



Internet of Things

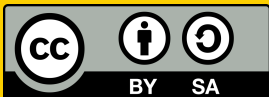
Lecture 15

Energy 2

Energy optimization

with material by Peter Marwedel

Michael Engel



Lecture slides licensed under
CC-by-SA 4.0 (unless noted otherwise)



Energy optimization

Challenge:

How can we execute the required functionality of an IoT device and reduce its energy consumption?

Approaches:

- Energy models → compiler optimizations
- Exploit energy properties of the memory hierarchy
- Dynamic Voltage-Frequency Scaling (DVFS)



Energy models → compiler optimizations

- Is optimization for low energy the same as for high performance?
- **No!**
 - High performance: available memory bandwidth fully used
 - Low energy consumption: memories are in 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,r10
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

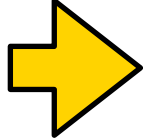


Energy models → compiler optimizations

Some compiler optimizations (usually done to improve execution performance) also reduce energy consumption:

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

```
for (i=0; i<=10; i++)  
    c = 2 * a[i] + a[i-1];
```



```
R2 = a[0];  
for (i=1; i<=10; i++) {  
    R1 = a[i];  
    c  = 2 * R1 + R2;  
    R2 = R1;  
}
```

Exploitation of the memory hierarchy:

The value $a[i]$, also used in subsequent loop iteration (as $a[i-1]$), is retained (**cached**) by the compiler in a processor register

Energy-aware compiler optimizations

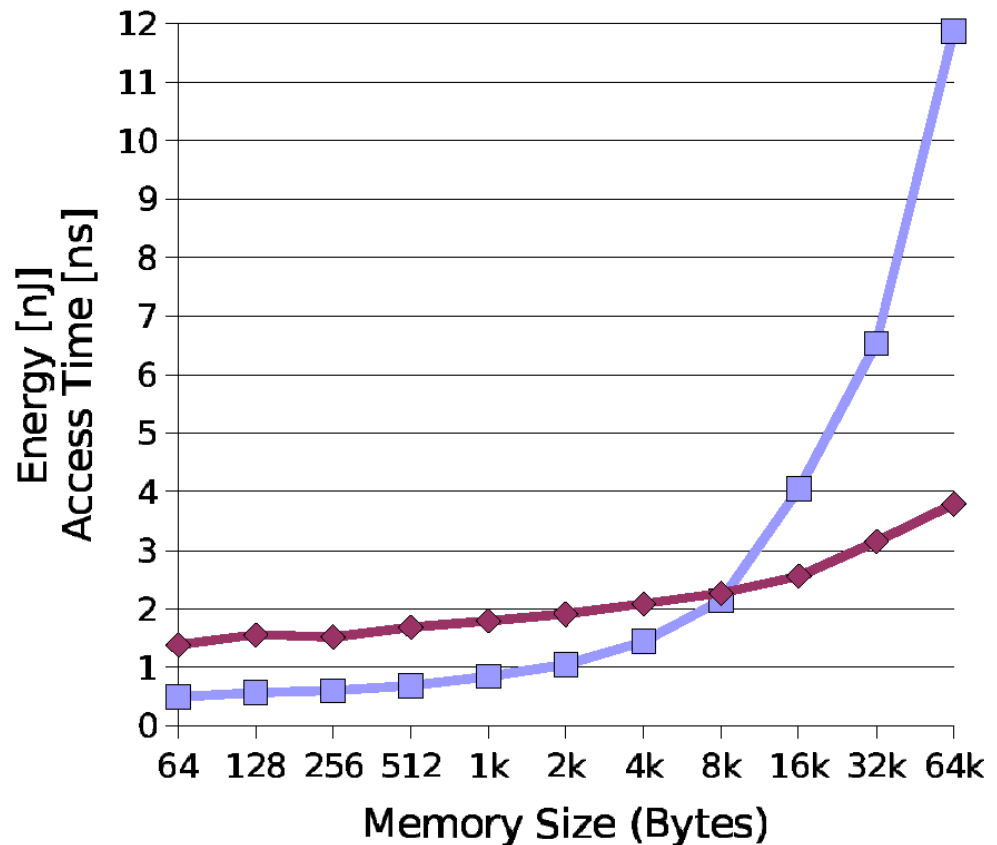
- Energy-aware scheduling
 - the order of the instructions can be changed as long as the meaning (semantics) does not change
 - Goal: reduction of the number of signal transitions
 - Popular (can be done as a post-pass optimization with no change to the compiler).
- Energy-aware instruction selection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



Energy and the memory hierarchy

Three key problems for future memory systems:

- (Average) Speed
 - **Energy/Power**
 - Predictability/WCET
-
- Speed and energy consumption increases superlinear with increasing memory sizes

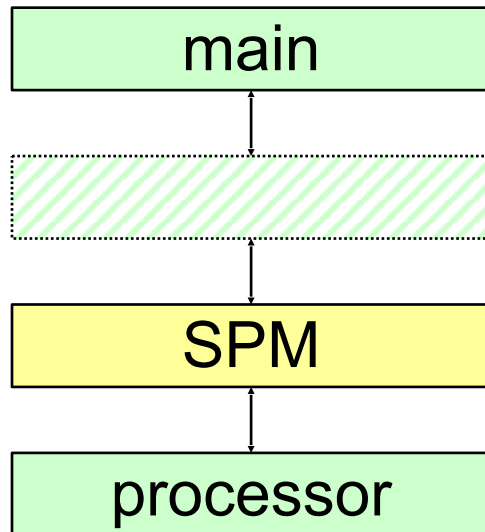


(note the log scale on the x axis!)

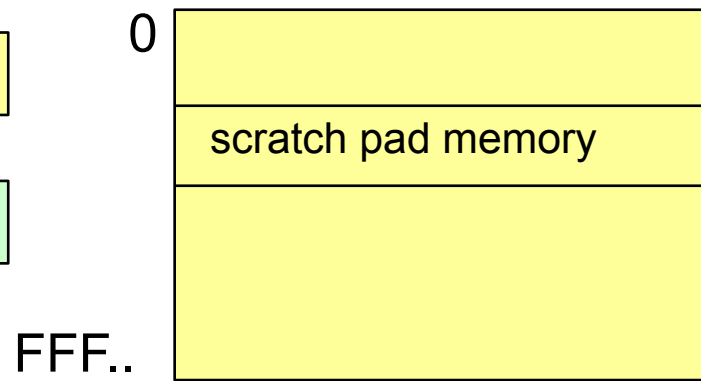
Energy optimization in the memory hierarchy

- Scratch pad memories (SPM) [1] are small, fast, energy-efficient memories closely coupled to the CPU (ARM term: tightly coupled memory TPM)

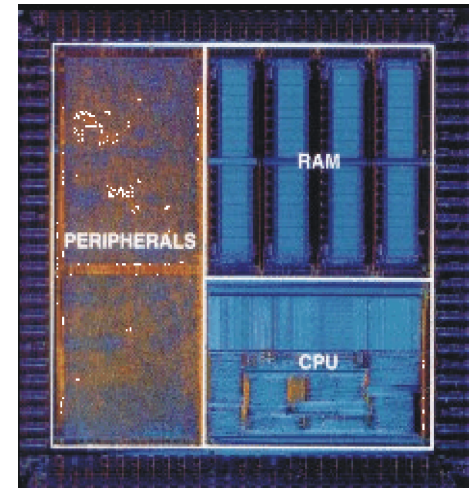
Hierarchy



Address space



Example

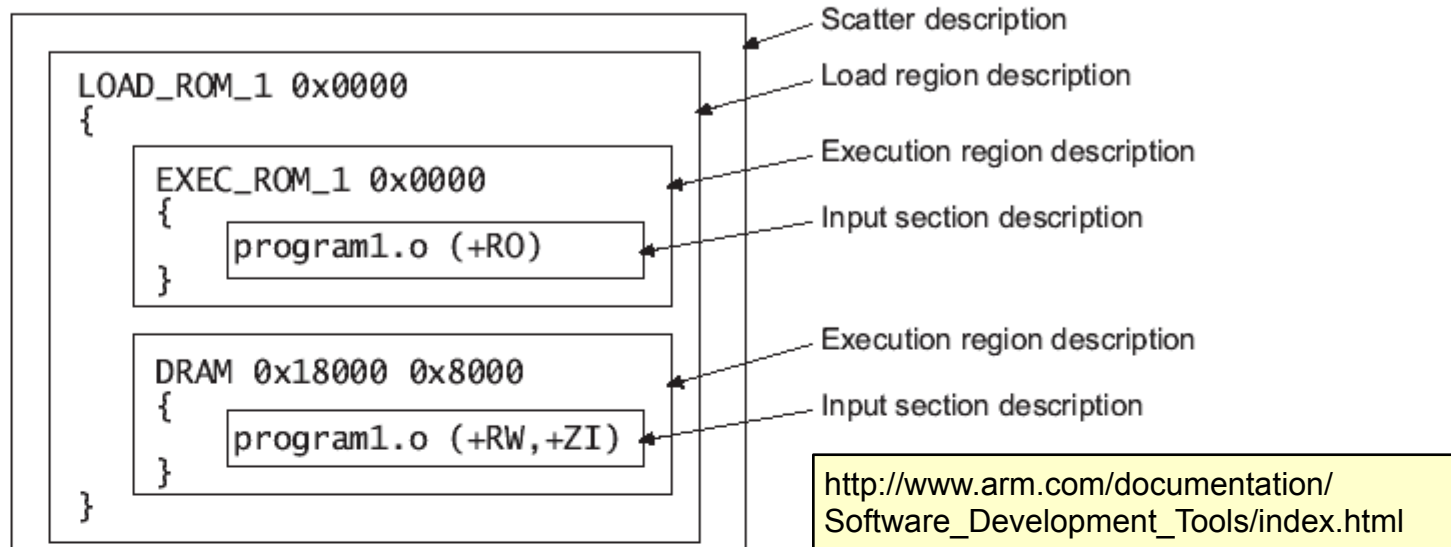


ARM7TDMI
cores, well-known
for low power
consumption

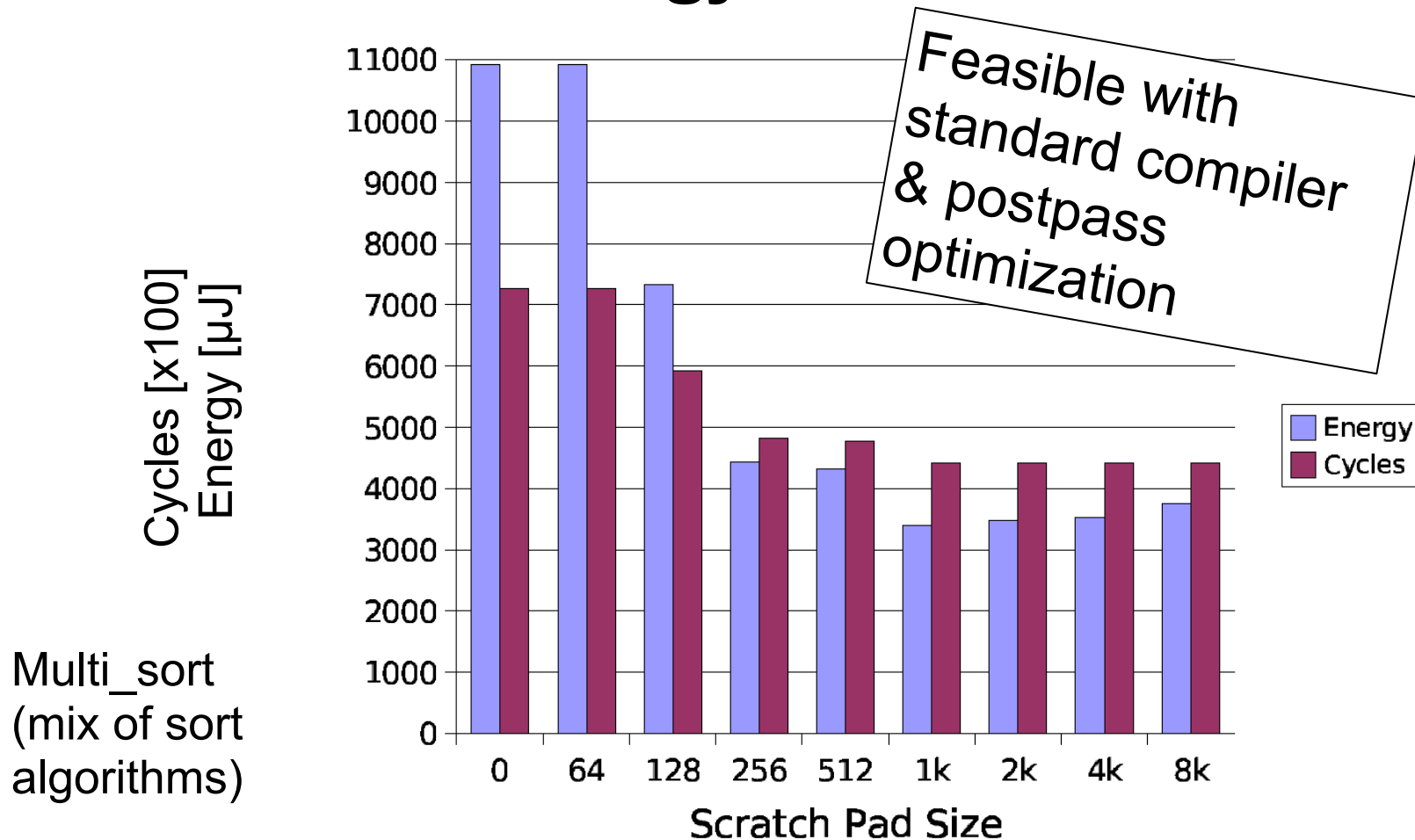
Support in regular compilers

- 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
```
- Input scatter loading file to linker for allocating section to specific address range



Reduction in energy and runtime



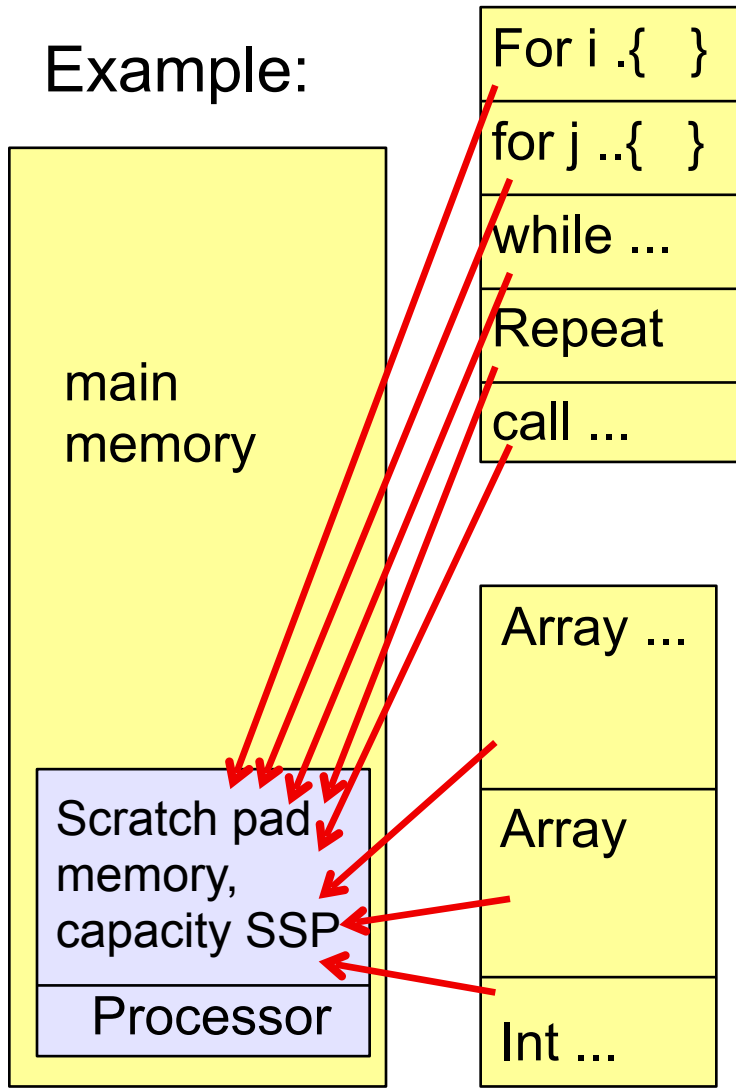
Measured processor / external memory energy +
CACTI values for SPM (combined model)

Numbers will change with technology,
algorithms remain unchanged



Migration of code and data – energy aware compiler

Example:



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

Migration of code and data – ILP optimization

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
(= $e(var_k) \cdot n(var_k)$)
- $x(var_k)$ = decision variable, =1 if variable k is migrated to SPM,
= 0 otherwise
- K = set of variables; similar for functions I

Integer programming [5] formulation:

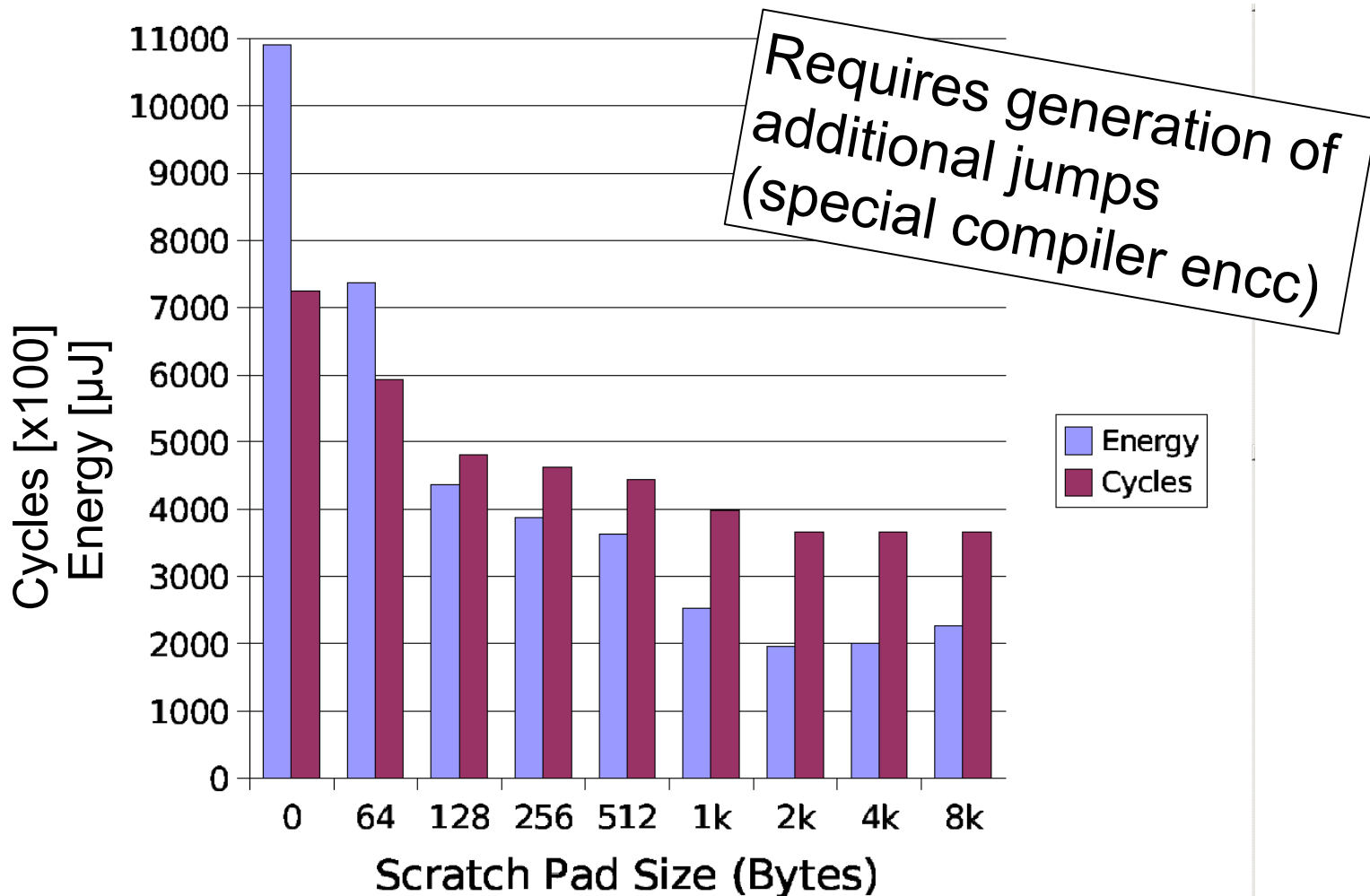
Maximize $\sum_{k \in K} x(var_k) E(var_k) + \sum_{i \in I} x(F_i) E(F_i)$

Subject to the constraint

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

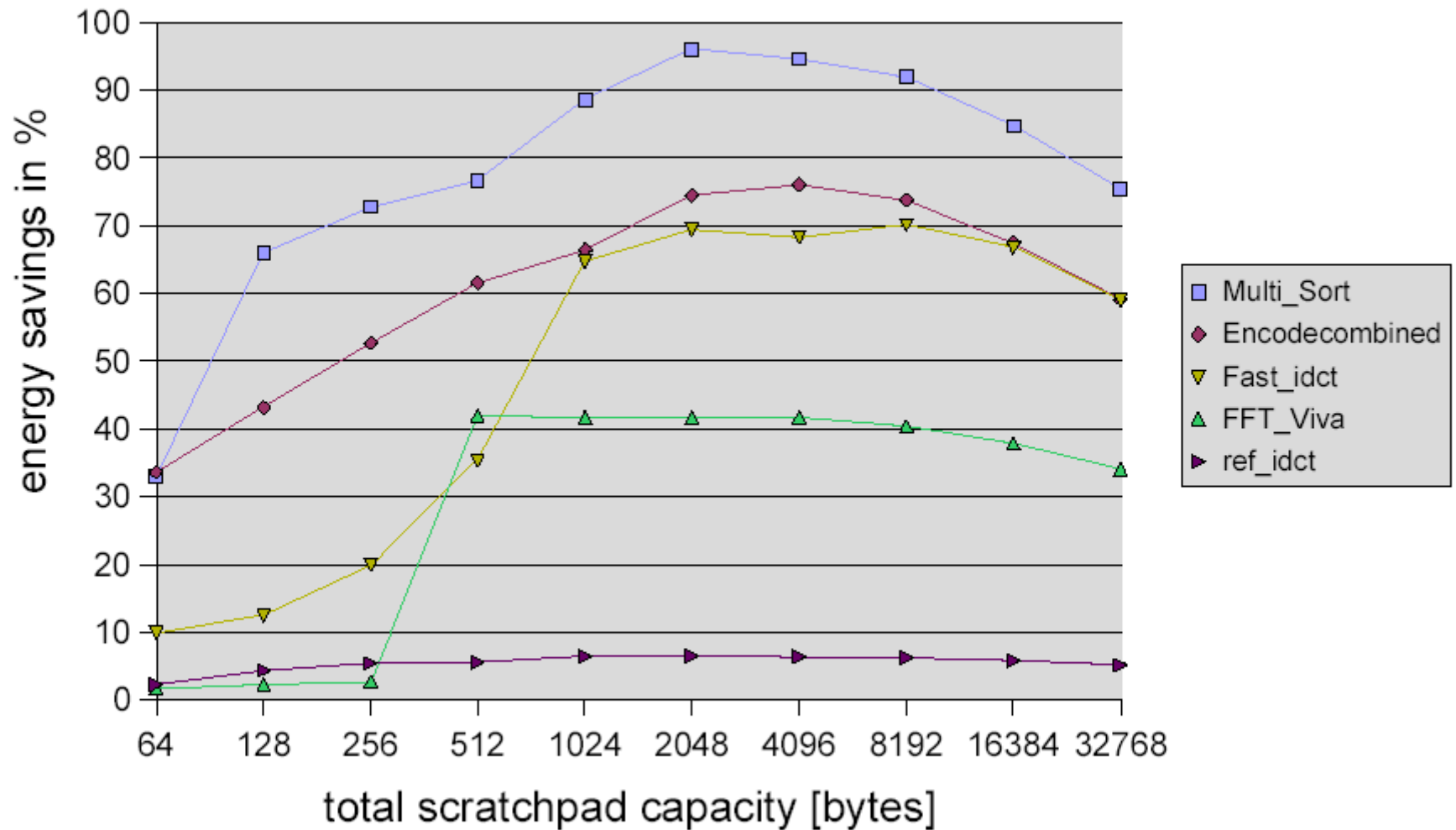


Code and stack allocation in SPM



Optimizations using the energy-aware C compiler (encc) from TU Dortmund [2]

Savings for memory system alone



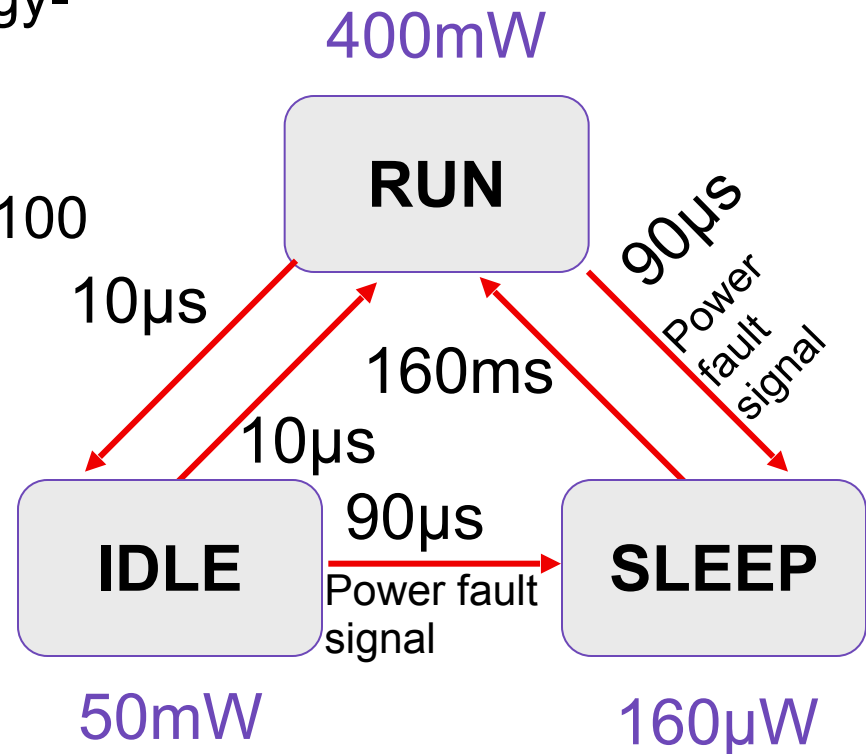
Dynamic processor power management

Most embedded processors can be switched between different modes of operation

- Switching to and from energy-saving idle or sleep modes implies overhead

Example: DEC StrongARM SA1100

- **RUN**: operational
- **IDLE**: a software routine may stop the CPU when not in use, while monitoring interrupts
- **SLEEP**: Shutdown of on-chip activity



Dynamic Voltage-Frequency Scaling (DVFS)

Power consumption of CMOS circuits (ignoring leakage):

$$P = \alpha C_L V_{dd}^2 f \text{ with}$$

α : switching activity

C_L : load capacitance

V_{dd} : supply voltage

f : clock frequency

Delay for CMOS circuits:

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

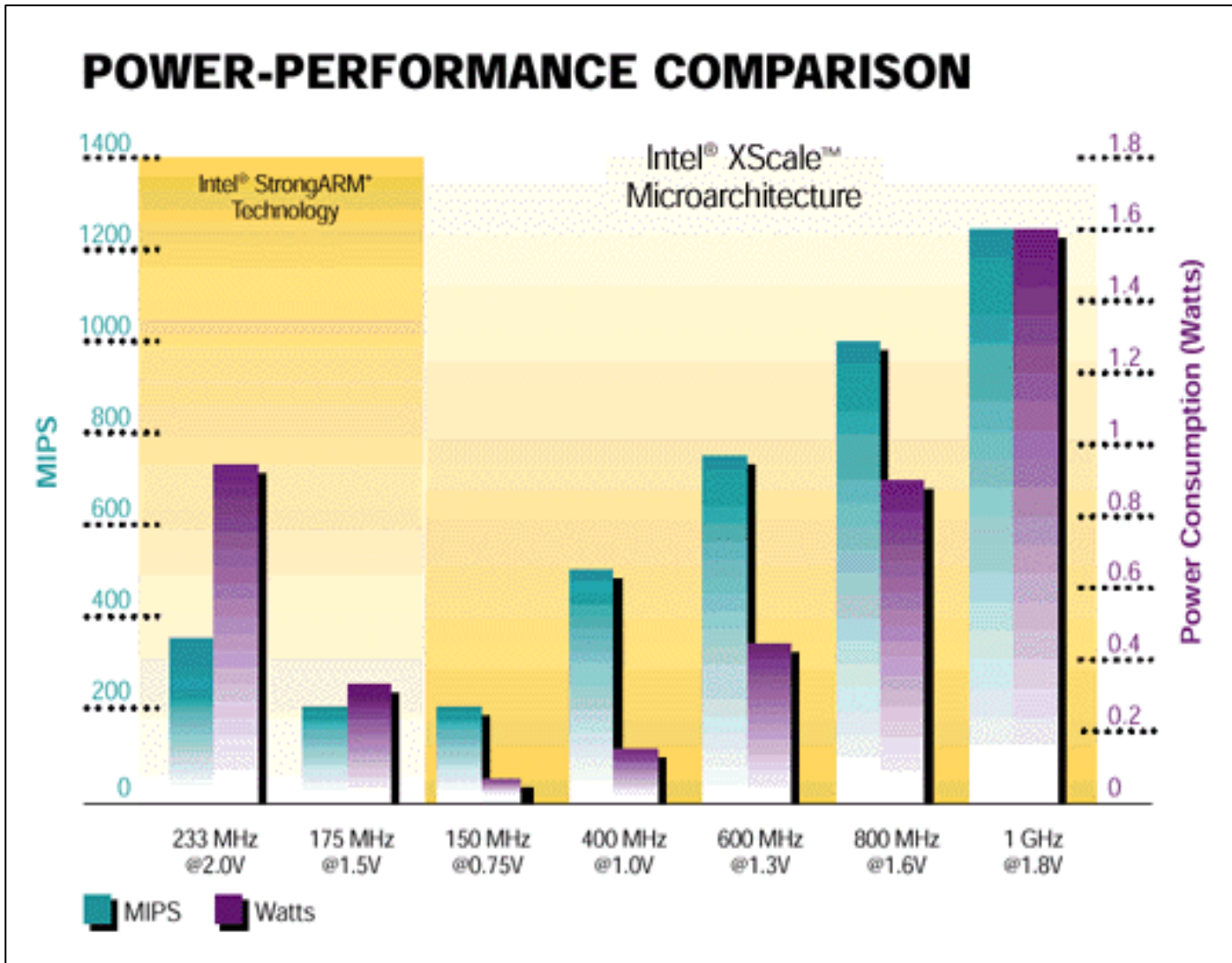
V_t : threshold voltage

($V_t < V_{dd}$)

➡ Decreasing V_{dd} reduces P quadratically,
while the run-time of algorithms is only linearly increased



Example: Intel XScale



OS has to schedule distribution of the energy budget

Low voltage parallel more efficient than high voltage sequential

Basic equations

Power:

$$P \sim V_{DD}^2,$$

Maximum clock frequency:

$$f \sim V_{DD},$$

Energy to run a program:

$$E = P \times t, \text{ with: } t = \text{runtime}$$

Time to run a program:

$$t \sim 1/f$$

Changes due to parallel processing, with α operations per clock:

Clock frequency reduced to:

$$f' = f / \alpha,$$

Voltage can be reduced to:

$$V_{DD}' = V_{DD} / \alpha,$$

Power for parallel processing:

$$P^\circ = P / \alpha^2 \text{ per operation,}$$

Power for α operations per clock:

$$P' = \alpha \times P^\circ = P / \alpha,$$

Time to run a program is still:

$$t' = t,$$

Energy required to run program:

$$E' = P' \times t = E / \alpha$$

*Rough
approximations!*

☞ Argument in favour of voltage scaling,
VLIW processors (e.g. in DSPs) and multi-cores

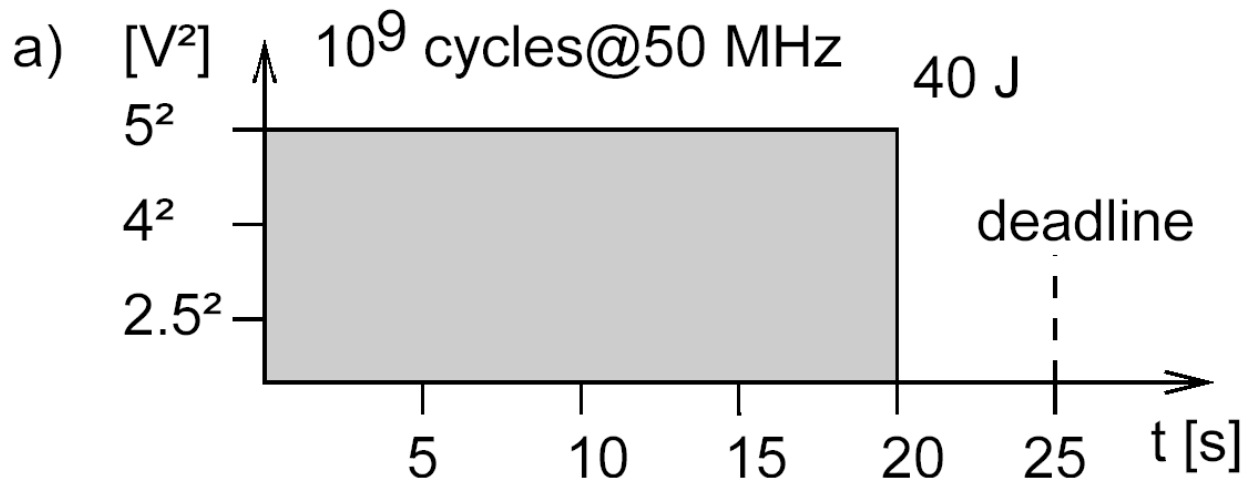


DVFS example: Processor with 3 voltages

Case a: execute the 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



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

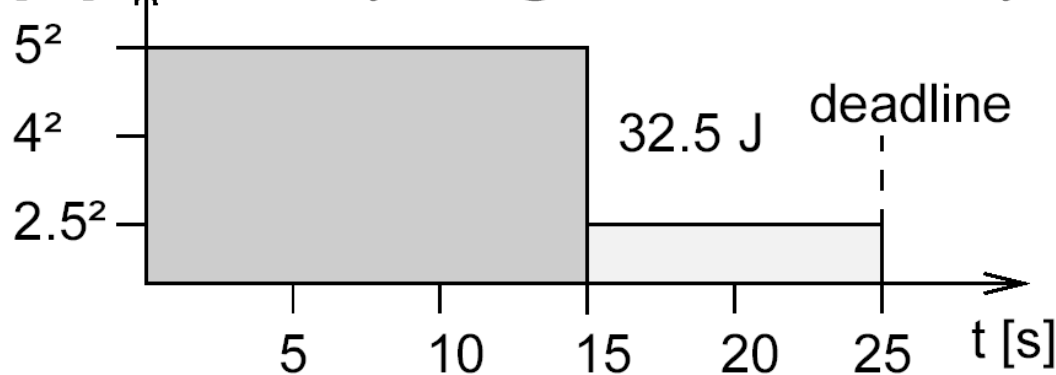


DVFS example: Processor with 3 voltages

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) $[V^2]$ 750M cycles @ 50 MHz + 250M cycles @ 25



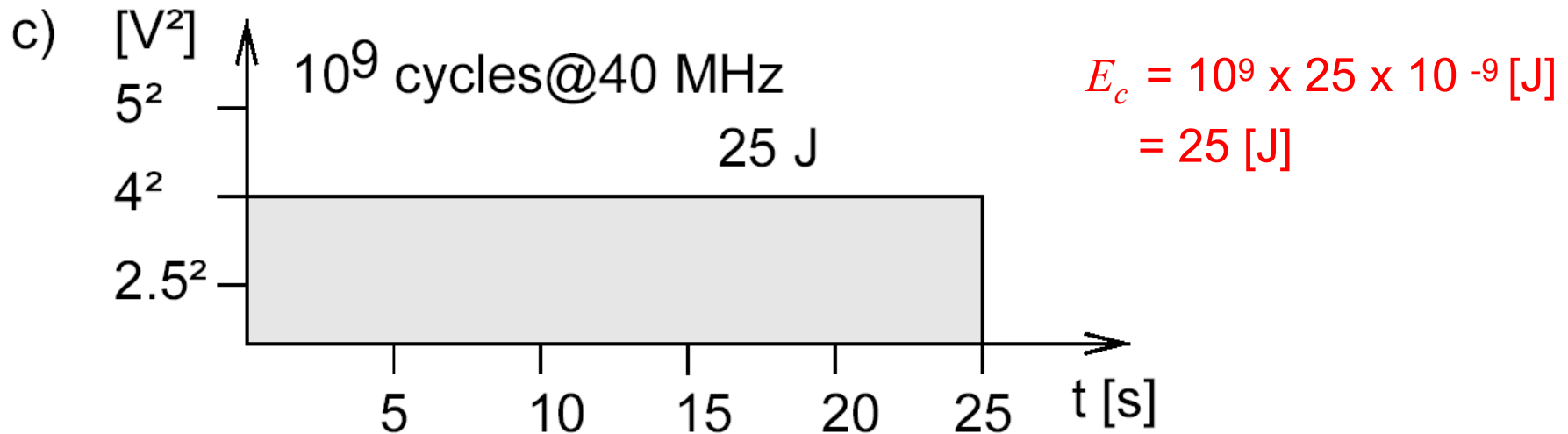
$$\begin{aligned}
 E_b &= 750 \cdot 10^6 \times 40 \cdot 10^{-9} + \\
 &\quad 250 \cdot 10^6 \times 10 \cdot 10^{-9} \text{ [J]} \\
 &= 32.5 \text{ [J]}
 \end{aligned}$$



DVFS example: Processor with 3 voltages

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



Conclusion

- Many different approaches to reduce the energy consumption in embedded (and, in turn, IoT) applications exist
 - We showed two examples, assignment of code sections to SPM and dynamic voltage-frequency scaling (DVFS)
- **Scratch-pad memory (SPM)**
 - Assign often used code and/or data sections to SPM
 - Can be understood as manual cache management
 - SPMs are more energy efficient than caches since caches require comparators for tags
- **Dynamic voltage-frequency scaling (DVFS)**
 - Adapt the voltage and frequency (which go together) to clock the processor as fast as required for the given task
 - Use idle or sleep modes otherwise

Most approaches involve complex optimizations, e.g. using ILP



References

- [1] R. Banakar, S. Steinke, Bo-Sik Lee, M. Balakrishnan and P. Marwedel, *Scratchpad memory: a design alternative for cache on-chip memory in embedded systems*, Proceedings of CODES, IEEE 2002
- [2] M. Verma, L. Wehmeyer and P. Marwecler, *Dynamic overlay of scratchpad memory for energy minimization*, International Conference on Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004., 2004, pp. 104-109
- [3] James Montanaro et al., *A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor*, Digital Technical Journal, vol. 9, no. 1. pp. 49–62, 1997
- [4] Etienne Le Sueur and Gernot Heiser, *Dynamic voltage and frequency scaling: the laws of diminishing returns*, In Proceedings of the 2010 international conference on Power aware computing and systems (HotPower'10). USENIX Association, USA
- [5] C.H. Papadimitriou and K. Steiglitz, K, *Combinatorial optimization: algorithms and complexity*, Mineola, NY: Dover, 1998, ISBN 0486402584.

