An introduction to memory and compiler optimizations for low- power and energy

Olivier Zendra
INRIA Nancy Grand Est / LORIA
(Nancy, FRANCE)

Olivier.Zendra@inria.fr
http://www.loria.fr/~zendra

Special thanks to

- Organizers for rescheduling my lecture and travel arrangements as late as last Friday
  - Esp. Arnaud Tisserand
  - Esp. ana-Bela Leconte
  - ...
- Olivier Sentieys for switching its time-slot with mine

Aim of this lecture

- Give a taste of what exists: survey
- Give pointers to some interesting works to make digging deeper possible and easier
- Software point of view (compiler, application)
- At compilation level
  - Static
  - Dynamic (i.e in VMs)
  - Not specific in some cases
- At memory management level

Introduction: compilation

- Translating source code into executable
  - With optimization
- Static (off line): cc, gcc
- Dynamic (on line)
  - «Beginning» of runtime: *Just In Time* (JVMs)
    - As a whole
    - Step by step
  - At runtime, with recompilations: optimizing JVMs (à la HotSpot)

Introduction: compilation

- Fixed architecture
- Unknown programs *a priori*
  - Optimize to better execute
- Hardware optimization
  - On-line logic
  - Dedicated circuits
  - Overhead at runtime (Power/Energy and Time)

Introduction: compilation

- 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 much harder to catch

Introduction: compilation

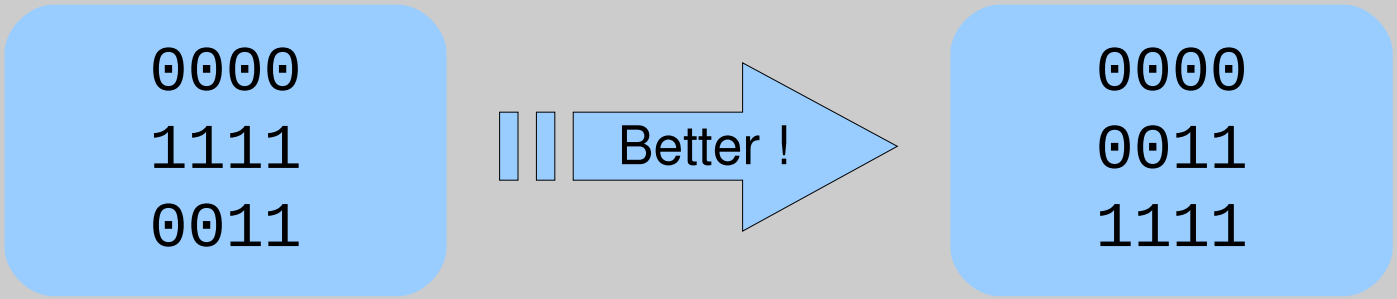
- Energy: new wrt. compilation. Historically, size & speed.
- Optimizations for speed and energy
  - Often related [Lee1997]
  - Not always: moving out of critical path is good for time, but is it for energy ?
- Optimizing for energy != for power density (hot spots)

Introduction

- Low level then high level
- «low» and «high» levels depend from point of view
  - High level optimizations take a larger context into account

Transitions and commutations

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



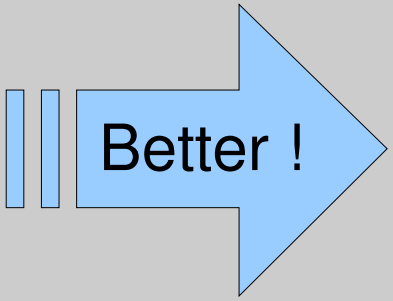
Transitions and commutations

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

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();
}
```



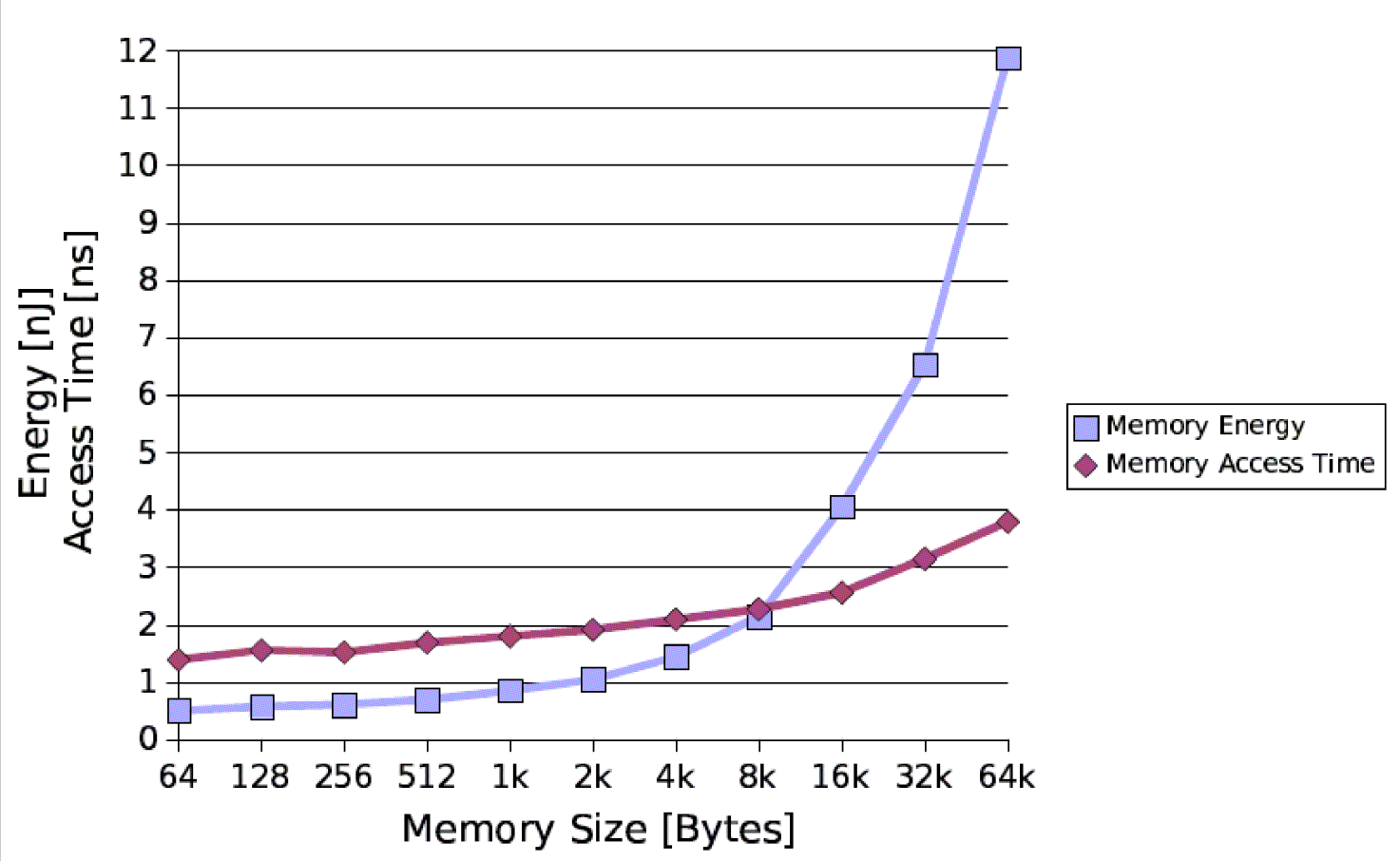

```
for(i=0;i<5000;i++){
   a();b();
   a();b();
}
```

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)...

Memory partitioning

- [Wehmeyer2004]: For all technologies, the larger the memory, the larger the E and T per access:



Memory partitioning: [Wehmeyer2004]

- Idea: have smaller memories
- SPM for instructions and data
- Partition it into smaller contiguous regions
- Energy improvements of up to 22% on the memory subsystem

One word about modes

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

One word about modes

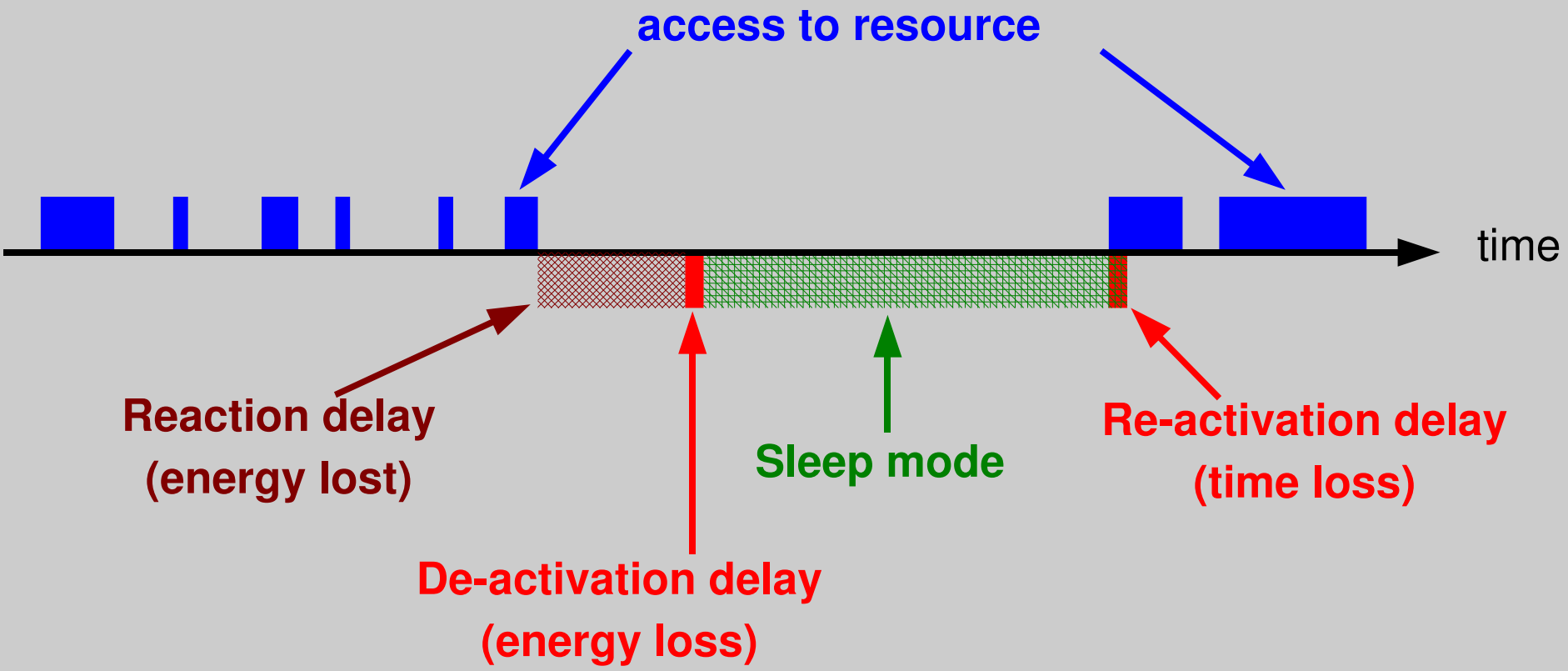
- Very effective to decrease energy
- For more (DVS/DFS): see other presentations
- Here: role of compilation and memory management to take advantage of sleep modes

Modes and compilation

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

Modes and compilation

- Hardware:

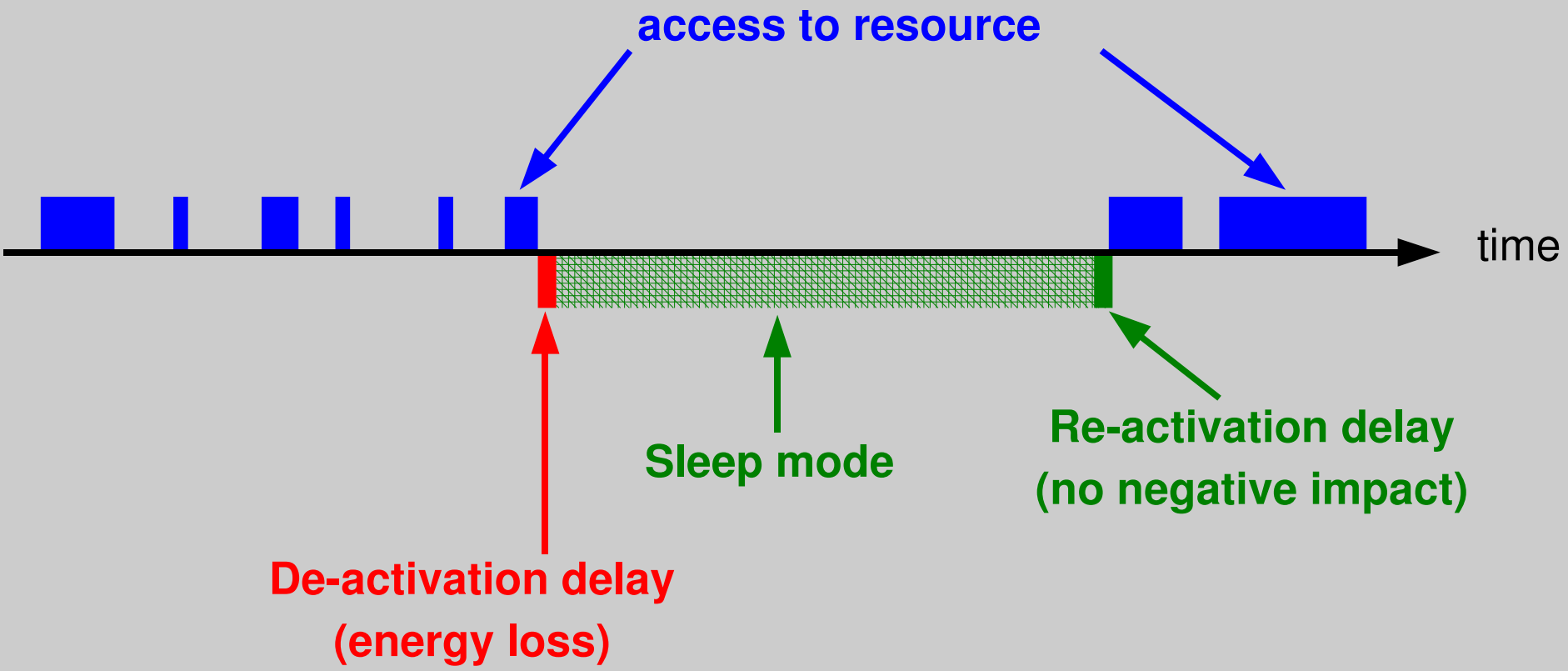


Modes and compilation

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

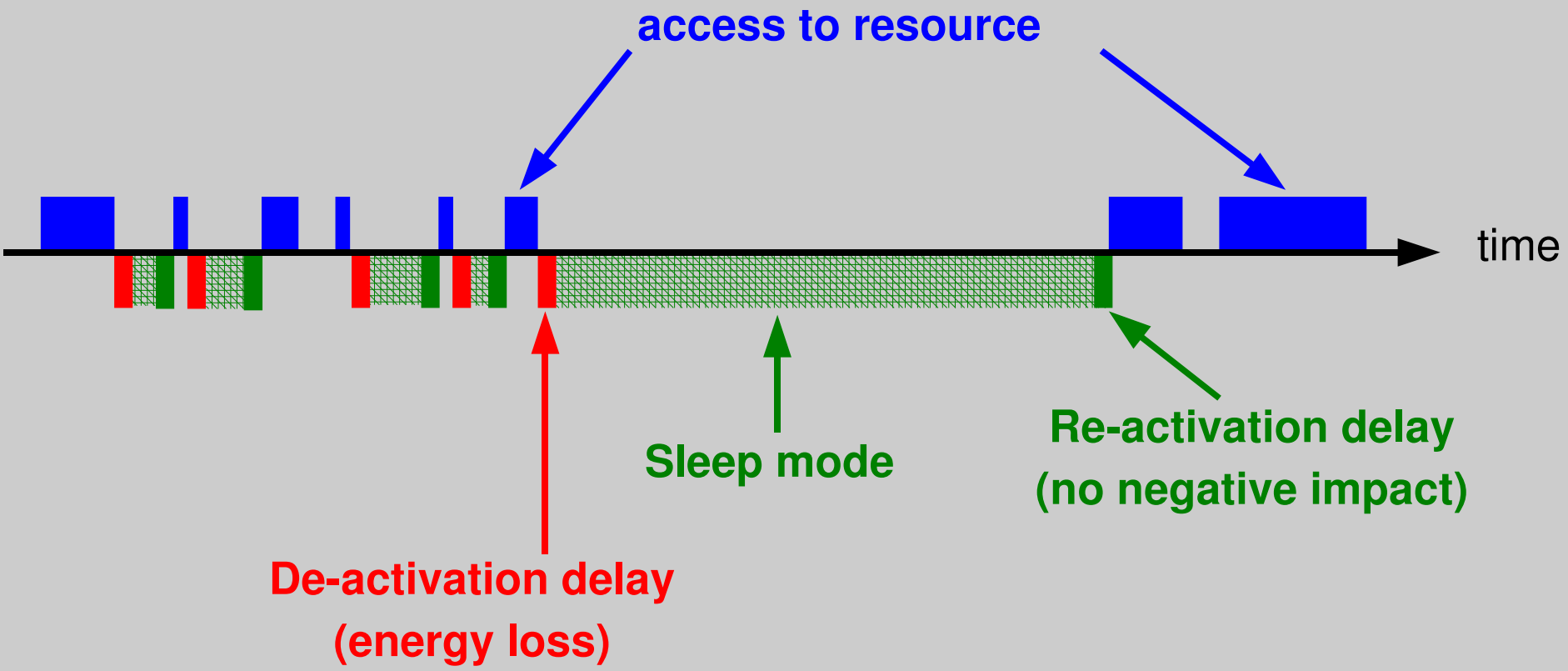
Modes and compilation

- Compiler:



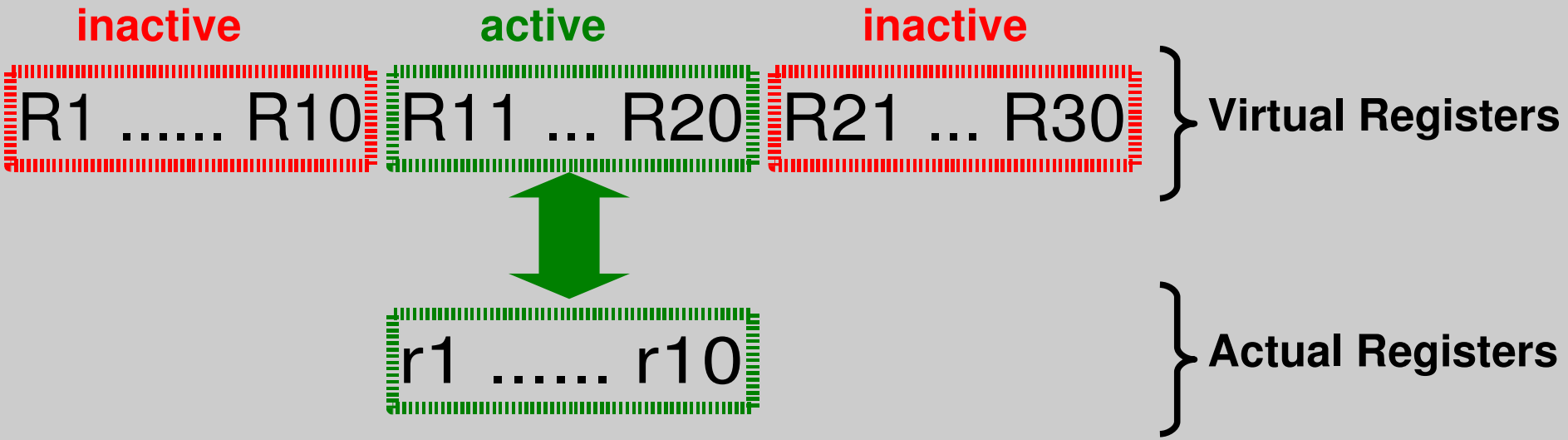
Modes and compilation

- Compiler:



Register windows: principle

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



Register windows: principle

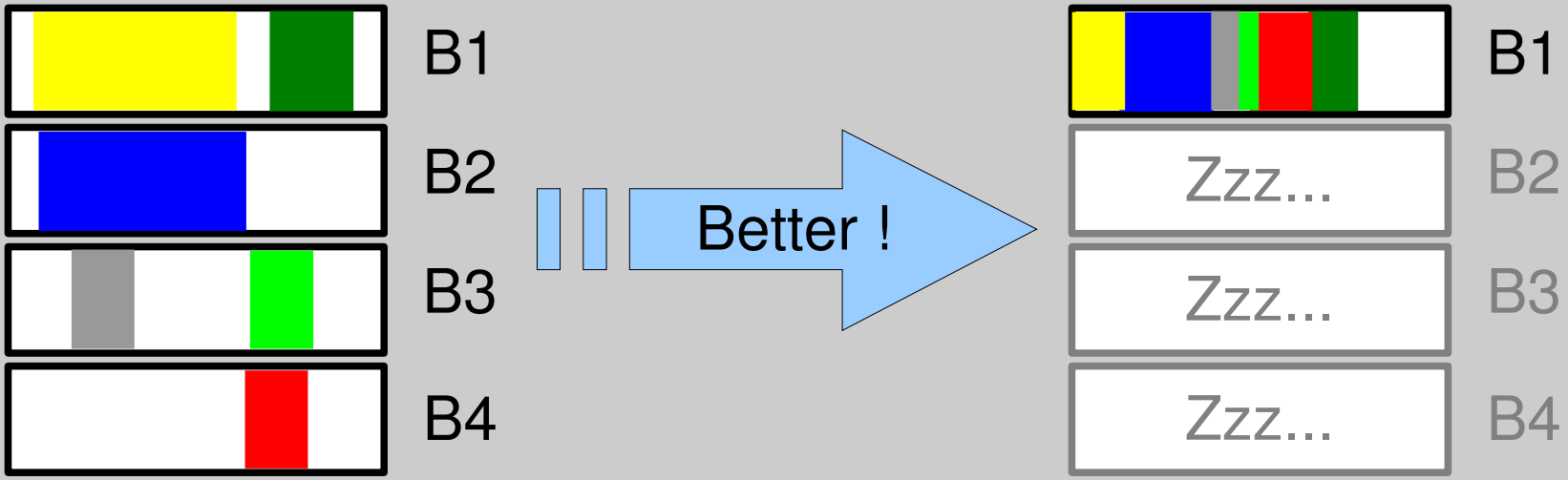
- Change register window according to program phase
  - «One phase runs into one window»
- Reduces *register spill* (=using memory when not enough registers available)
- Management overhead (window change: swap registers ↔ RAM)

Register windows: impact

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

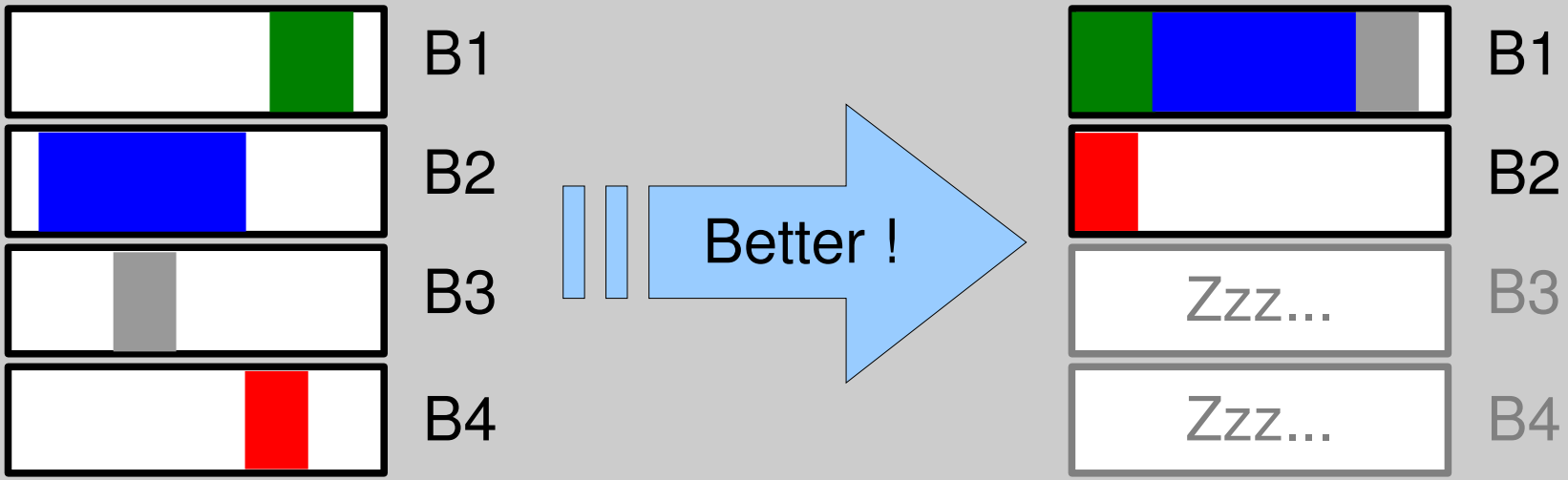
Compaction: idea

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



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 favors parallel accesses to several banks)

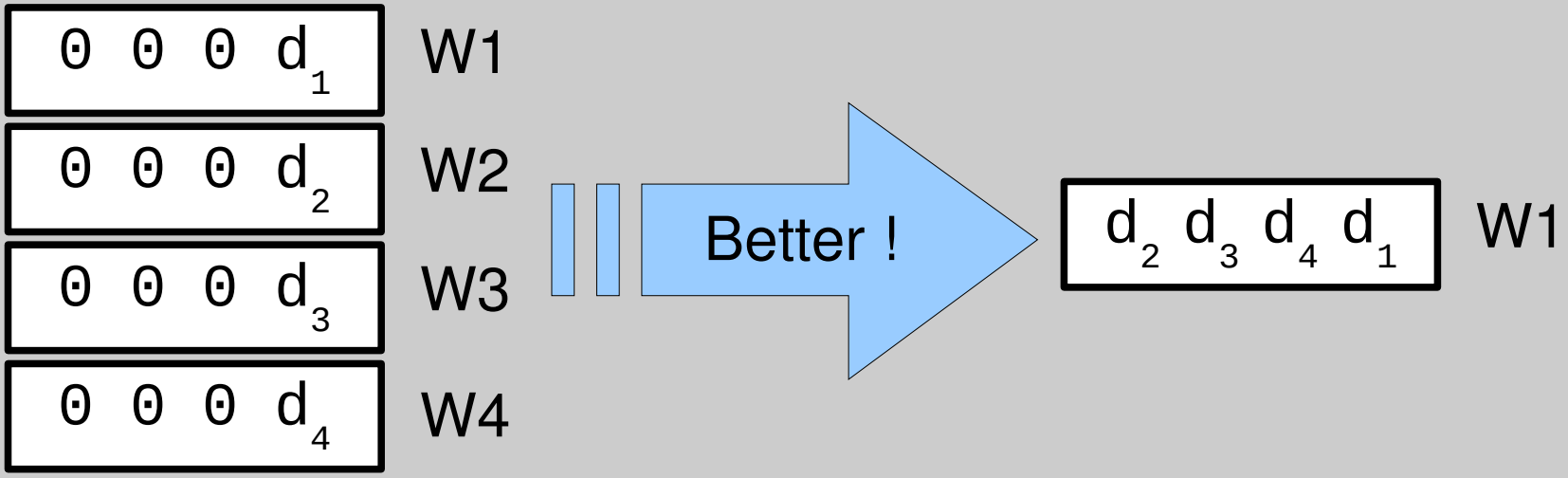


Compaction: contiguity

- Example: [DeLaLuz2006]
  - 3 state model for memory banks:
    - R/W (full dynamic + full leakage)
    - Active (0 dynamic + full leakage)
    - Inactive (supply gated)
  - Applications with dynamic allocation (malloc...)
  - Cluster data with temporal affinity in small # of banks
  - Data migration to better fill banks
    - Bank filled below Migration Threshold is migrated into others and supply gated

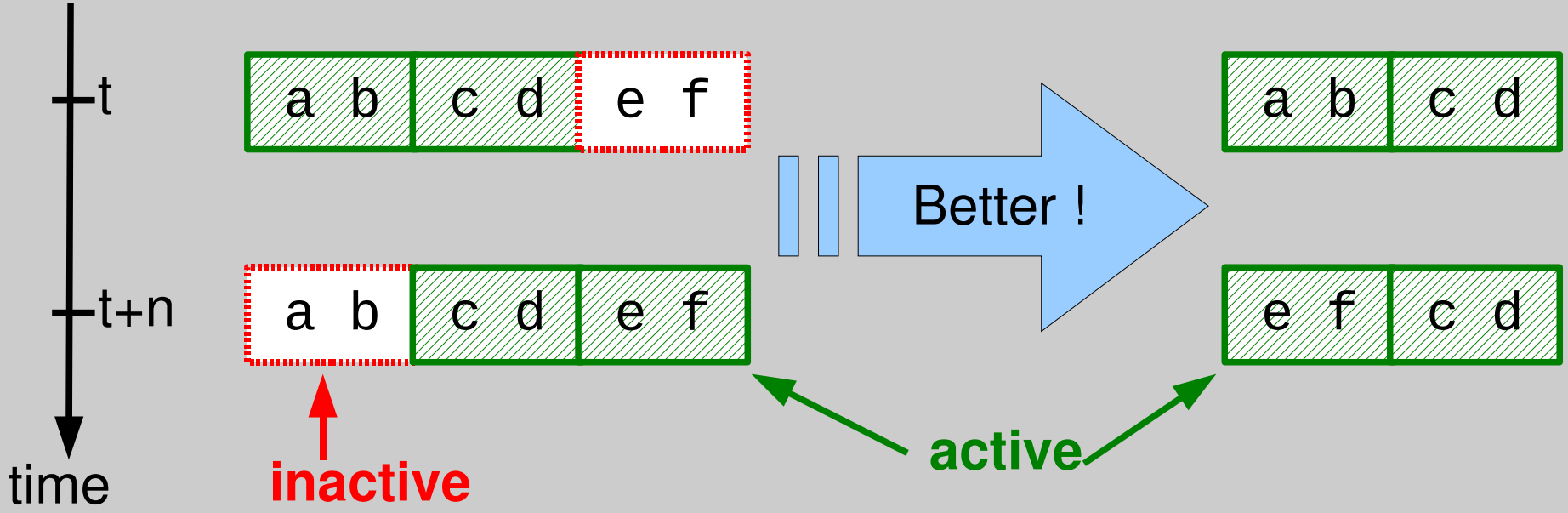
Compaction: coalescing

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



Compaction: coalescing

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



Compaction: compression

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

Mixed Compaction

- Data migration to compact + compression:
  - Migration between DRAM banks at runtime
  - Compression to be more efficient
- So as to better use low-power modes
- [Ozturk2005a]: uniform banking
- [Ozturk2005b]: non-uniform banking
  - Energy savings

Compaction: impact

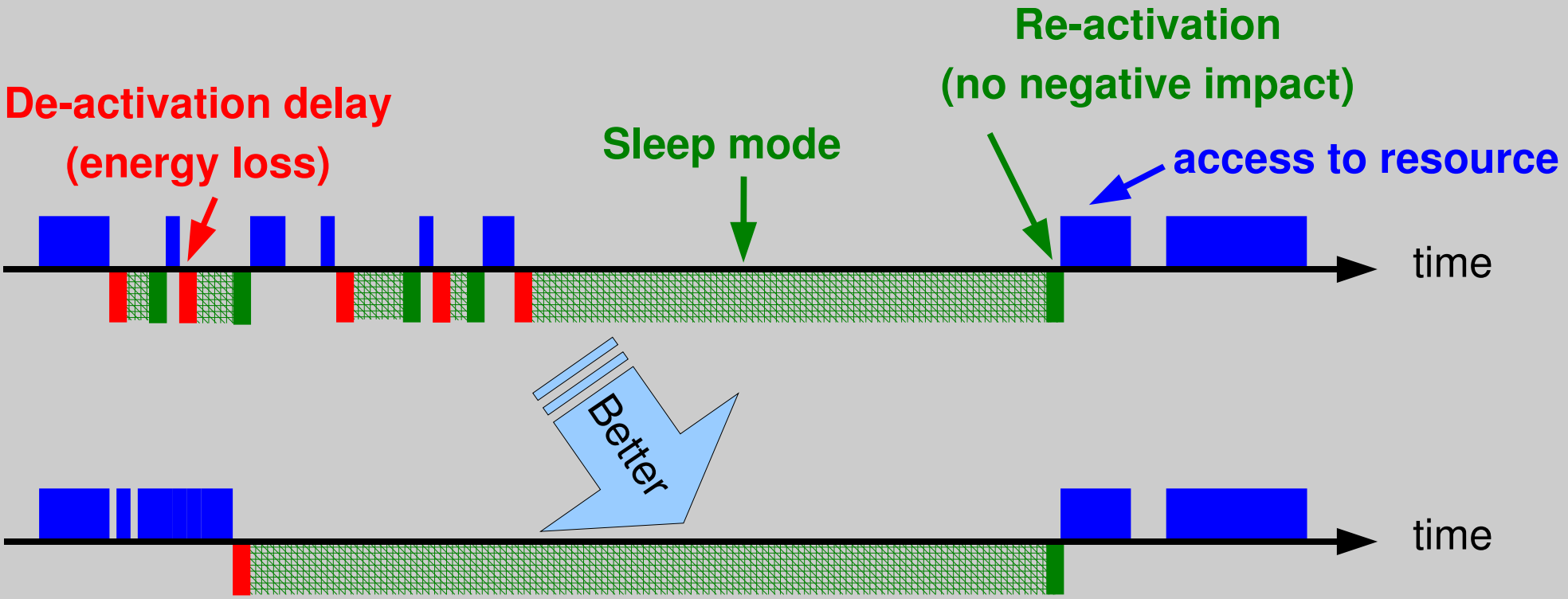
- On variables [Zhuang2003]:
  - Cycles -3%
  - Stack -69%
- On registers [Tallam2003]: -10 to -50% # of registers
- On data (fields) [Zhang2002]:
  - Heap: -25%
  - Energy: -30%
  - Runtime: -12%
    - With ISA *Data Compression eXtensions*: -30%

Access re-scheduling: principle

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

Access re-scheduling: code level

- Changes on code
  - Advance some accesses:

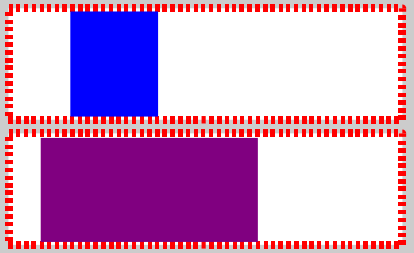


Access re-scheduling: loop level

- 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)

Access re-scheduling: loop fission

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



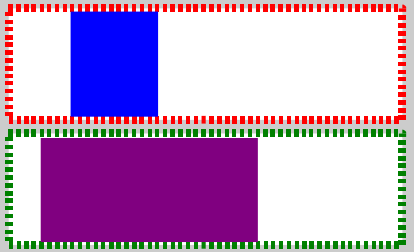
B1

B2

active

Better !

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

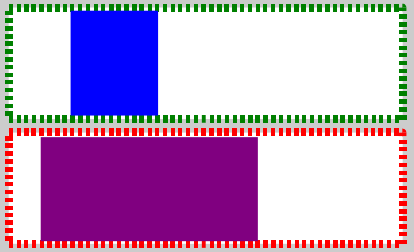


B1 **active**

B2 **inactive**

then

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



B1 **inactive**

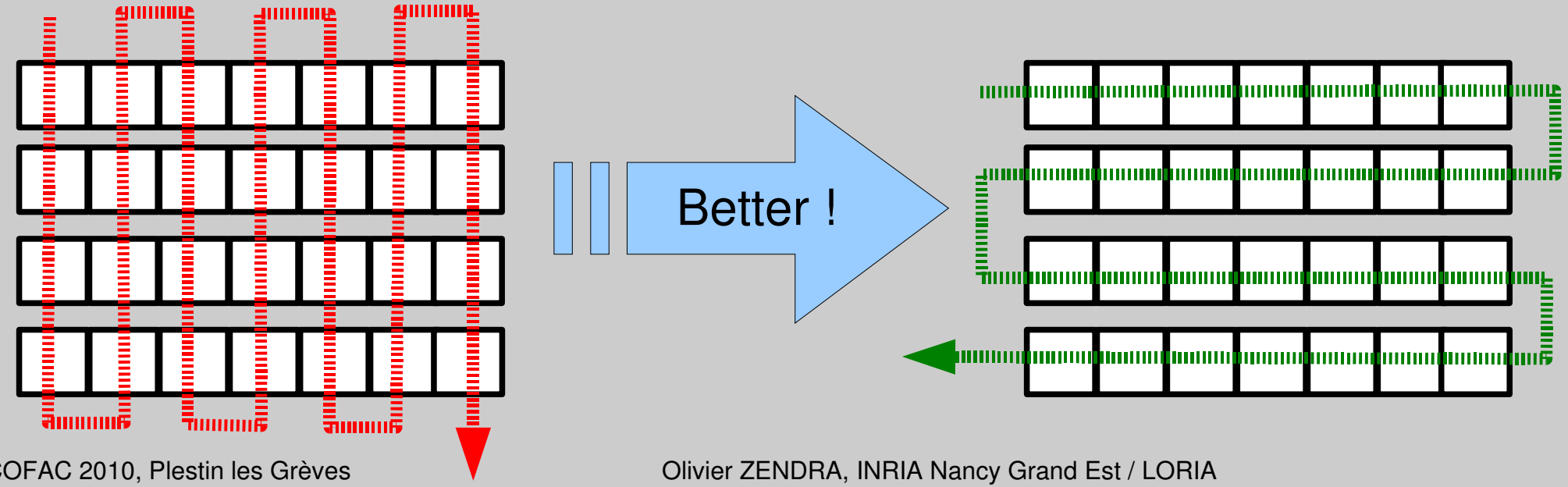
B2 **active**

Access re-scheduling: data level

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

Access re-scheduling: data level

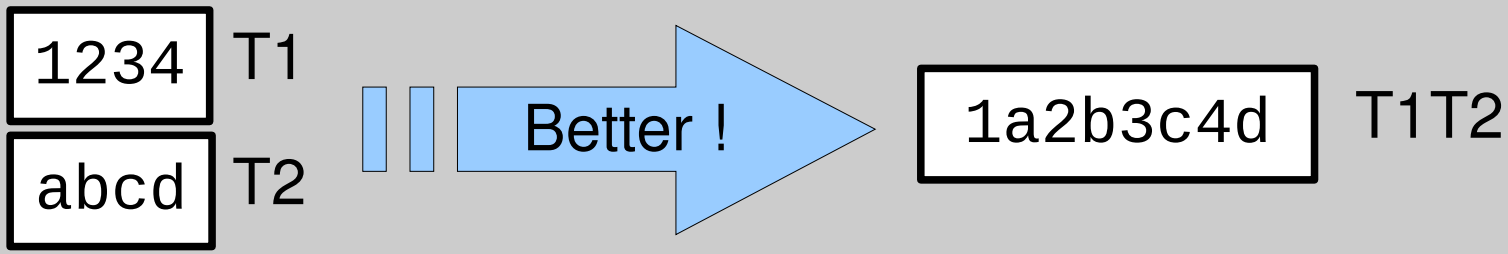
- Arrays: access according to layout
  - Energy -10% / basic mode control [Athavale2001]



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];
}
```



Cache lines sleep mode: [Zhang2003]

- Code analysis at compilation
  - Extract data usage timeline
- Compiler inserts instructions to put cache lines that are not used at the time into sleep mode
- Non destructive: line contents is kept
- 47% to 62% less energy consumption in cache wrt. HW approach, for array-based and pointer intensive computation

«*Scratch-Pad*» Memory (SPM): motivation

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

«*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
  - See [IdrissiAouad2007] for synthesis

«*Scratch-Pad*» Memory: advantages / caches

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

«*Scratch-Pad*» Memory: in actual systems

- Numerous chips with SPM [Nguyen2007a]:
  - Motorola MPC500, Analog Devices ADSP-21XX, Philips LPC2290;
  - Analog Devices ADSP-21160m, Atmel AT91-C140, ARM 968E-S, Hitachi M32R-32192, Infineon XC166
  - Analog Devices ADSP-TS201S, Hitachi SuperH-SH7050, Motorola Dragonball
- More than 80 embedded processors with SPM but no cache [Nguyen2007b]

«*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]

«S*cratch-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 performances
- Good real time characteristics

«*Scratch-Pad*» Memory: static management: [Avissar2002]

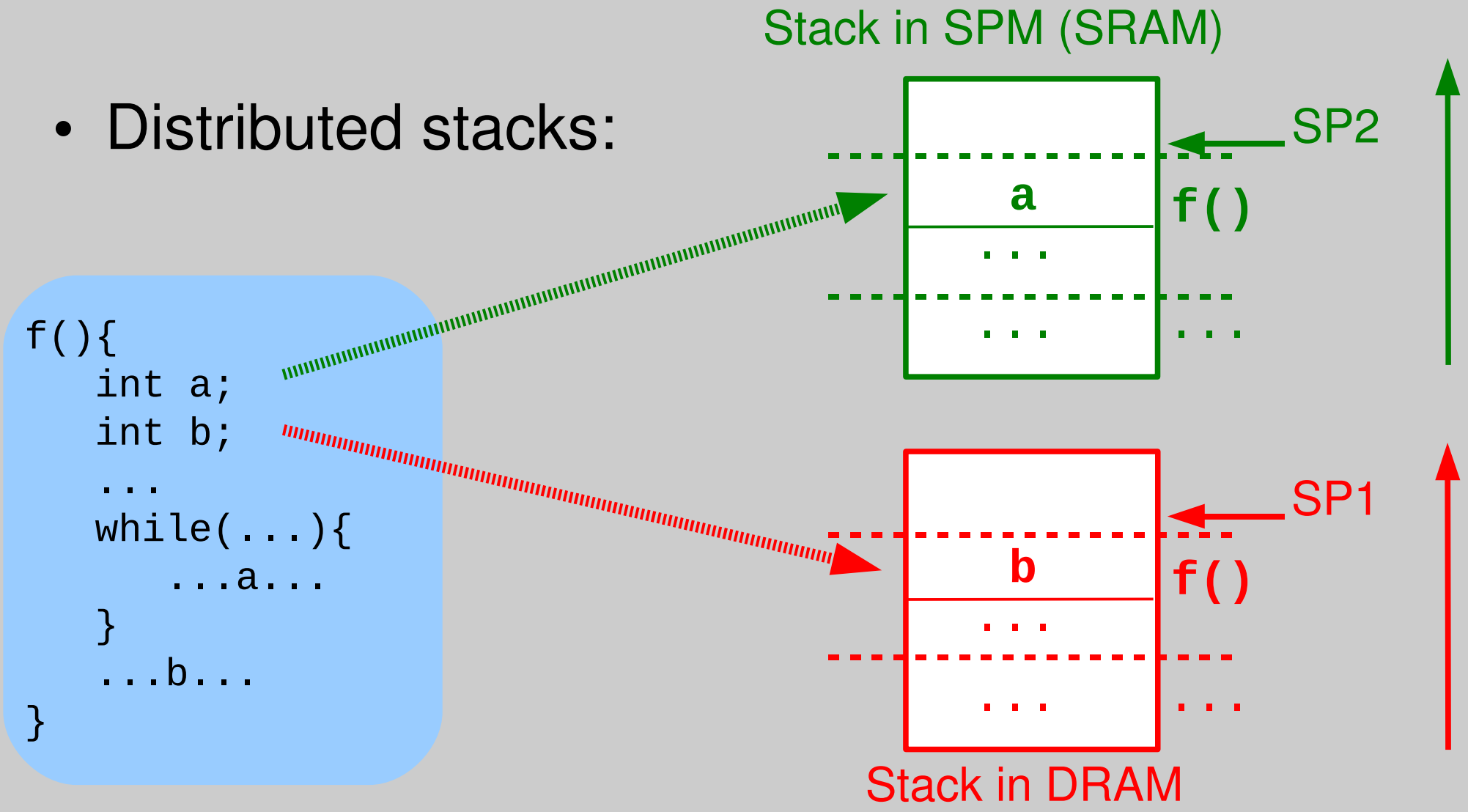
- N memory levels
- 3 types of static allocation:
  - Greedy & unified stack: smallest in SPM & unique stack in DRAM (usual)
  - Greedy & distributed stack: ... & stack in several areas (SPM, DRAM...)
  - (best) Linear formulation & distributed stack: optimize model &... : runtime -56% / all DRAM

«*Scratch-Pad*» Memory: static management: [Avissar2002]

- With profiles: access frequency
- Distributed stacks
  - “Hotter” locals in SPM
  - Several stack pointers (SPx) to update when entering/exiting from a routine
    - Better with only 1 for short routines (all variables in same stack)
  - Execution time -44% / unique stack !
  - Optimum if no recursion

«*Scratch-Pad*» Memory: static management: [Avissar2002]

- Distributed stacks:




```
f(){
   int a;
   int b;
   ...
   while(...){
      ...a...
   }
   ...b...
}
```

«S*cratch-Pad*» Memory: static management: [Avissar2002]

- Optimization on linear formulation
  - Runtime -11%
- 20% of data in SRAM almost as efficient as 100% on all benchmarks
  - Some benchmarks, same with <=5% in SRAM
  - Avoid DRAM alone
- Optimal pour globals
- No heap...

«*Scratch-Pad*» Memory: static management: [Avissar2002]

- Works also with multi-program
- Divide all SRAM between programs based on collective execution profiles
  - Optimal if executed together

«S*cratch-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

«S*cratch-Pad*» Memory: dynamic management: principles

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

«*Scratch-Pad*» Memory: dynamic management: example

- Static version:

```
int a[800];          // SPM (1000)
int b[800];          // DRAM
...
while (i<100000) a[...]... // OK

while (i<100000) b[...]... // bof
...a...b...
```

«S*cratch-Pad*» Memory: dynamic management: example

- Dynamic version:

```
int a[800];          // SPM (1000)
int b[800];          // DRAM
...
while (i<100000) a[...]... // OK
// copy a to DRAM, then b to SPM
while (i<100000) b[...]... // OK
...a...b...
```

«*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)

«*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

«S*cratch-Pad*» Memory: dynamic management: [Dominguez2005]

- First compilation method allowing dynamic management of SPM that allocates heap data in SPM
  - Size allocated at a *site* (malloc / new) unknown
    - Size <= Sizeof(SPM) ?
  - Memory moves = invalid pointers
    - Expensive updates

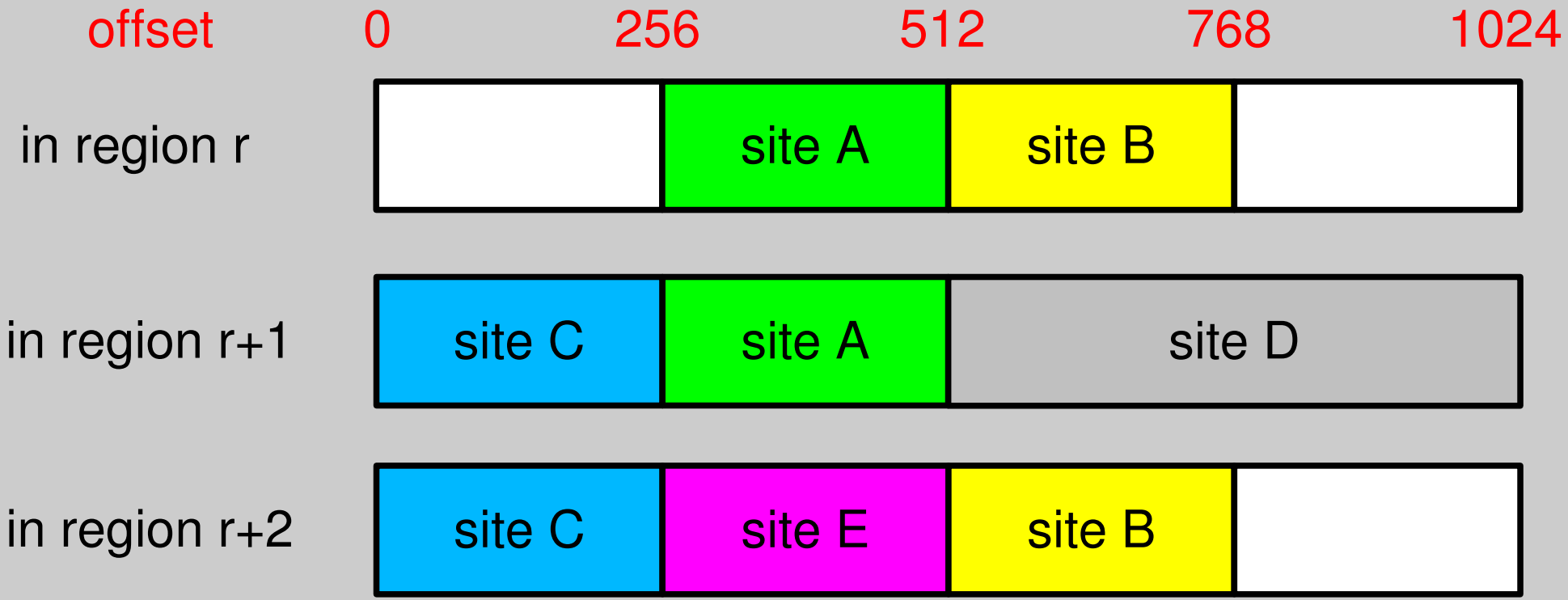
«S*cratch-Pad*» Memory: dynamic management: [Dominguez2005]

- 3 steps:
  - Partition program into regions (begin-end of routine and loop)
  - Determine execution order between regions
  - Insert code at beginning of region *r* to copy parts of the heap (*bins*) to/from SPM (according to usage in *r*)

«S*cratch-Pad*» Memory: dynamic management: [Dominguez2005]

- One *bin* = one fixed-size subset of the objects allocated at a given site
  - Sizeof(*bin*) <= Sizeof(SPM) is guaranteed
  - Offset(*bin*) fixed (if *bin* present in region)
    - Pointers remain valid after moves between SPM and DRAM
  - Heuristics based on cost to decide size and contents of *bins*
    - Sites with higher per byte access frequency have larger *bin*

«*Scratch-Pad*» Memory: dynamic management: [Dominguez2005]



«*Scratch-Pad*» Memory: dynamic management: [Dominguez2005]

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

«*Scratch-Pad*» Memory: dynamic management: [Kandemir2005]

- SPM divided into banks
- Compiler guided
  - Data layout optimization
  - Data migration
- In order to maximize SPM bank idleness
- Opportunities++ for low-power modes
  - Reduce leakage

«*Scratch-Pad*» Memory: coupling with... cache !

- SPM is good to capture large granularity
  - Even with dynamic management, small phases do escape
- Caches adapt permanently, but cost energy
- Idea: use cache as a “backup” for finer details/phases
  - Hide the defects of the SPM management
- Further reduction of energy [Egger2006]

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

Importance of global system analysis

- Compilation: rather mono-program
  - Especially static compilation
  - Dynamic compilation: multi-program (JVMs...)
- Crucial to maximize gains
  - Eg. buffer size 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
  - Compiler has to provide the parallelism
    - Lot of work (not yet for generic processors)

Conclusion and perspectives

- Importance of memory and how it is used: 75,4% of total energy of a SoC in 2022 [ITRS]
  - SPM
  - *Energy-aware Garbage Collectors* ?

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

- [DeLaLuz2006] V. De La Luz, M. T. Kandemir, I. Kolcu: Reducing memory energy consumption of embedded applications that process dynamically allocated data. IEEE Trans. on CAD of Integrated Circuits and Systems, 25(9), 2006. pp 1855—1860.
- [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.
- [Egger2006] B. Egger, J. Lee, H. Shin: *Scratchpad Memory Management for Portable Systems with a Memory Management Unit.* EMSOFT, 2006
- [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.

References

- [Hom2005] Jerry Hom, Ulrich Kremer: *Inter-program optimizations for conserving disk energy*. ISLPED 2005: 335-338.
- [IdrissiAouad2007] M. Idrissi Aouad, O. Zendra: *A Survey of Scratch-Pad Memory Management Techniques for low-power and -energy*. ICOOOLPS 2007.
- [ITRS] International Technology Roadmap for Semiconductors. http://public.itrs.net/
- [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.
- [Kandemir2005] M. T. Kandemir, M. J. Irwin,G. Chen, I. Kolcu: *Compiler-guided leakage optimization for banked scratch-pad memories.* IEEE Trans. VLSI Syst, 13(10), pp 1136–1146, 2005.
- [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.

References

- [Nguyen2007a] Nghi Nguyen, Angel Dominguez, Rajeev Barua: *Scratch-pad memory allocation without compiler support for java applications.* CASES 2007, pp. 85—94
- [Nguyen2007b] Nghi Nguyen, Angel Dominguez, Rajeev Barua: *Memory Allocation for Embedded Systems with a Compile-Time-Unknown Scratch-Pad Size*. To appear in the ACM Transactions on Embedded Computing Systems (TECS), 2007
- [Ozturk2005a] O. Ozturk M. T. Kandemir. *Integer linear programming based energy optimization for banked DRAMs.* ACM Great Lakes Symposium on VLSI, 2005. pp 92—95.
- [Ozturk2005b] O. Ozturk M. T. Kandemir. *Nonuniform Banking for Reducing Memory Energy Consumption.* DATE 2005. pp 814—819.
- [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.

References

- [Tallam2003] Sriraman Tallam, Rajiv Gupta: *Bitwidth aware global register allocation*. POPL 2003: 85-96.
- [Wehmeyer2004] L. Wehmeyer, U. Helmig, P. Marwedel: *Compiler-optimized usage of partitioned memories.* WMPI 2004.
- [Zhang2002] Youtao Zhang, Rajiv Gupta: *Data Compression Transformations for Dynamically Allocated Data Structures*. CC 2002: 14-28.
- [Zhang2003] W. Zhang, M. Karaköy, M. T. Kandemir, G. Chen: *A compiler approach for reducing data cache energy*. ICS 2003, pp 76 – 85
- [Zhuang2003] Xiaotong Zhuang, ChokSheak Lau, Santosh Pande: *Storage assignment optimizations through variable coalescence for embedded processors*. LCTES 2003: 220-231

benchmarks used

- [Avissar2002]: Consortium Trimaran (bmm, fir), Rutter (btoa), MiBench (crc32, djikstra), UTDSP (fft, iir, latnrm)
- [Athavale2001]: adi, amhmtm, btrix, dtdtz, eflux, tomcat, tsf, vpenda
- [Dominguez2005]: Huff, Drystone, Susan, Gsm, KS
- [Hom2005]: mpeg_play, mpg123, sftp
- [Ravindran2005]: fir, rawd, sha, g721enc, g721dec, gsmenc, gsmdec, epic, unepic, cjpeg, djpeg, rijndael, pgpenc, pgpdec (Mediabench & MiBench)
- [Tallam2003]: Mediabench (adpcm, g721), NetBench (crc, dh), Bitwise du MIT (SoftFloat, NewLife, MotionTest, Bubblesort, Histogram), thres
- [Zhang2002]: treeadd, bisort, tsp, perimeter, health, mst
- [Zhuang2003]: epic, gsm, g721, Mpeg2d, Mpeg2e, Bzip2, Gzip, Mcf, Twolf, Vpr (MediaBench & Spec2000int)