Memory and compiler optimizations for low-power and -energy

Olivier Zendra
TRIO team
INRIA-Lorraine / LORIA
(Nancy, FRANCE)

olivier.zendra@loria.fr

http://www.loria.fr/~zendra

Low-power and -energy: motivation ?

- Tremendous issues in embedded systems
  - More and more widespread systems
    - Especially autonomous ones
- (Massively) multi-processors systems, grid: also become issues
  - Power supply
  - Thermal dissipation
  - (Energy cost)

Introduction: low-energy, where to act ?

- $P = C.V^2.f$; $E = P_{avg}.Time$
- Hardware design
  - Microelectronics, physics...
- Hardware optimization
  - On-line logic
  - Dedicated circuits
  - Overhead at runtime (Energy and Time)

Introduction: low-energy, where to act ?

- Software optimization (at compile-time, static)
  - Off-line logic
  - No overhead at runtime
  - More resources available (Time, RAM)
  - Much larger context possible
  - Exact runtime behavior harder to catch

Compilation and low-energy

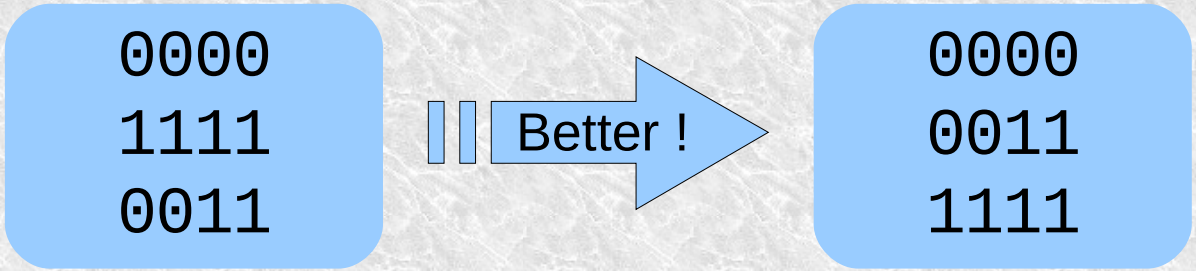
- Energy: relatively new in compilation. Historically, size & speed.
- Optimizations for speed and energy
  - Often related [Lee1997]
  - Not always: moving out of critical path is good for time, but is it for energy ?
- Optimizing for energy != for power density (hot spots)

This talk

- Goal: raise awareness to low-energy
- Survey of techniques and solutions
- Focus on some specific aspects
- Compiler optimizations
- Memory management

1) Transitions and commutations

- Transitions between successive instructions: cost energy
- Compiler reschedule instructions to minimize this cost [Graybill2002 p193]



1) Transitions and commutations

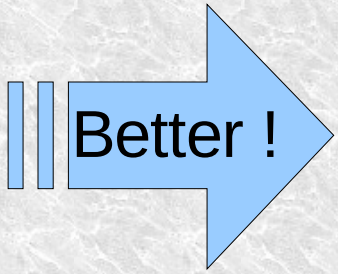
- Register renaming to decrease commutations for the register name
  - Commutation activity on this field -11% [Kandemir2000]
- Global impact ?

2) Loops

- Numerous works [Catthoor,...]...
- Historically for speed
- Ex.: loop unrolling
  - 1 loop with length *n* run *i* times becomes

    1 loop with length *n*x* run *n/x* times

```
for(i=0;i<10000;i++){
   a();b();
}
```




```
for(i=0;i<5000;i++){
   a();b();
   a();b();
}
```

2) Loops

- Impact of loop unrolling:
  - Static instructions duplicated
    - Code size ++
    - Energy ++
  - Less dynamic instructions (for control)
    - Time--
    - Energy --
  - Balance overhead and gain !
- More later (memory, modes)...

3) Execution modes

- Follow program phases
- CPU: DVS/DFS (Dynamic Voltage Scaling / Dynamic Frequency Scaling)
  - $P = C.V^2.f$; $E = P_{avg}.Time$
  - Addresses dynamic P
- Other: sleep modes, hibernation
  - Tend to 0 (unused resource)
  - Addresses both dynamic and static P

3) Execution modes

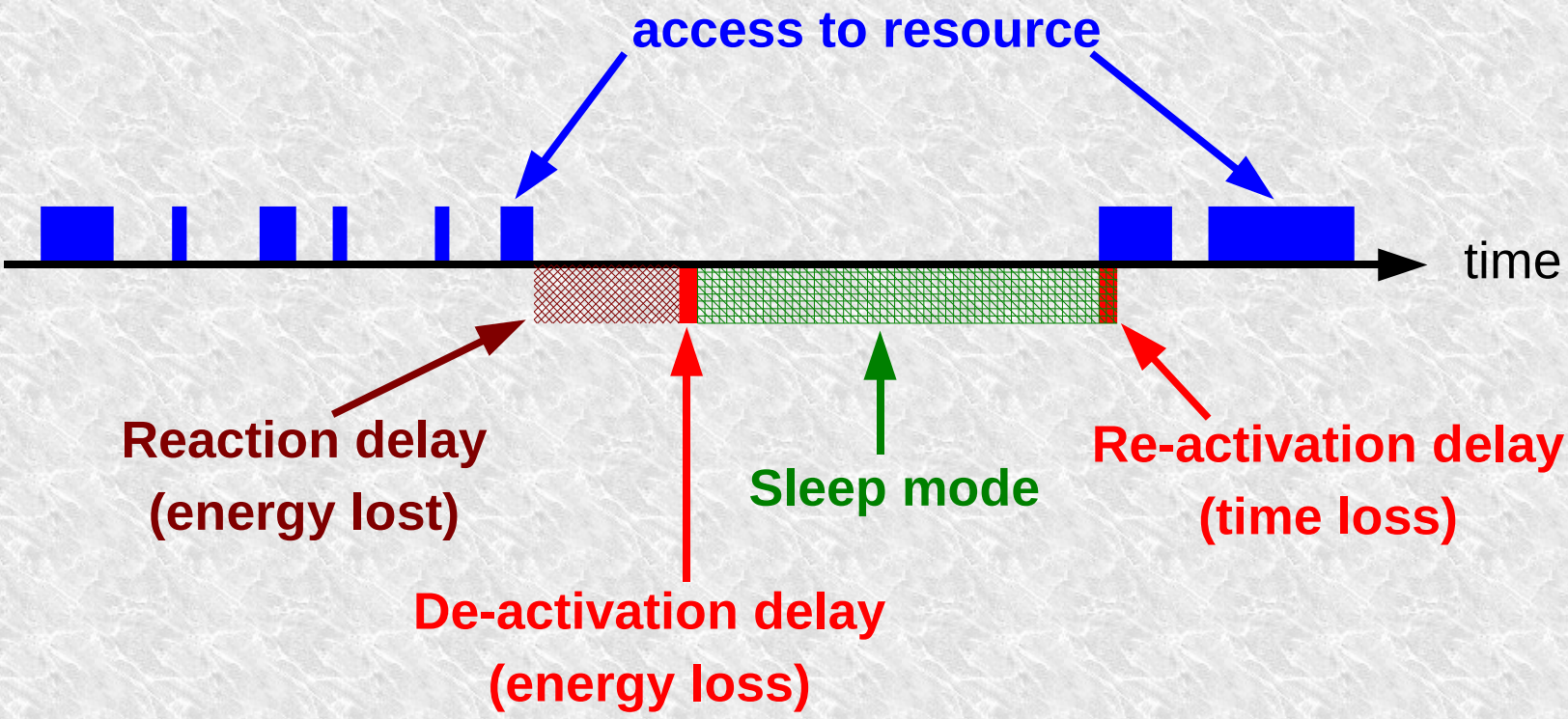
- Very effective to decrease energy
- Here: role of compilation and memory management to take advantage of sleep modes

3) Modes and compilation

- Hardware detects phases of low utilization on one resource
  - Easy *a posteriori*
  - Harder to predict, less certainty
  - Hence useless delay before appropriate action

3) Modes and compilation

- Hardware:

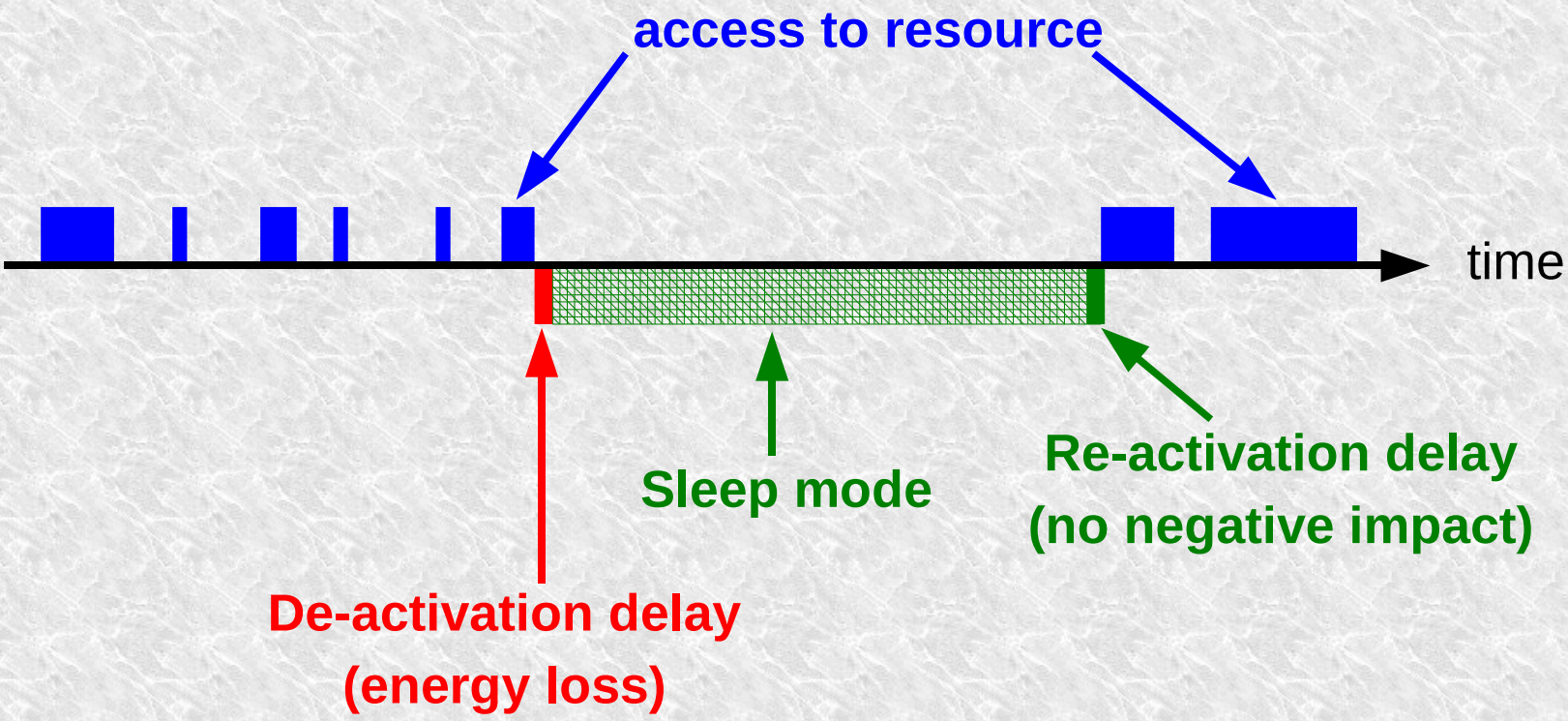


3) Modes and compilation

- Compiler knows points where resource is unused
  - Can be freed immediately
  - Can “warn” of future sleep period
    - And of future re-start
  - No unneeded delay when going to sleep mode or waking-up
    - Better with Energy
    - Better with Time

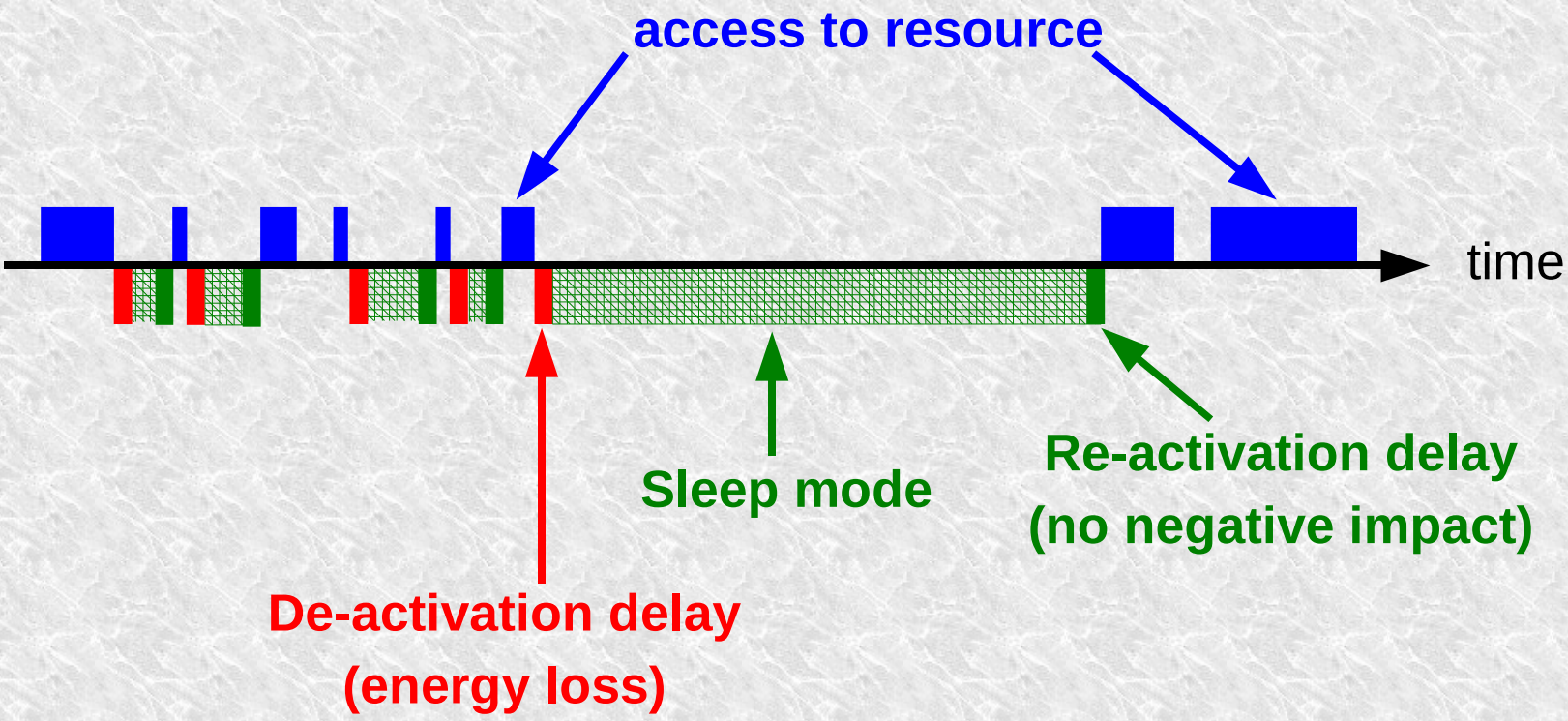
3) Modes and compilation

- Compiler:



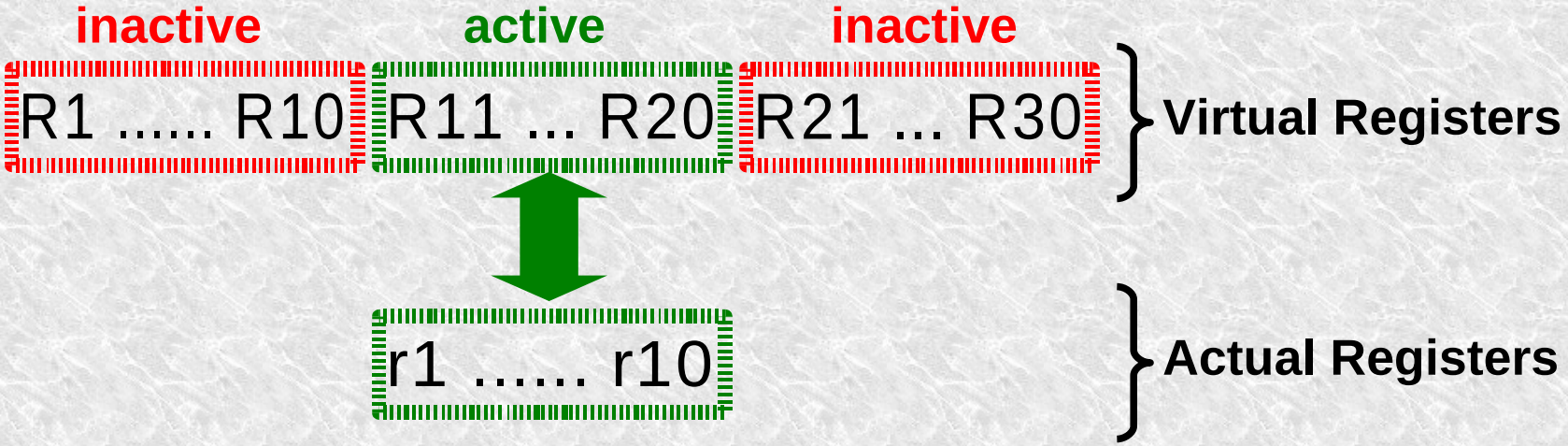
3) Modes and compilation

- Compiler:



4) Register windows: principle

- More virtual registers than actual ones
- V.R. separated in *n* « windows »
- Only 1 register window active at a time



4) Register windows: principle

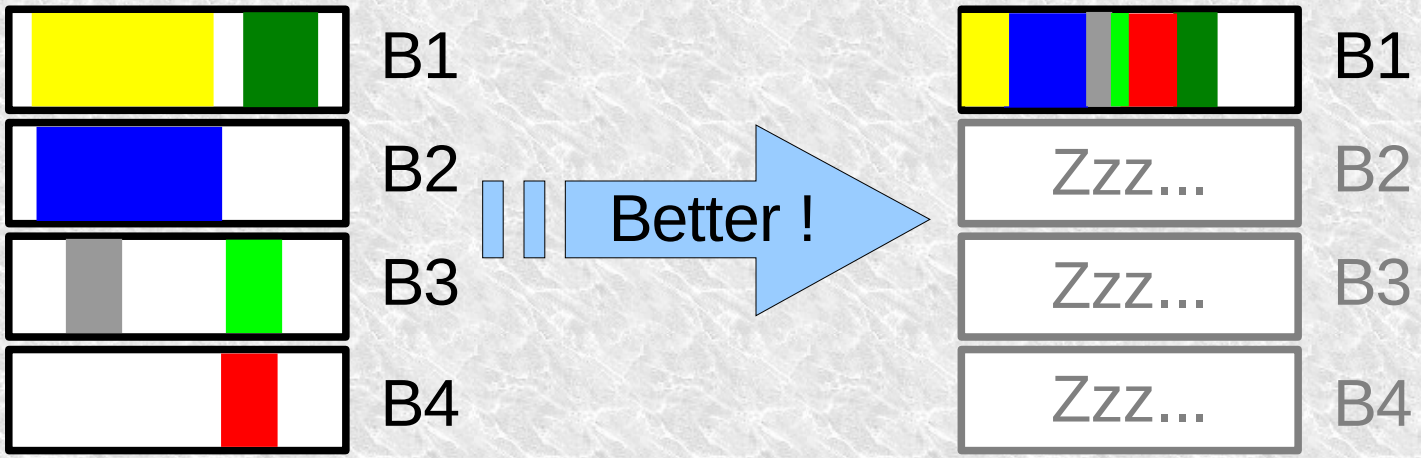
- Change register window according to program phase
  - «One phase runs into one window»
- Reduces *register spill* (=using memory when not enough registers available)
- Management overhead (window change: swap registers ↔ RAM)

4) Register windows: impact

- Work with registers rather than memory:
  - Transfers--
  - (Sleep mode)++
  - Speed++ (created for this)
    - [Ravindran2005] +11%
  - Energy--
    - [Ravindran2005] -25%

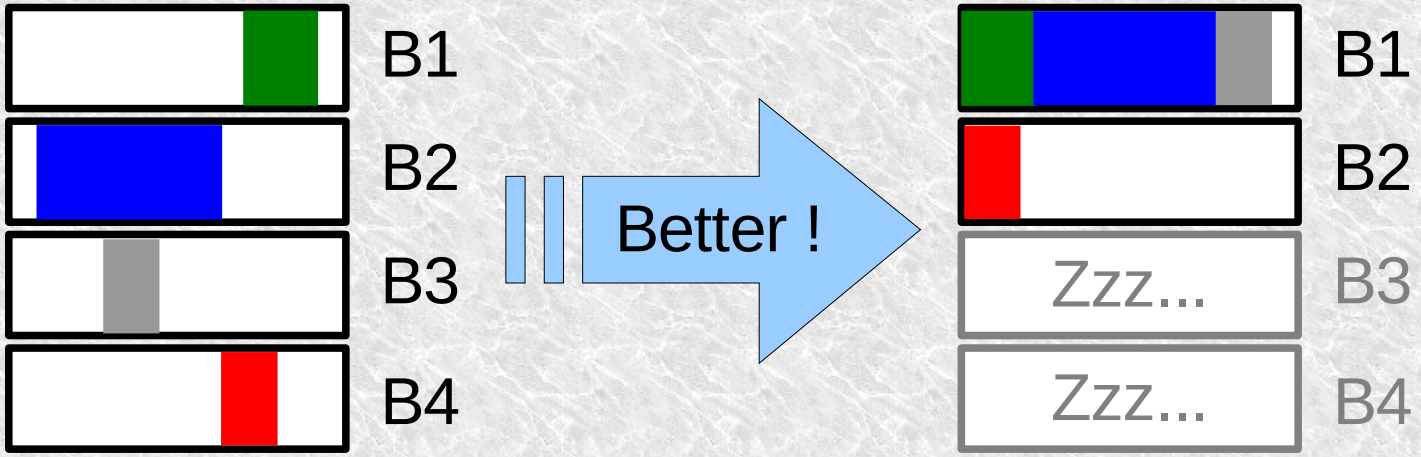
5) Compaction: idea

- Compaction = Less Space
  - Less energy
  - Opportunities for sleep mode (memory banks)



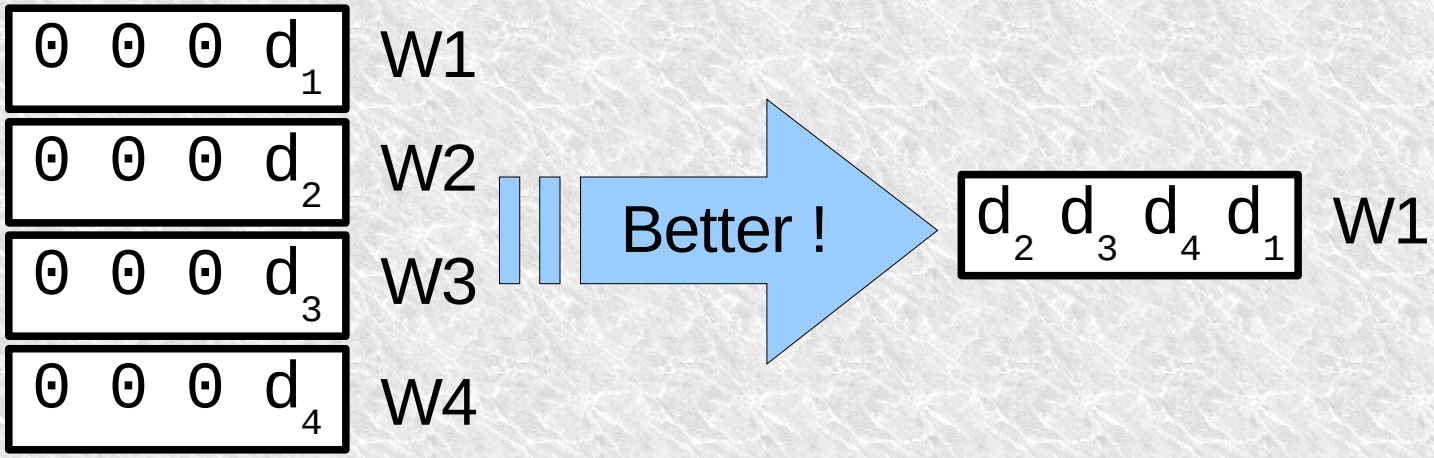
5) Compaction: contiguity

- Allocate and keep data in a minimum of well-filled areas
  - Moves are possible (to avoid fragmentation)
  - May be against speed (the latter may prefer parallel accesses to several banks)



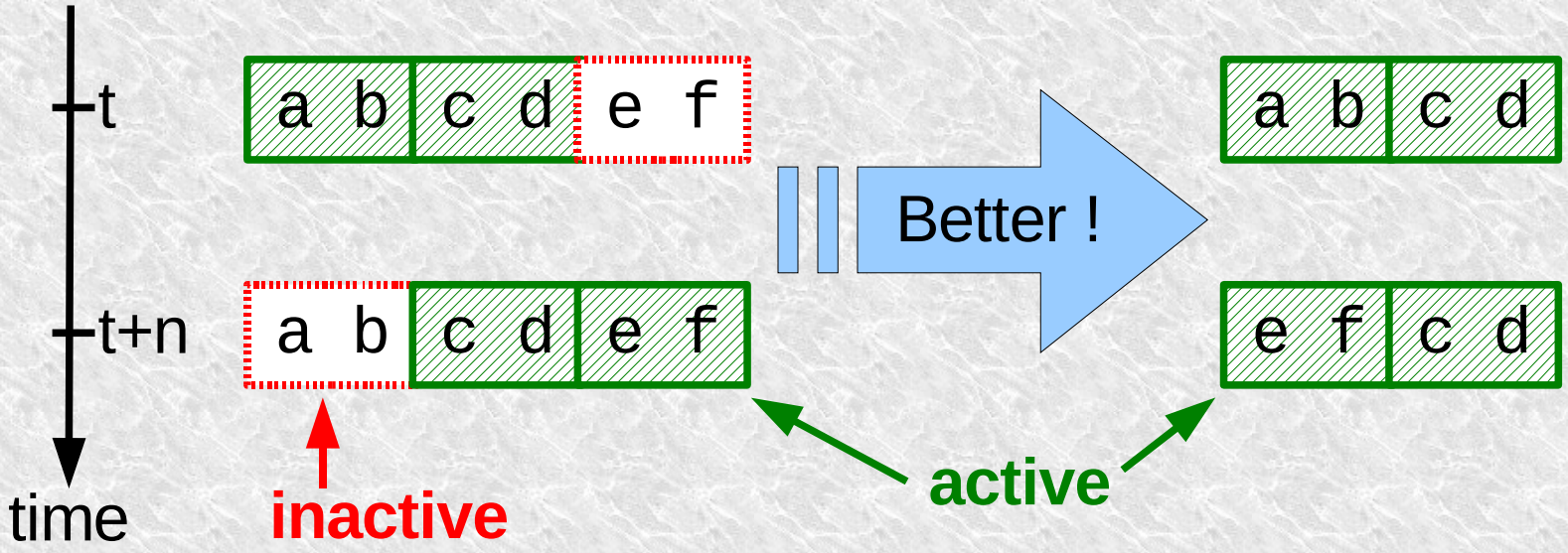
5) Compaction: coalescing

- Variables coalescing: *n* «small» pieces of data into only 1 slot
  - Subword data
    - Bitwidth aware register allocation



5) Compaction: coalescing

- Lifetime analysis: data that do not coexist in the same slot



5) Compaction: compression

- Data compression (larger scale):
  - Beware the potential overhead
  - Strong opportunities
    - Data size --
    - Sleep modes++
    - For long-lived, seldom accessed data

5) Compaction: impact

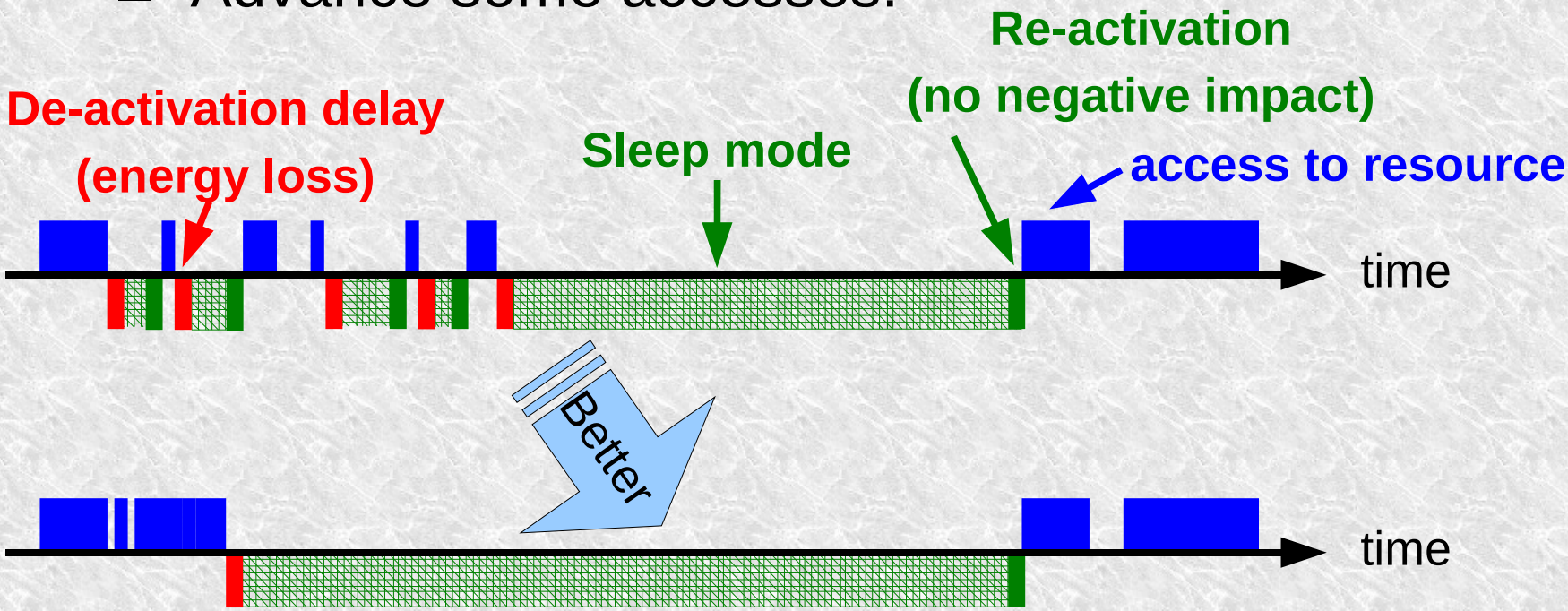
- On variables [Zhuang2003]:
  - Cycles -3%
  - Stack -69%
- On registers [Tallam2003]: -10 to -50% # of registers
- On data (fields) [Zhang2002]:
  - Heap: -25%
  - Energy: -30%
  - Runtime: -12%
    - With ISA *Data Compression eXtensions*: -30%

6) Access re-scheduling: principle

- Improve locality
  - Group accesses to resources
- Increase periods over which a specific resource is unused
- Helps getting into sleep modes

6) Access re-scheduling: code level

- Changes on code
  - Advance some accesses:

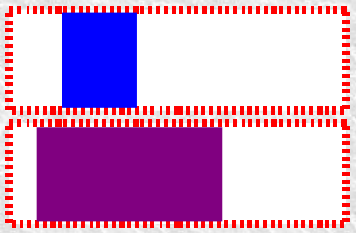


6) Access re-scheduling: loop level

- Loop fission
  - 1 loop becomes *n* loops
  - Process different pieces of data (arrays...): better locality, sleep mode opportunities
  - [Graybill2002,ch10] Energy-- on most expensive loop > energy++ on others (control)

6) Access re-scheduling: loop fission

```
for(i=0;i<10000;i++){
        ...a[...];
        ...b[...];
}
```



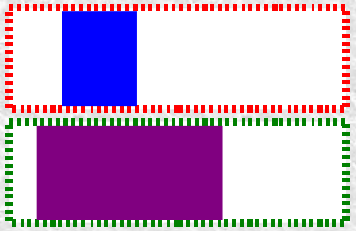
B1

B2

active

Better !

```
for(i=0;i<10000;i++){
        ...a[...];
}
```

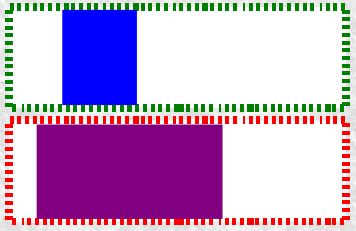


B1 **active**

B2 **inactive**

then

```
for(i=0;i<10000;i++){
        ...b[...];
}
```



B1 **inactive**

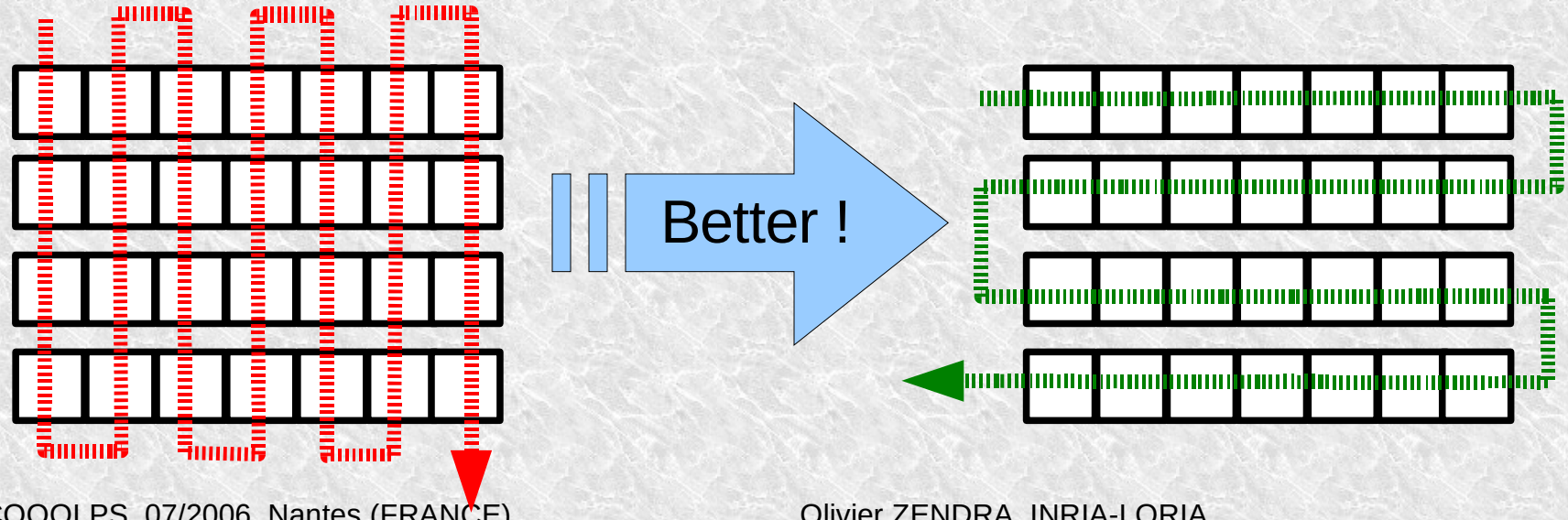
B2 **active**

6) Access re-scheduling: data level

- Change data layout
- Dual of code change
- Very interesting for arrays

6) Access re-scheduling: data level

- Arrays: access according to layout
  - Energy -10% / basic mode control [Athavale2001]



6) Access re-scheduling: data level

- Arrays interlacing (when accessed simultaneously):
  - Energy -8% / basic mode control [Athavale2001]

```
for(i=0;i<10000;i++){
        ...T1[x];
        ...T2[x];
}
```

7) «*scratch-pad*» memory (SPM): motivation

- Caches = speed++
- But poorly adapted to embedded systems
  - Circuit size++ (cache+logic)
  - Energy++
  - Poorly predictable: an issue with real time
- Many cacheless systems

7) «*scratch-pad*» memory: principle

- Small, fast memory area (SRAM,...)
  - Like cache
- Directly and explicitly managed at software level
  - No circuit for its management
  - By developer
  - By compiler

7) «*scratch-pad*» memory: advantages / caches

- Size-- (memory without logic)
  - [Banakar2002] -34% / cache
- Cost--
- Energy--
  - [Banakar2002] -40% / cache
- Predictability++

7) «*scratch-pad*» memory: application domains

- Great if data accesses are known and regular
  - Matrix multiply, audio-video compression algorithms, filtering...
- Good (>cache) if mapping into SPM optimal based on access probabilities
  - Lists, n-trees with low-variation topology [Absar2006]

7) «*scratch-pad*» memory: static management

- Choices (placements) performed entirely off line (at compile time)
  - No move
  - Take into account runtime information with execution profiles (*profiling*)
- Good performance
- Good real time characteristics

8) «*scratch-pad*» memory: dynamic management: principles

- Dynamic allocation (runtime) but decided at compile time
- (Dis)placements performed at runtime
- Regions, program phases, instead of whole program
- More complex, more recent

8) «*scratch-pad*» memory: dynamic management: principles

- Allocation choices based on
  - Usage frequency
  - Transfer costs
  - Size

8) «*scratch-pad*» memory: dynamic management: pros

- Better memory (re)use
  - (Temporary) end of use = freeing SPM immediately is possible
- Better on more complex situations
  - Dynamic creation of tasks, variable dayta size, etc. (MPEG21, MPEG4)

8) «*scratch-pad*» memory: dynamic management: cons

- Real time harder
- Size++ (logic)
- Management overhead (T & E) / static
  - Logic
  - SPM-RAM transfers
    - Cost decreased with DMA support [Francesco2004]
    - Direct allocation in SPM possible
      - Transfer cost = 0

8) «*scratch-pad*» memory: dynamic management: results

- Runtime: -35% / static placement (except heap) in SPM
- Energy: -40% / static placement (except heap) in SPM

9) Importance of global system analysis

- = inter-program optimization
- Scheduling: intrinsic
- Hardware: all programs considered, but not as a whole
- Memory management:
  - OS: multi-program
  - Application: mono-program

9) Importance of global system analysis

- Compilation: rather mono-program
  - Especially static compilation
  - Dynamic compilation: multi-program (JVMs...)
- Crucial to maximize gains
  - Eg. buffer sizing and access clustering : energy -7% to -49% with multi-program optimization wrt. mono-program optimization [Hom2005]

Conclusion and perspectives

- Hardware & compilation complete each other:
  - Compilation: much larger context possible (lots of resources)
  - But exact runtime behavior harder to catch
  - Try to have both
    - Optimizing Virtual Machine does it. But expensive in terms of resources at runtime !

Conclusion and perspectives

- Need support for hardware-software (compiler) interface at the ISA level: synergies
  - «Direct» management of resources by compiler
  - Co-optimizations compiler + hardware, with information transmission between the two

Conclusion and perspectives

- VLIW processors, EPIC: high potential with parallelism
  - Speed++
  - Interesting energy-wise
  - The compiler has to provide the parallelism
    - Lot of work (not yet for generic processors)

Conclusion and perspectives

- Importance of memory and its use: 70% to 90% of energy in 2010 [ITRS]
  - SPM
  - *Energy-aware Garbage Collectors ?*

The end

- Low-energy (and -power) is + and + important
- Hardware can't do all the job
- Compilation, optimization, memory management needed as well on his front
  - Our job(s) !

References

- [Absar2006] Mohammed Javed Absar, Francky Catthoor: *Compiler-Based Approach for Exploiting Scratch-Pad in Presence of Irregular Array Access*, DATE 2005, IEEE Computer Society
- [Athavale2001] R. Athavale, Narayanan Vijaykrishnan, Mahmut T. Kandemir, Mary Jane Irwin: *Influence of Array Allocation Mechanisms on Memory System Energy*. IPDPS 2001: 3.
- [Avissar2002] O. Avissar, R. Barua D. Stewart: *An Optimal Memory Allocation Scheme for Scratch-Pad Based Embedded Systems*. ACM Transactions on Embedded Computing Systems (TECS), 1(1),pp. 6-26, November 2002.
- [Banakar2002] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, Peter Marwedel: *Scratchpad memory: design alternative for cache on-chip memory in embedded systems*. CODES 2002: 73-78

References

- [Dominguez2005] Angel Dominguez, Sumesh Udayakumaran, Rajeev Barua: *Heap Data Allocation to Scratch-Pad Memory in Embedded Systems*. Journal of Embedded Computing, 1(4), 2005, IOS Press.
- [Francesco2004] Poletti Francesco, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, Jose Manuel Mendias: *An integrated hardware/software approach for run-time scratchpad management*. DAC 2004: 238-243
- [Graybill2002] Robert Graybill, Rami Melhem: *Power aware computing*, 2002, Kluwer Academic Publishers.
- [Hom2005] Jerry Hom, Ulrich Kremer: *Inter-program optimizations for conserving disk energy*. ISLPED 2005: 335-338.
- [ITRS] International Technology Roadmap for Semiconductors. http://public.itrs.net/

References

- [Kandemir2000] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, W. Ye, I. Demirkiran: *Register relabeling: A post-compilation technique for energy reduction*, Workshop on Compilers and Operating Systems for Low Power, October 2000.
- [Lee1997] M. Lee, V. Tiwari, S. Malik, M. Fujita: *Power analysis and minimization techniques for embedded DSP software.* IEEE Trans. Very Large Scale Integration, vol. 5, pp. 123--135, Mar. 1997.
- [Ranvindran2005] R.A. Ravindran, R.M. Senger, E.D. Marsman, G.S. Dasika, M.R. Guthaus, S.A. Mahlke, R.B. Brown: *Partitioning variables across register windows to reduce spill code in a low-power processor*, IEEE Transactions on Computers, Vol. 54, Issue 8, 2005, pp. 998-1012.
- [Tallam2003] Sriraman Tallam, Rajiv Gupta: *Bitwidth aware global register allocation*. POPL 2003: 85-96.

References

- [Zhang2002] Youtao Zhang, Rajiv Gupta: *Data Compression Transformations for Dynamically Allocated Data Structures*. CC 2002: 14-28.
- [Zhuang2003] Xiaotong Zhuang, ChokSheak Lau, Santosh Pande: *Storage assignment optimizations through variable coalescence for embedded processors*. LCTES 2003: 220-231