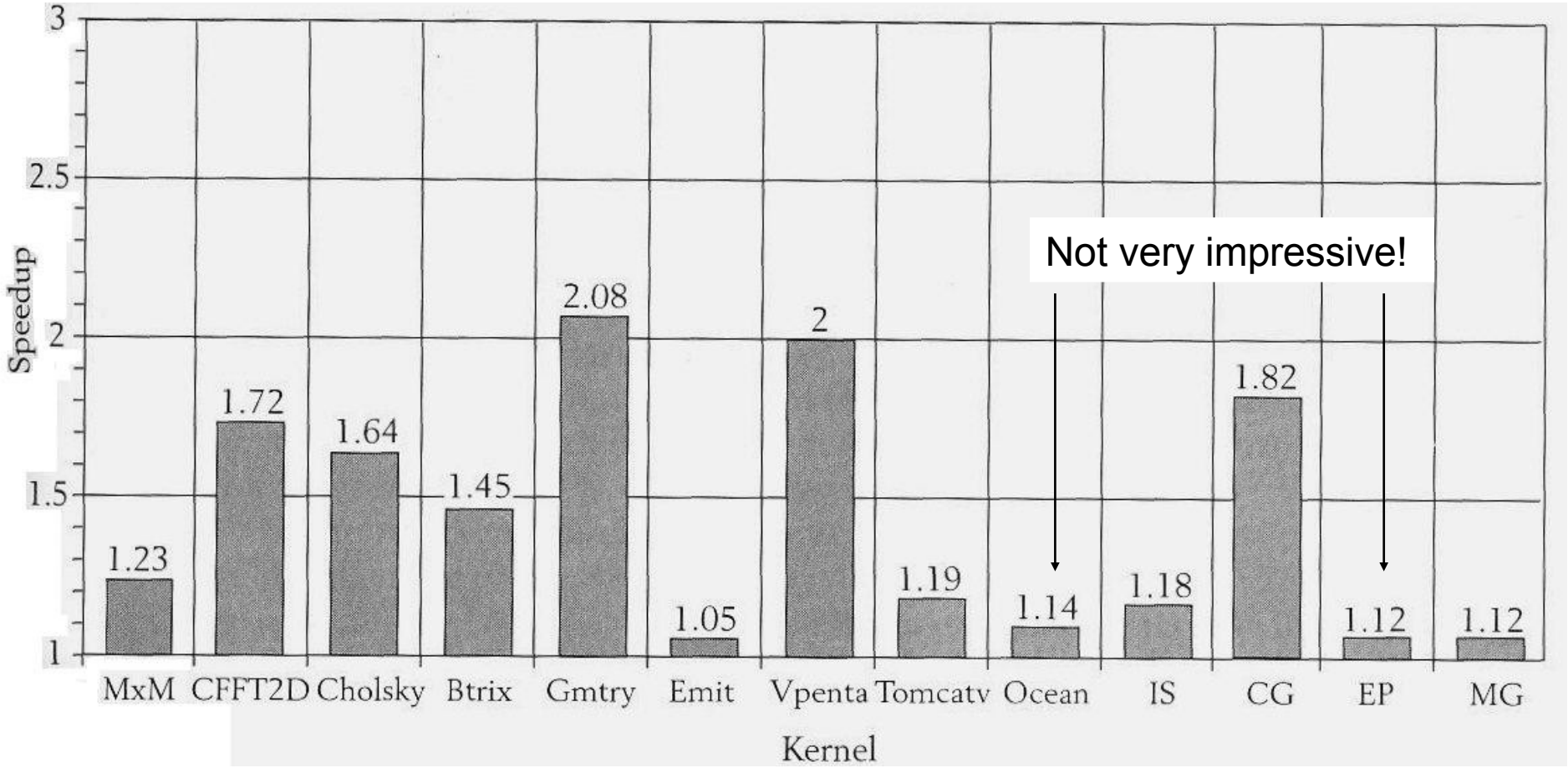
Includes routing of values between various registers!

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

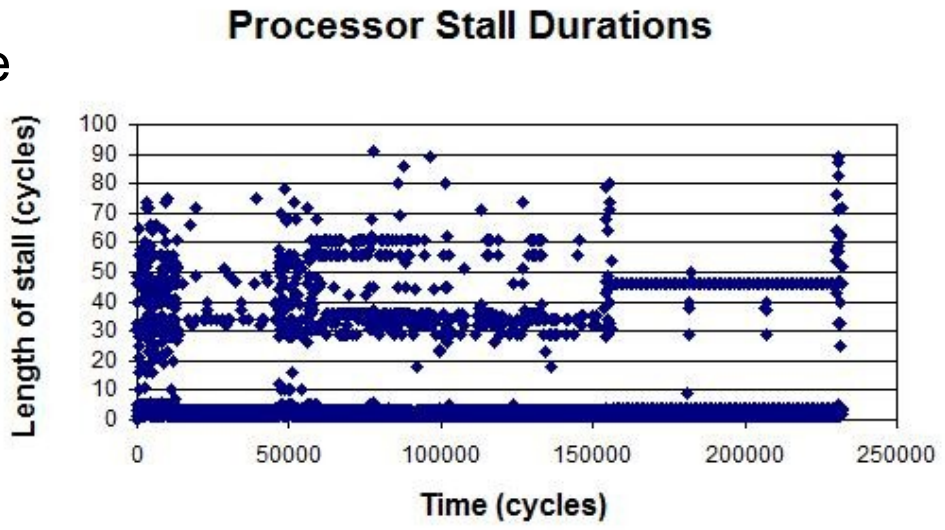
Break-even stall duration for profitable switching

- 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]

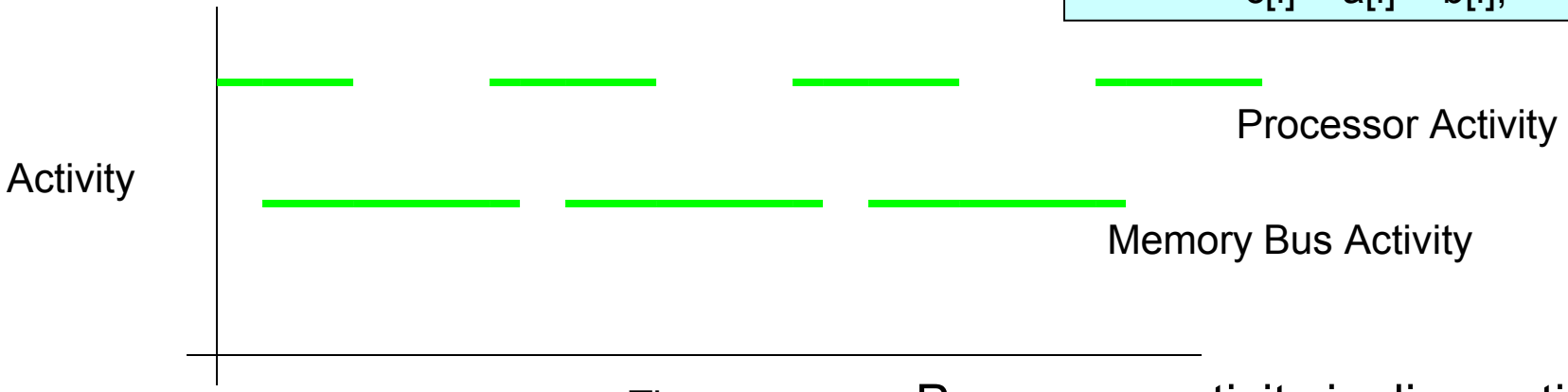
Optimization for exploiting processor-memory interface: Problem Definition (2)

- 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];
```

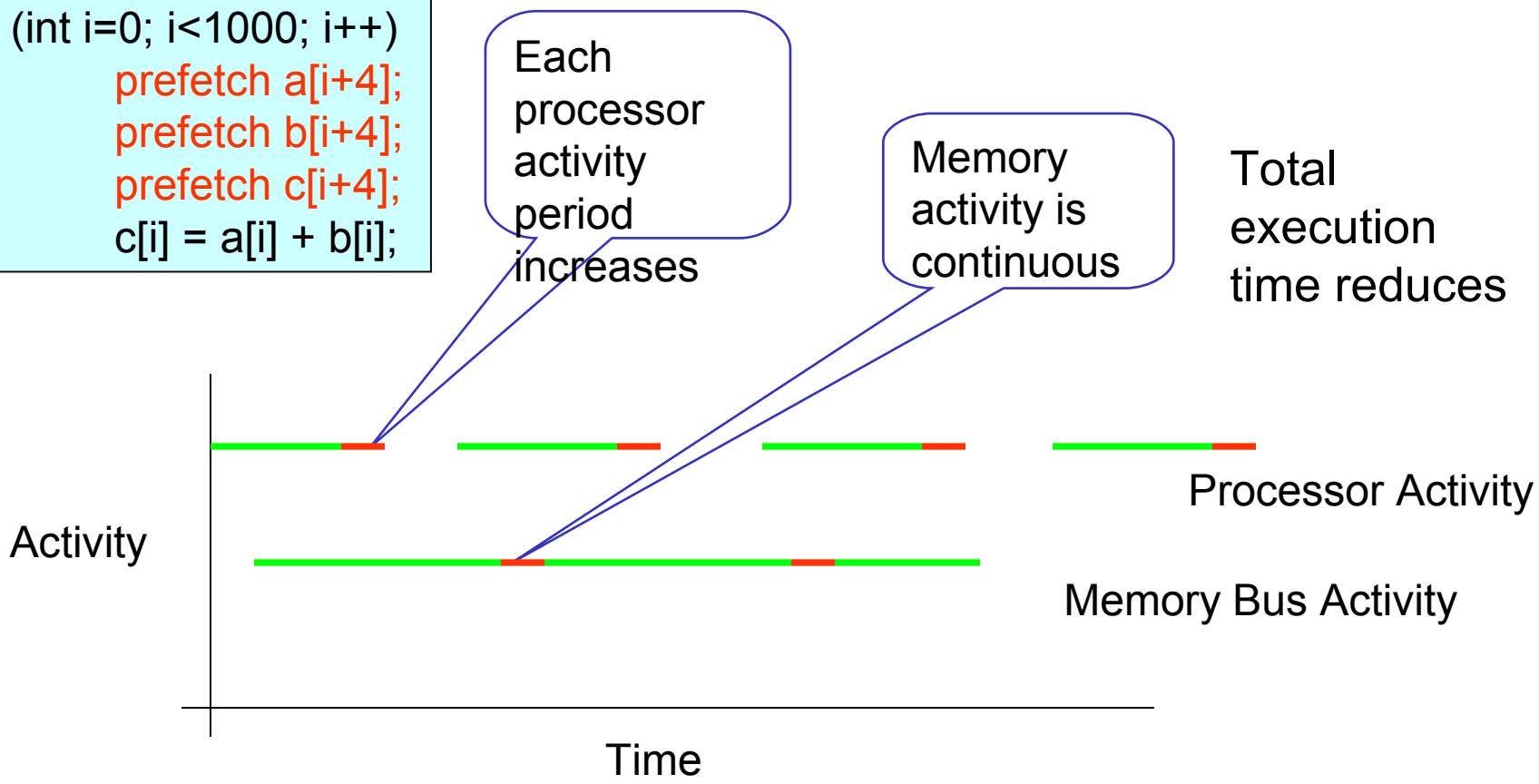


Processor activity is dis-continuous

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];
```

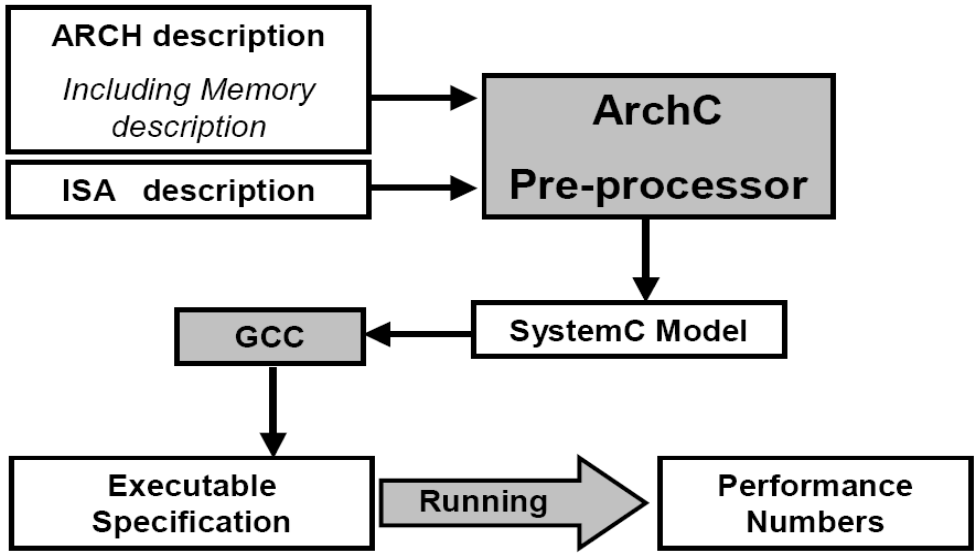


Processor activity is dis-continuous

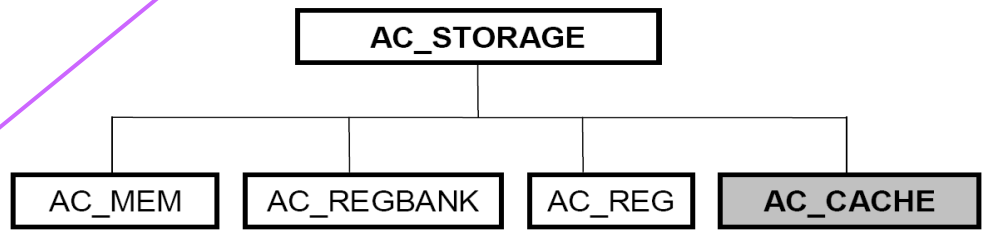
Memory activity is continuous

Memory hierarchy description languages: ArchC

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

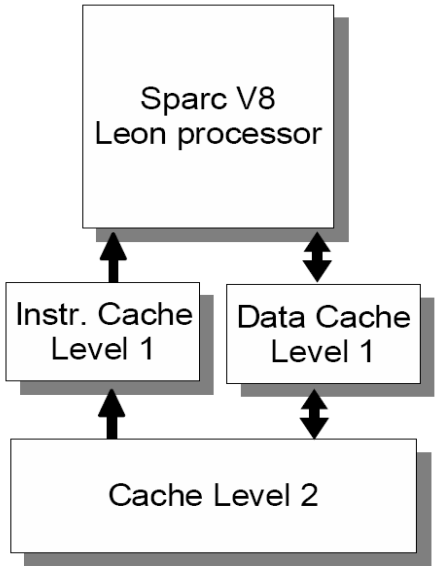


Storage class structure



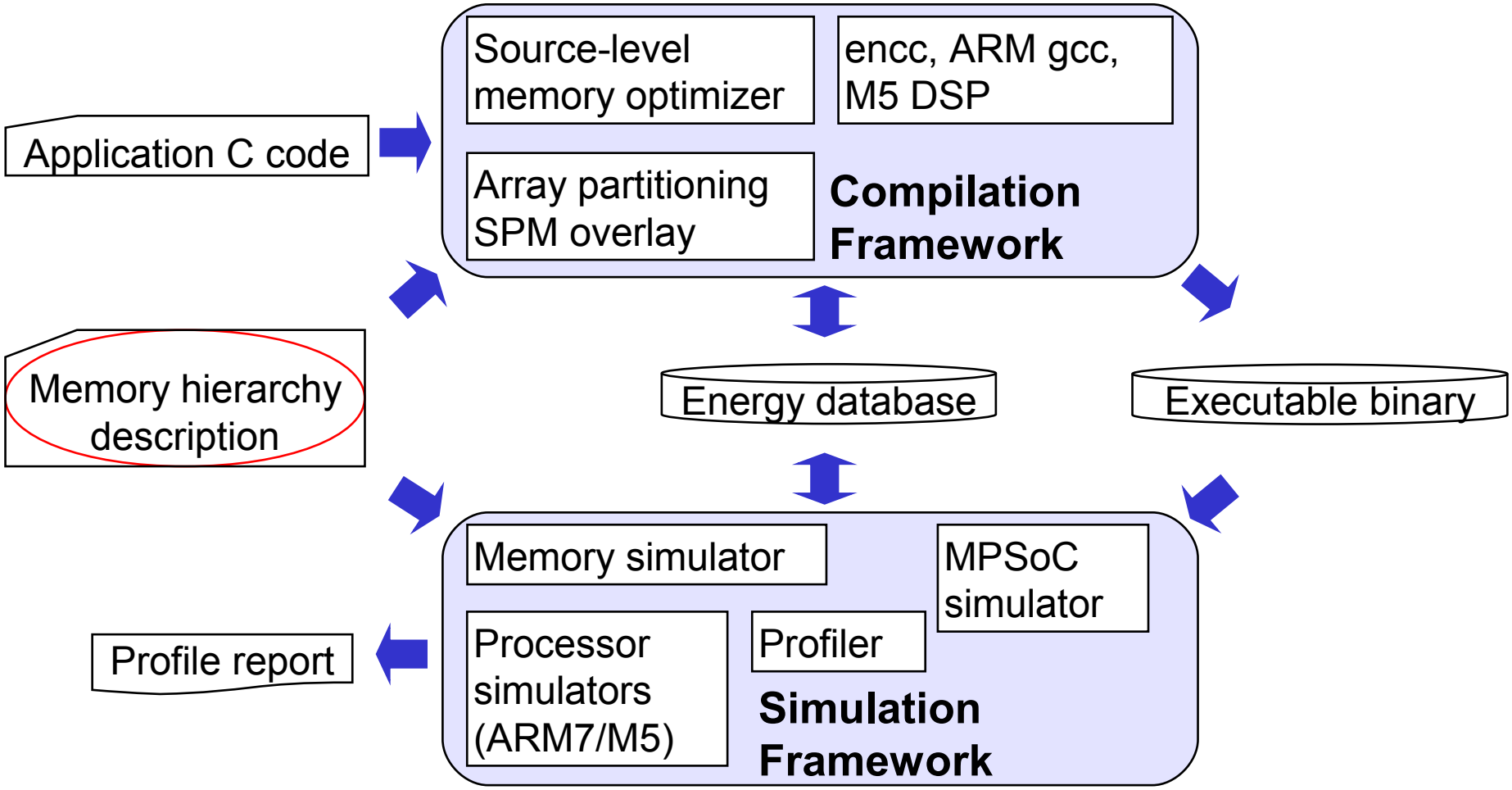
[P. Viana, E. Barros, S. Rigo, R. Azevedo, G. Araújo: Exploring Memory Hierarchy with ArchC, *15th Symposium on Computer Architecture and High Performance Computing*, 2003, pp. 2 – 9]

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

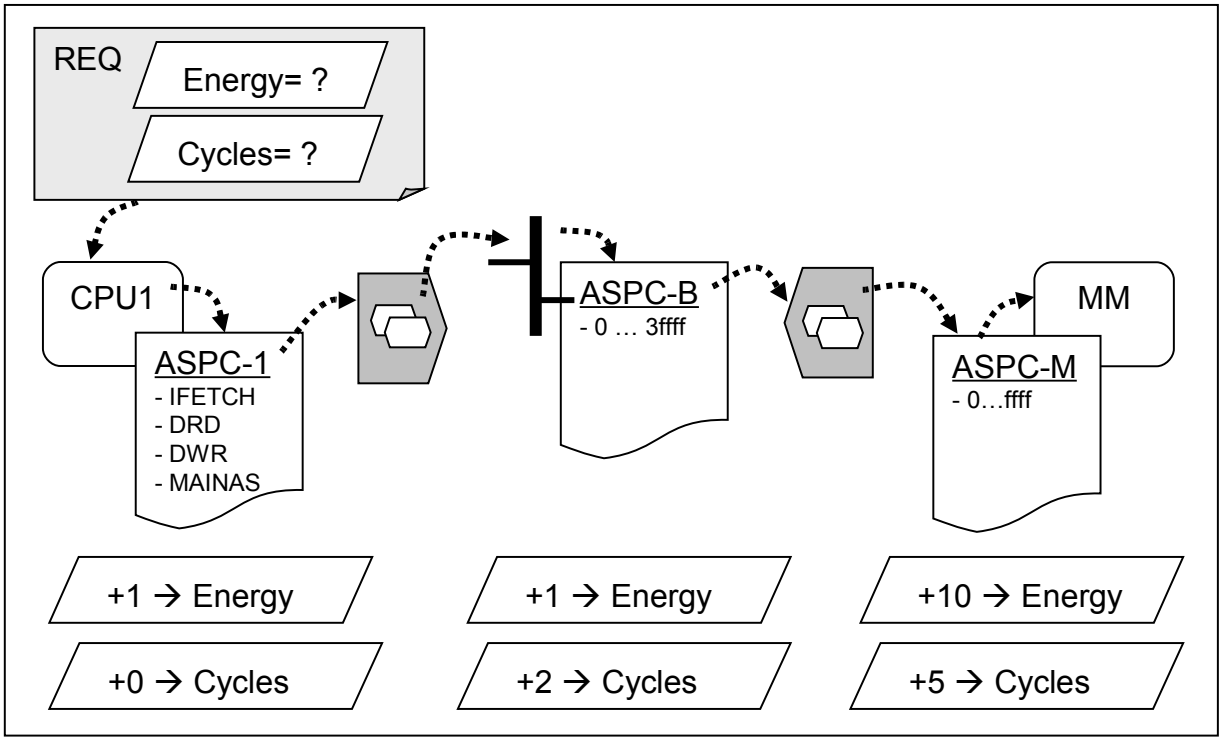


[M. Verma, L. Wehmeyer, R. Pyka, P. Marwedel, L. Benini: Compilation and Simulation Tool Chain for Memory Aware Energy Optimizations, *Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS VI)*, 2006].

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

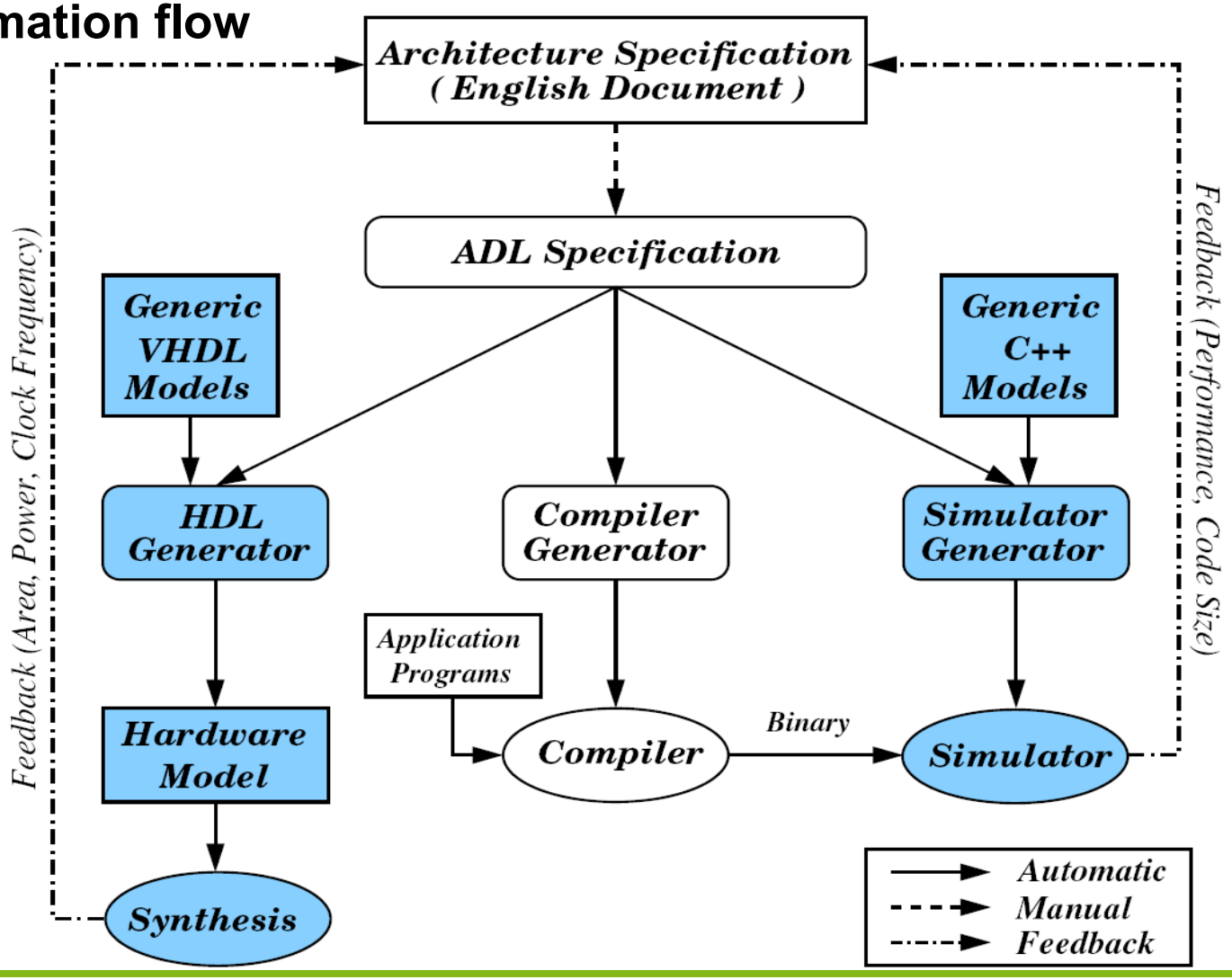
- API query to model simplifies integration into compiler
- External XML representation



[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

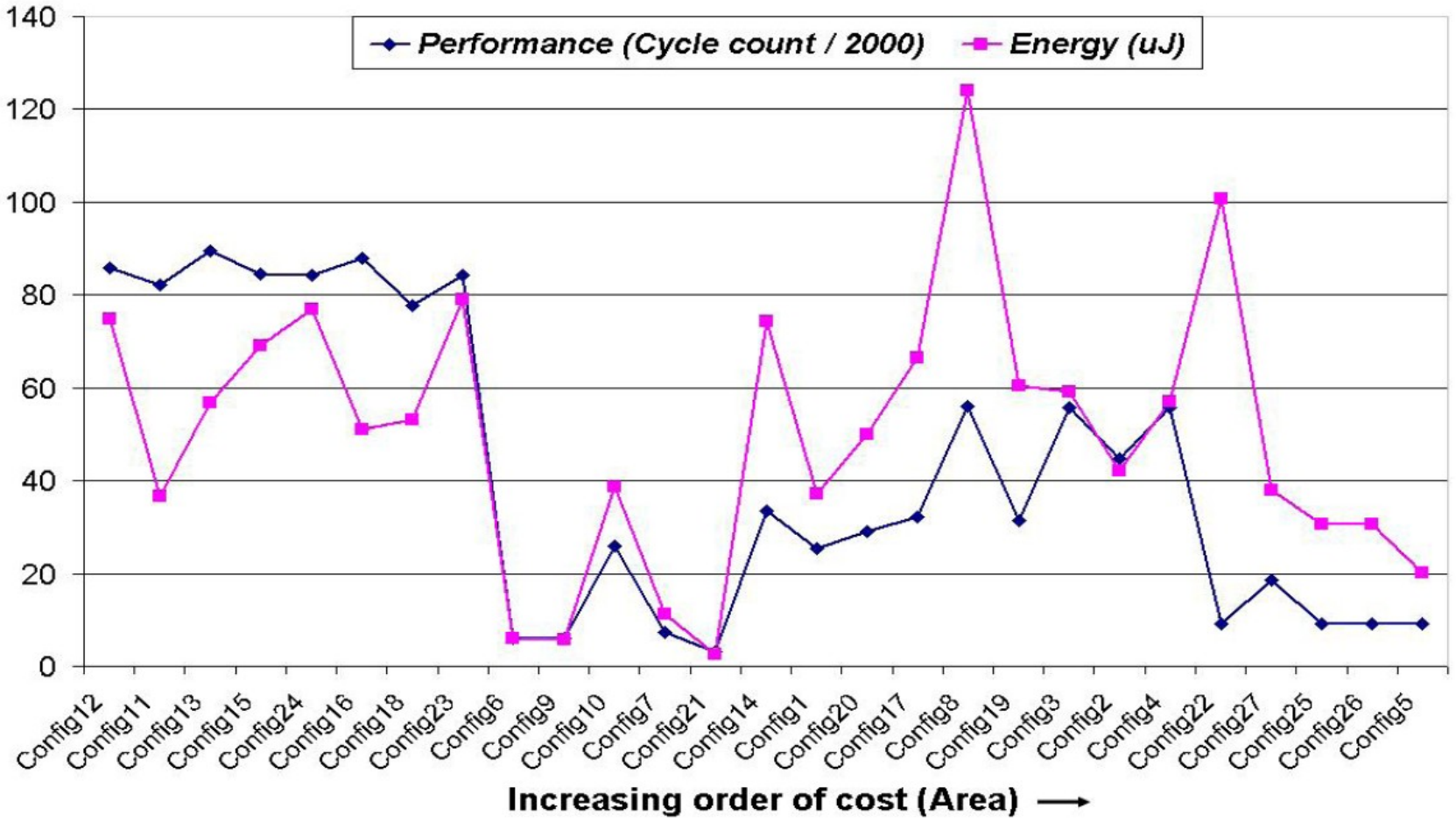
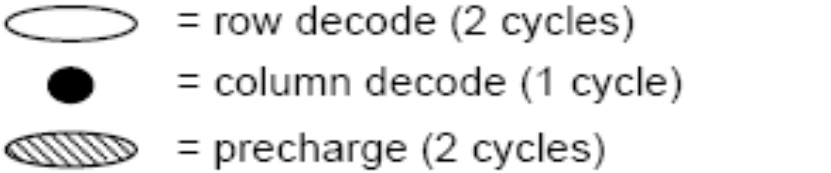


Figure 4.11: Memory exploration results for GSR

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

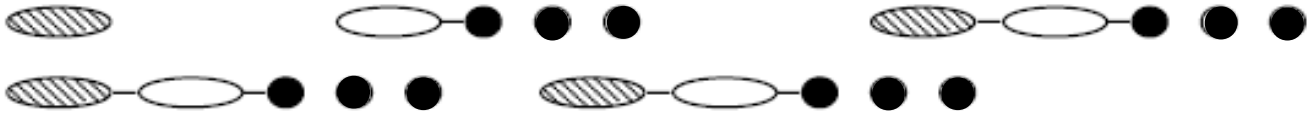
```
for(i=0; i<9;i+=3){
a=a+x[i]+x[i+1]+x[i+2]+
      y[i]+y[i+1]+y[i+2];
b=b+z[i]+z[i+1]+z[i+2]+
      u[i]+u[i+1]+u[i+2];}
```

(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

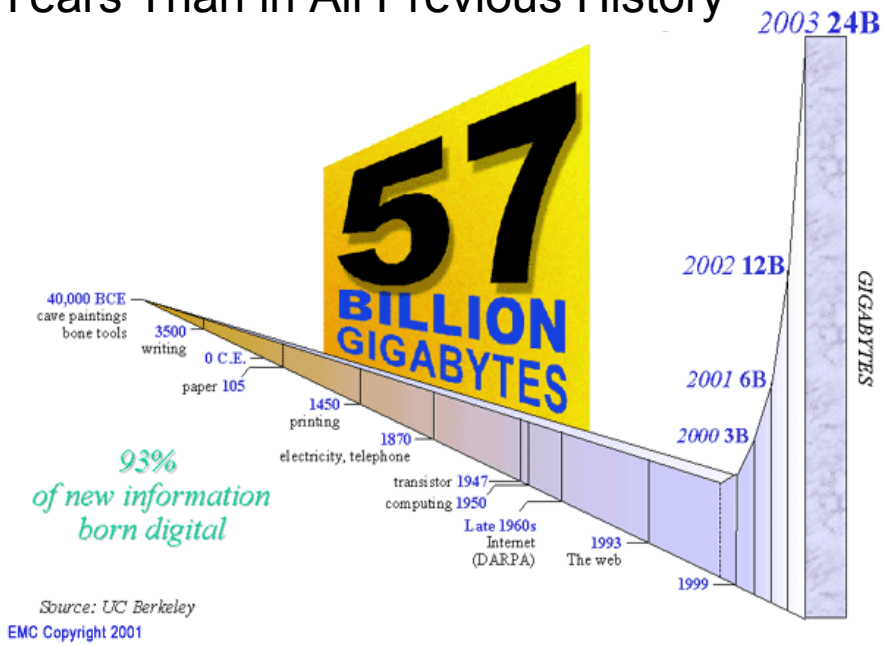
(f) Optimized code with accurate timing

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

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

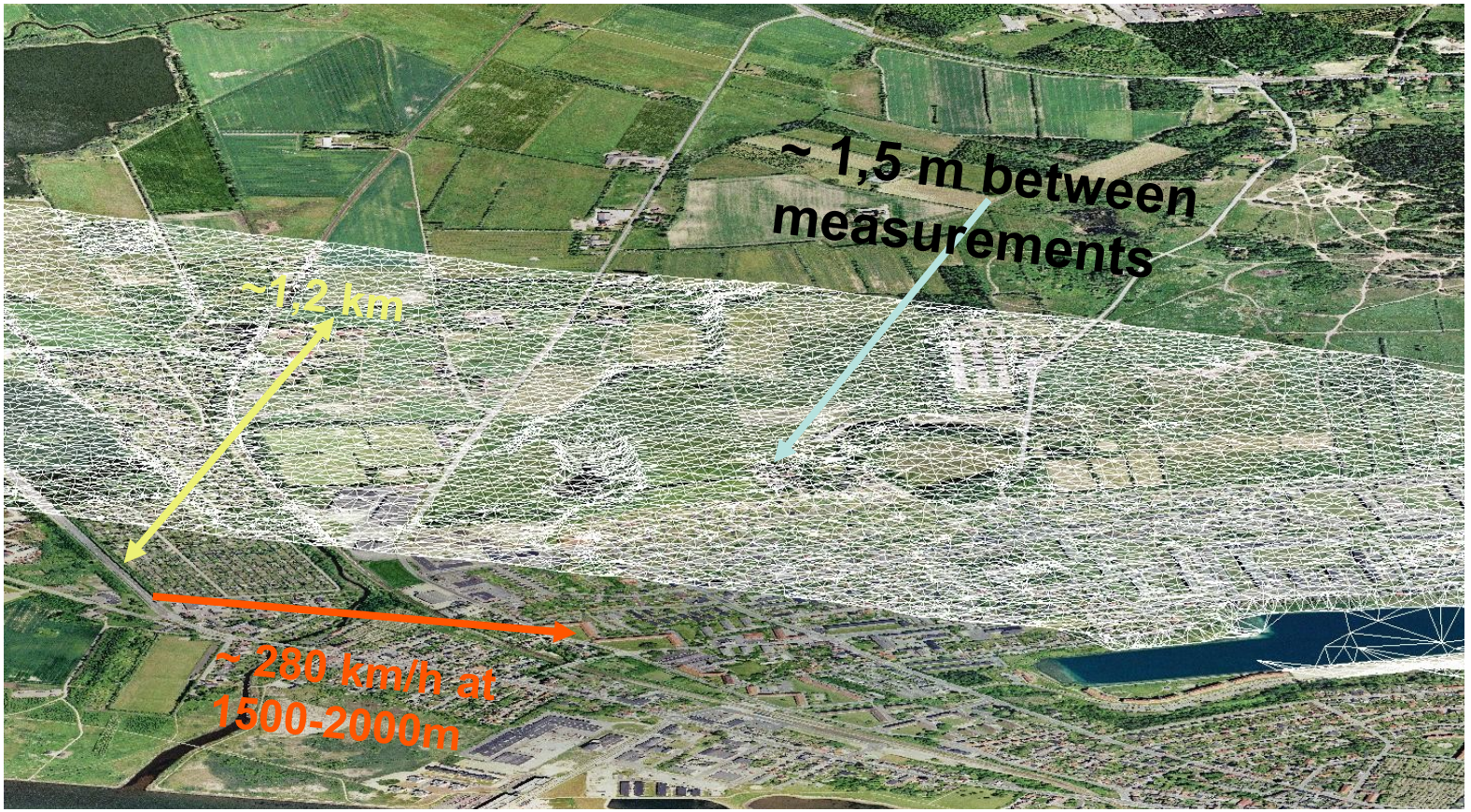
WEB: Web crawl of 200M pages and 2000M links, Akamai stores 7 billion clicks per day

Geography: NASA satellites generate 1.2TB per day

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

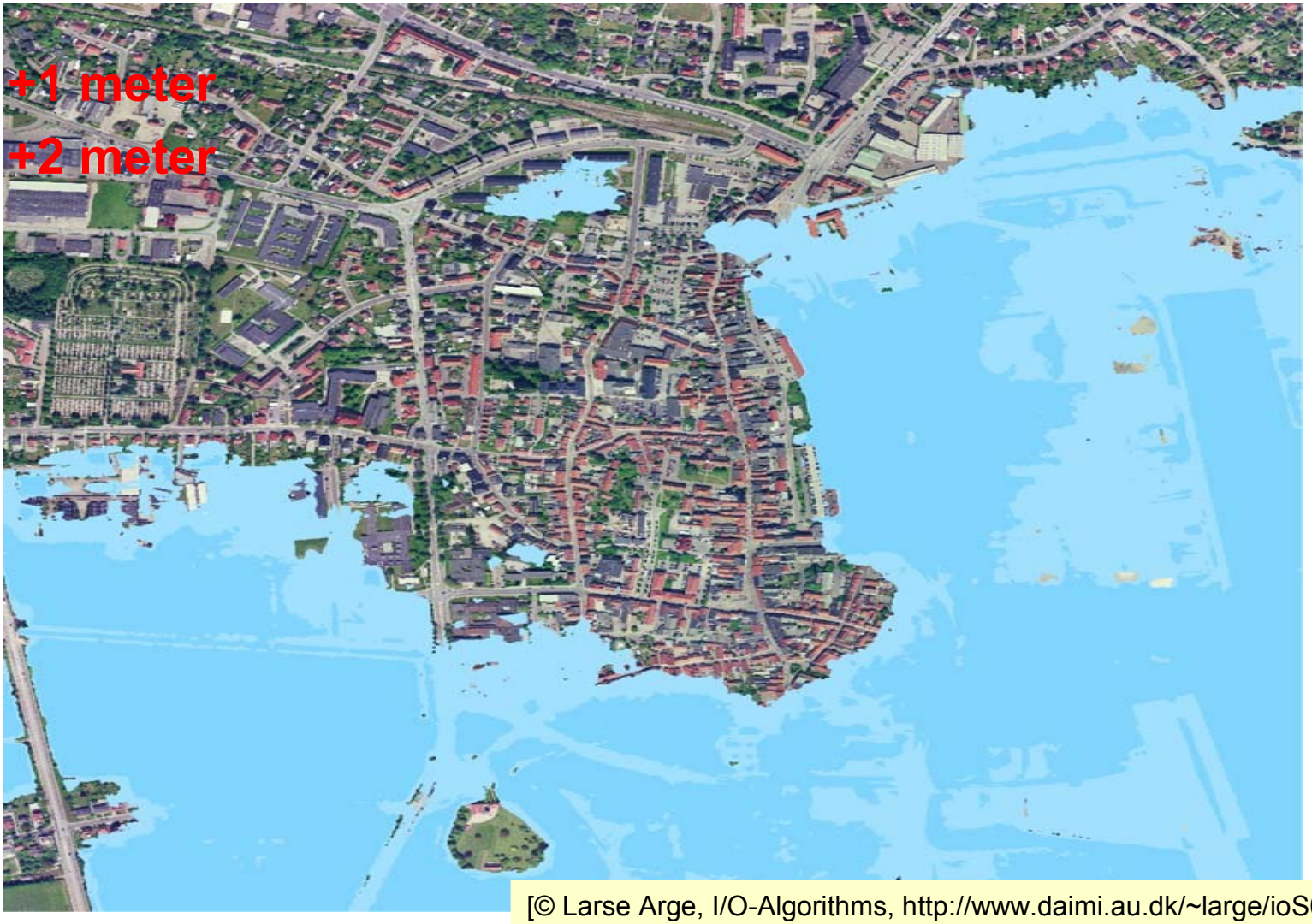
Example: LIDAR Terrain Data

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/]

technische universität
dortmund

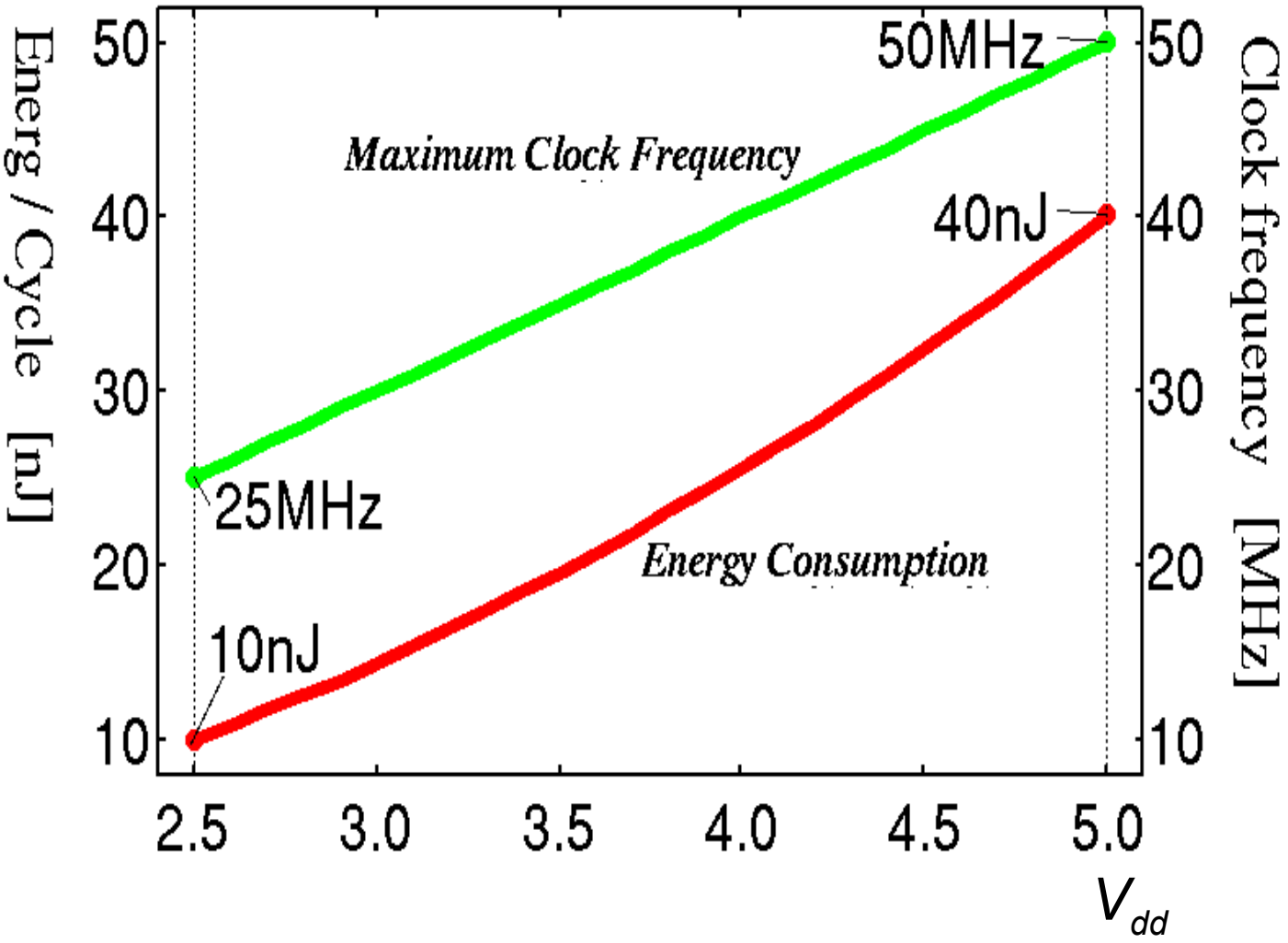
fakultät für informatik
informatik 12

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})

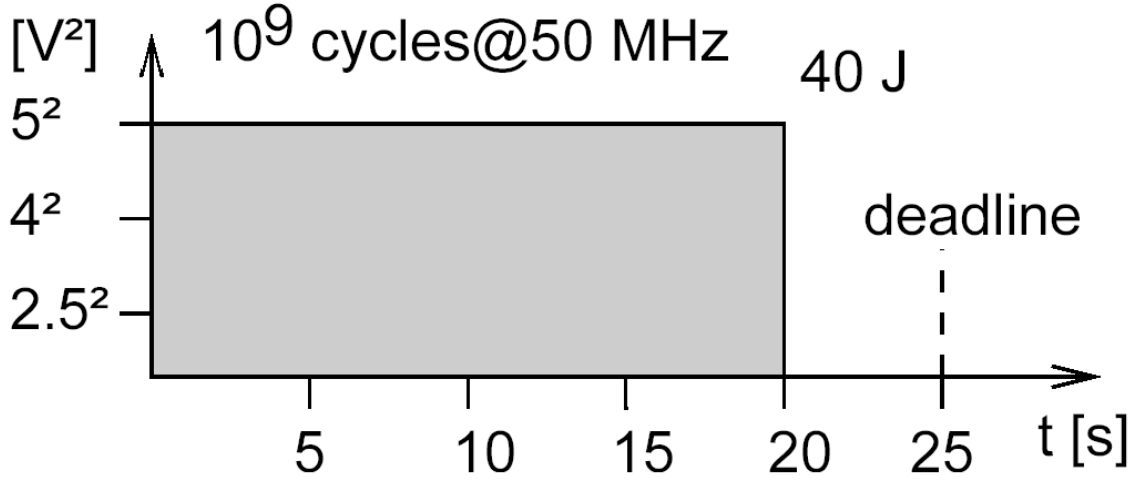
☞ Decreasing V_{dd} reduces P quadratically, while the run-time of algorithms is only linearly increased (ignoring the effects of the memory system).

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)

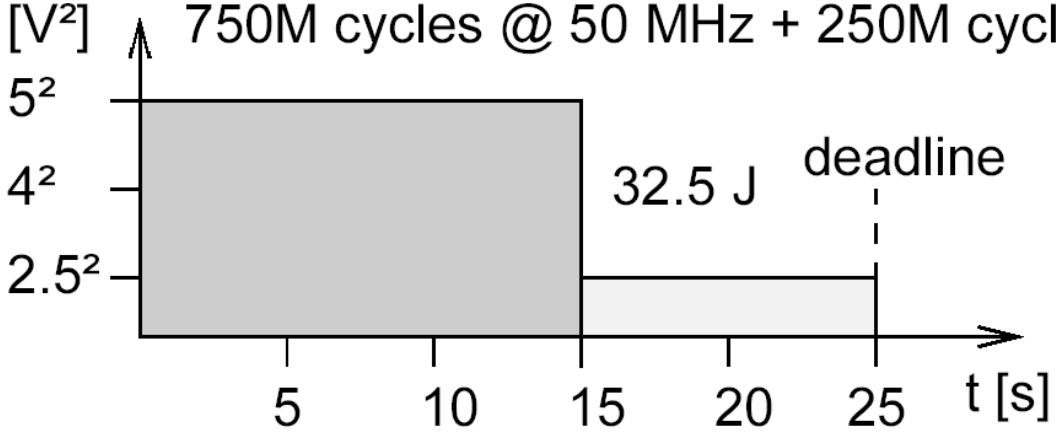


$E_a = 10^9 \times 40 \times 10^{-9}$
$= 40$ [J]

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

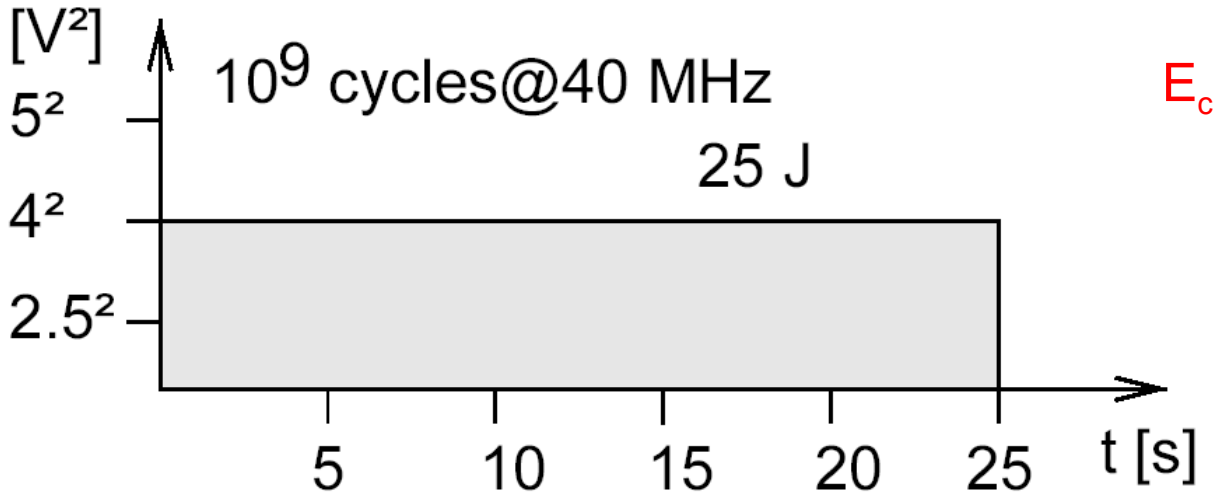


E_b= 750 10^6 x 40 10^{-9} +
250 10^6 x 10 10^{-9}
= 32.5 [J]

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

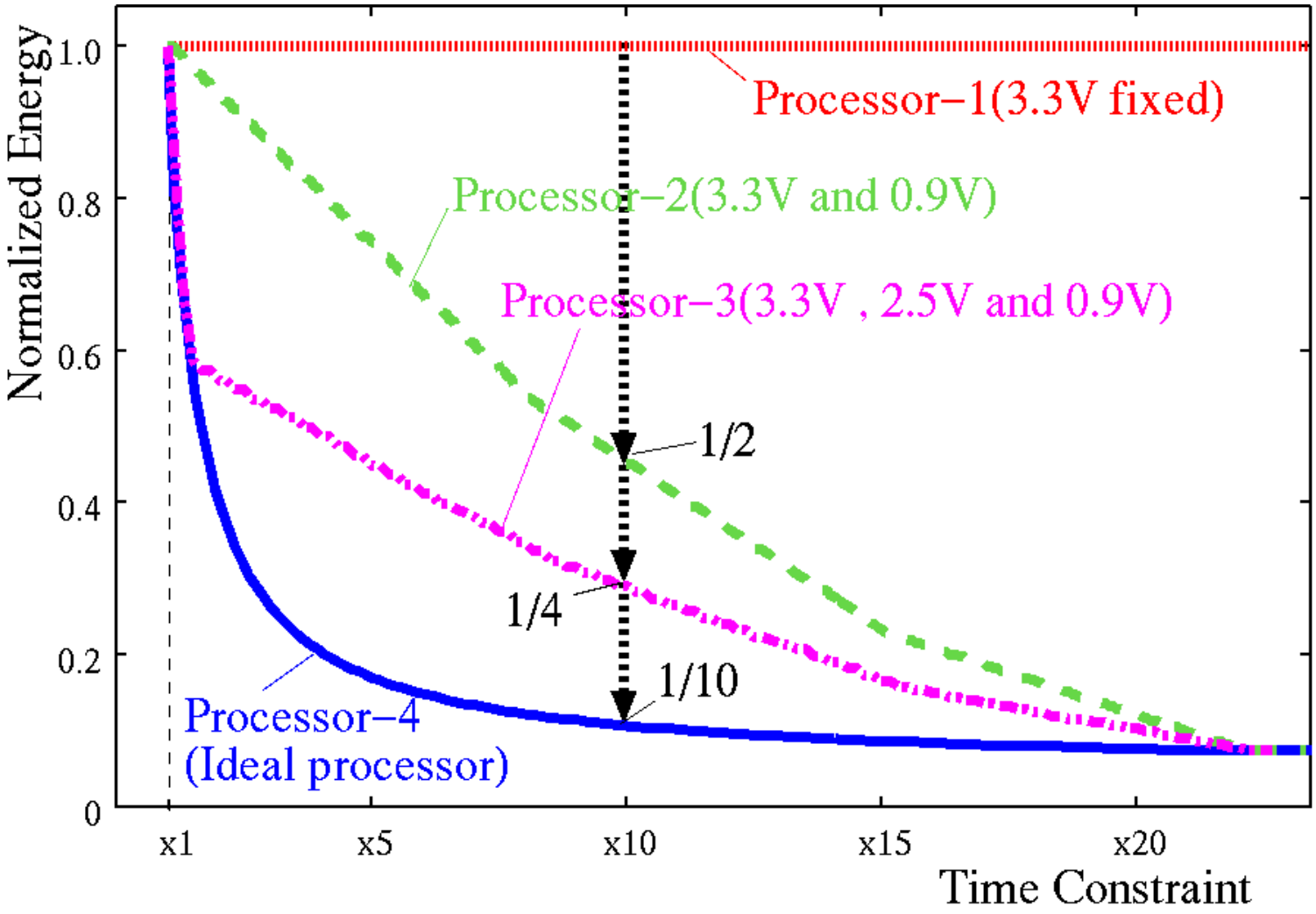
and $$\sum_{i=1}^{L} \sum_{j=1}^{N} \frac{x_{i,j}}{F_i} \leq T$$

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

- Retargetability
- 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)