technische universität
dortmund

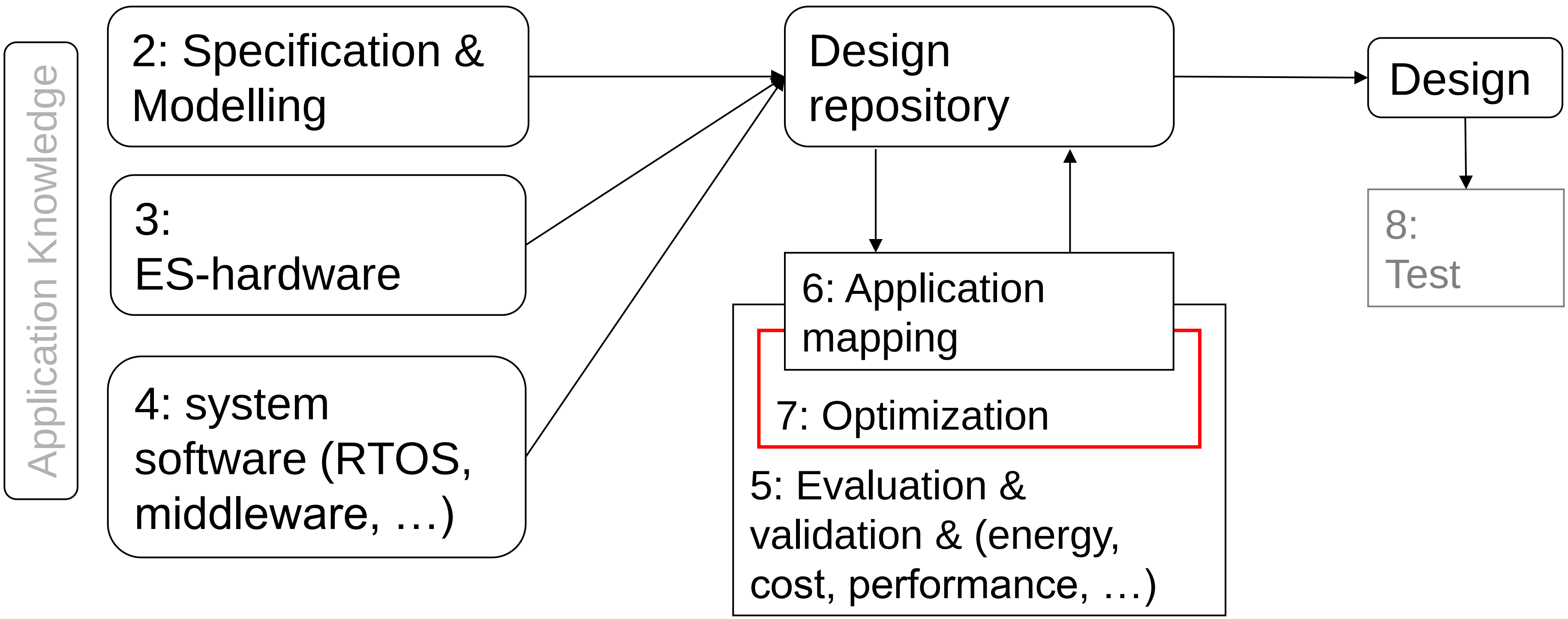
fakultät für informatik
informatik 12

Optimizations
- Compilation for Embedded Processors -

Peter Marwedel
TU Dortmund
Informatik 12
Germany

2014年 01 月 14 日

Structure of this course



Numbers denote sequence of chapters

Compilers for embedded systems

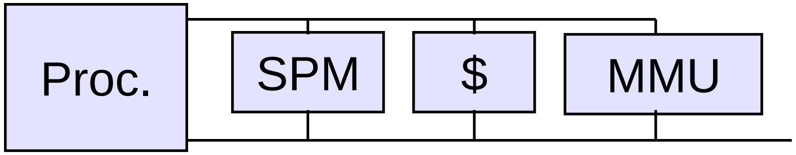
Book section 7.3

- Introduction
- Energy-aware compilation
- Memory-architecture-aware compilation
- Reconciling compilers and timing analysis
- Compilation for digital signal processors
- Compilation for VLIW processors
- Compiler generation, retargetable compilers, design space exploration

SPM+MMU (1)

How to use SPM in a system with virtual addressing?

- **Virtual SPM**
  Typically accesses MMU
  \+ SPM in parallel
  ☞ not energy efficient



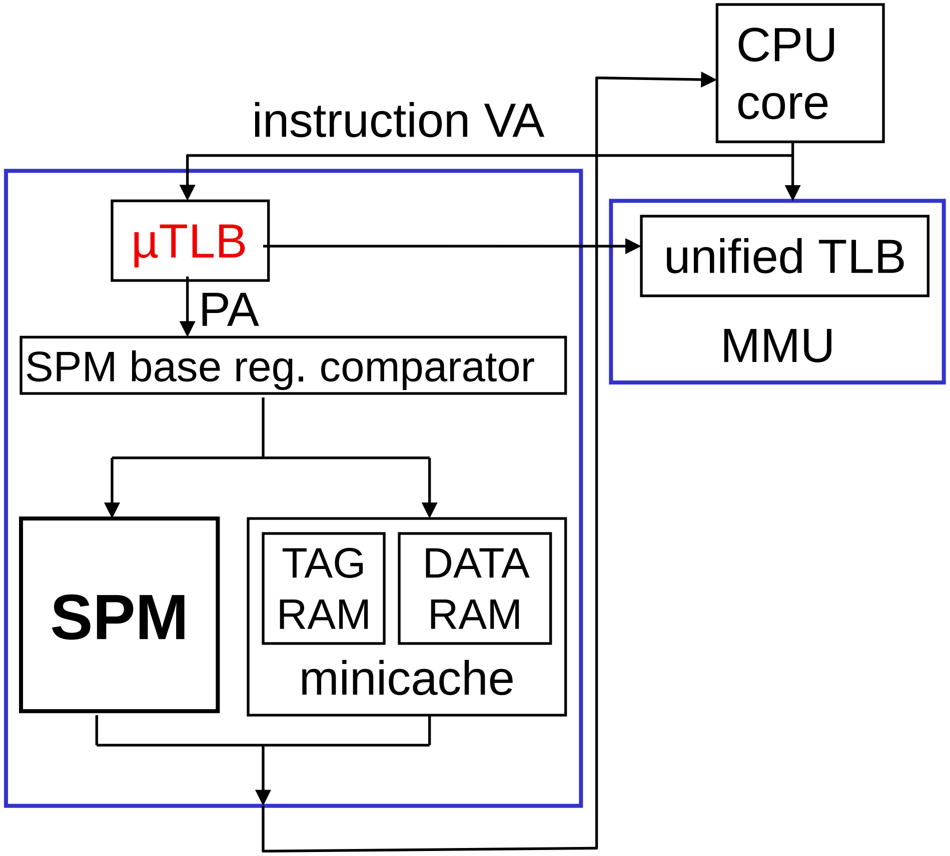
- **Real SPM**
  ☞ suffers from potentially long VA translation

- Egger, Lee, Shin (Seoul Nat. U.):
  Introduction of small **µTLB** translating recent addresses fast.

[B. Egger, J. Lee, H. Shin: Scratchpad memory management for portable systems with a memory management unit, *CASES*, 2006, p. 321-330 (best paper)]

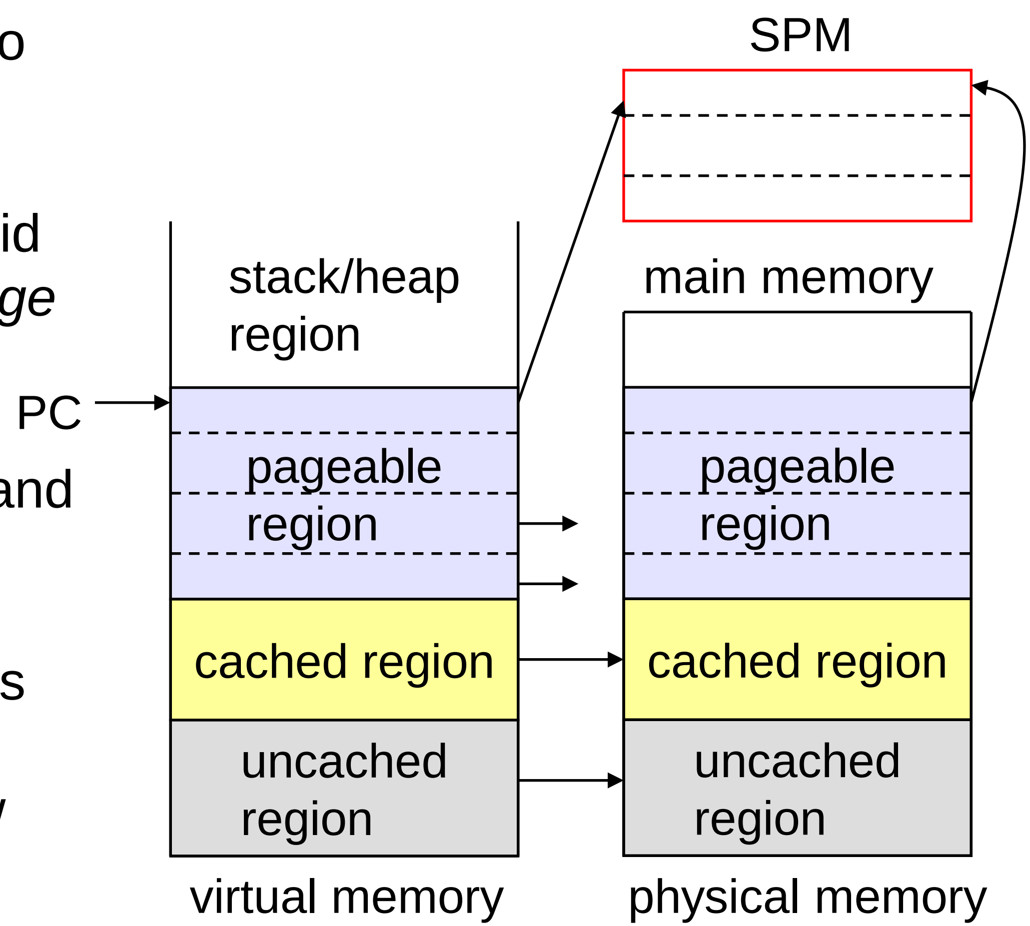
SPM+MMU (2)

- µTLB generates physical address in 1 cycle
- if address corresponds to SPM, it is used
- otherwise, mini-cache is accessed
- Mini-cache provides reasonable performance for non-optimized code
- µTLB miss triggers main TLB/MMU
- SPM is used only for instructions
- instructions are stored in pages
- pages are classified as cacheable, non-cacheable, and "pageable" (= suitable for SPM)

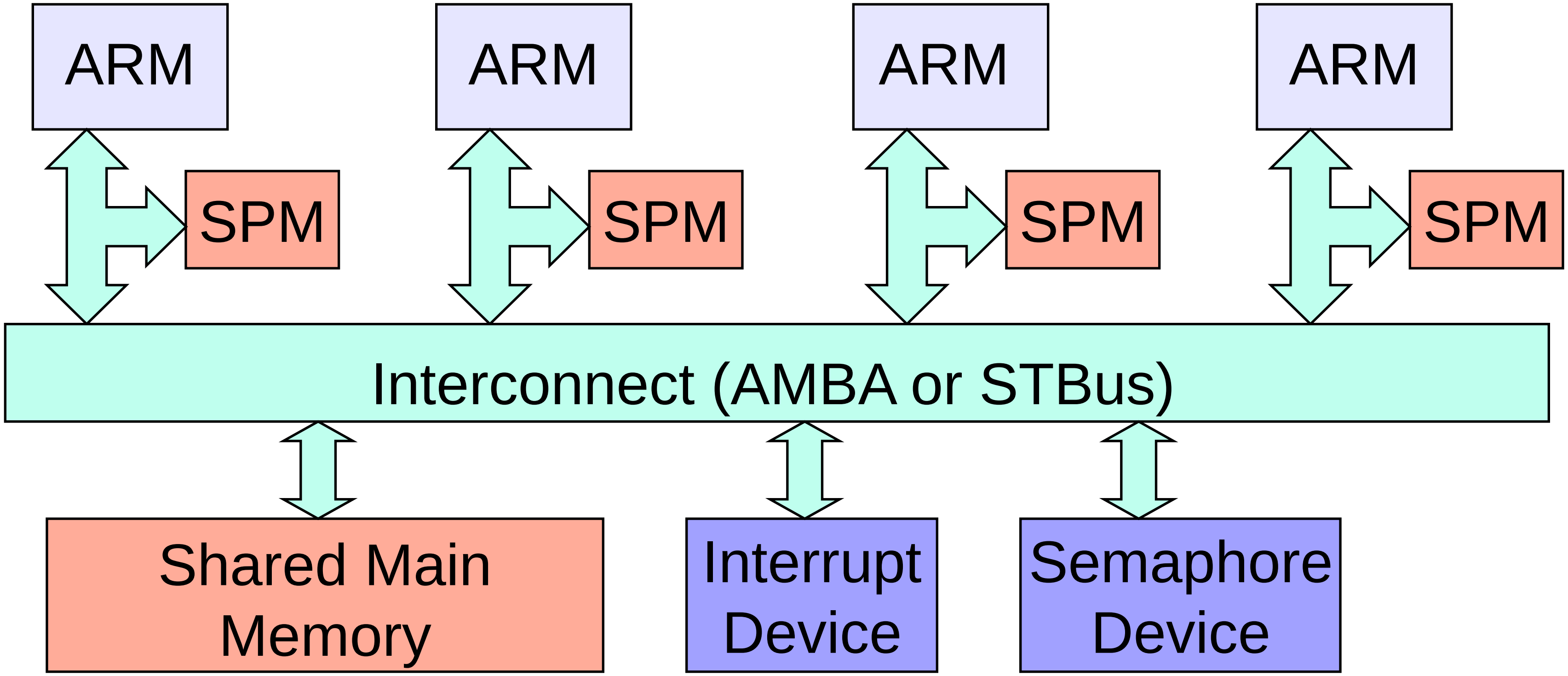


SPM+MMU (3)

- Application binaries are modified: frequently executed code put into pageable pages.
- Initially, page-table entries for pageable code are marked invalid
- If invalid page is accessed, a *page table exception* invokes SPM manager (SPMM).
- SPMM allocates space in SPM and sets page table entry
- If SPMM detects more requests than fit into SPM, SPM eviction is started
- Compiler does not need to know SPM size

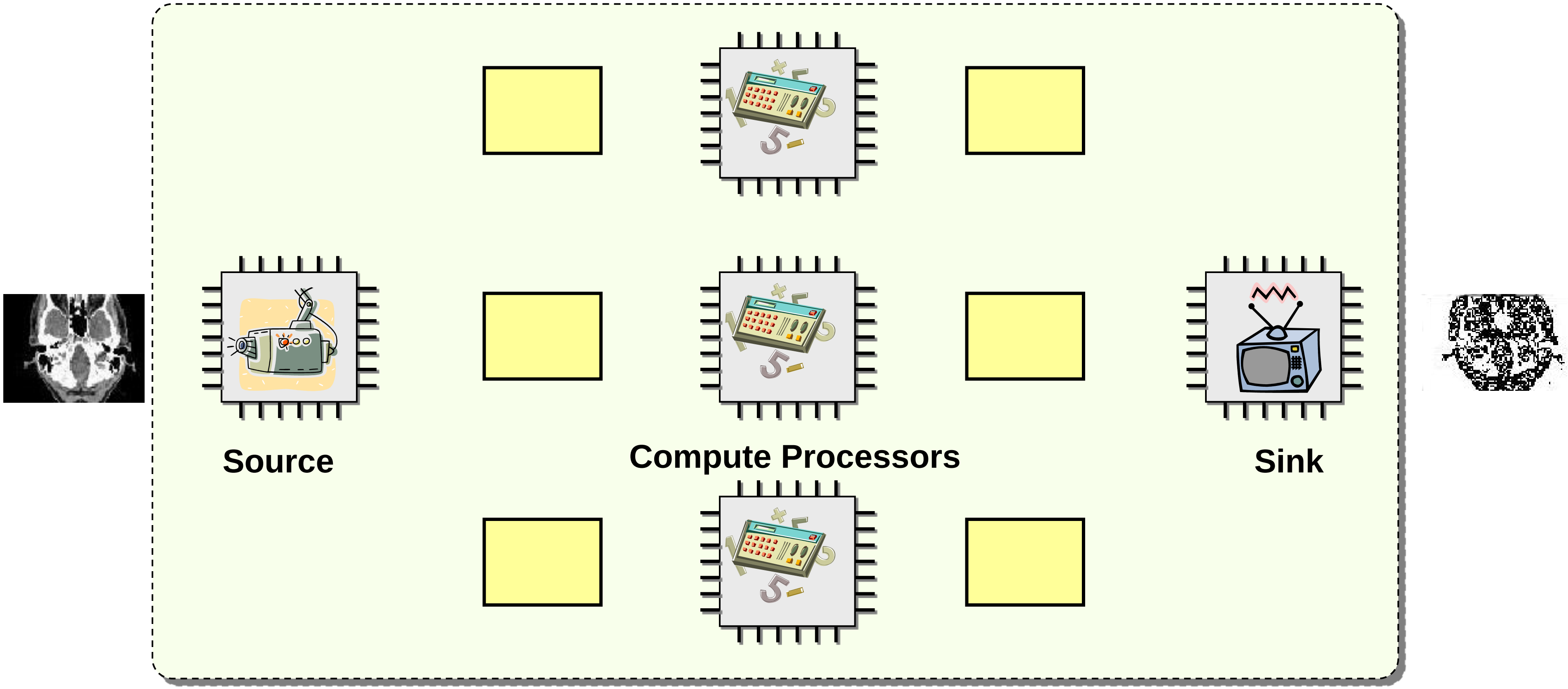


Multi-processor ARM (MPARM) Framework



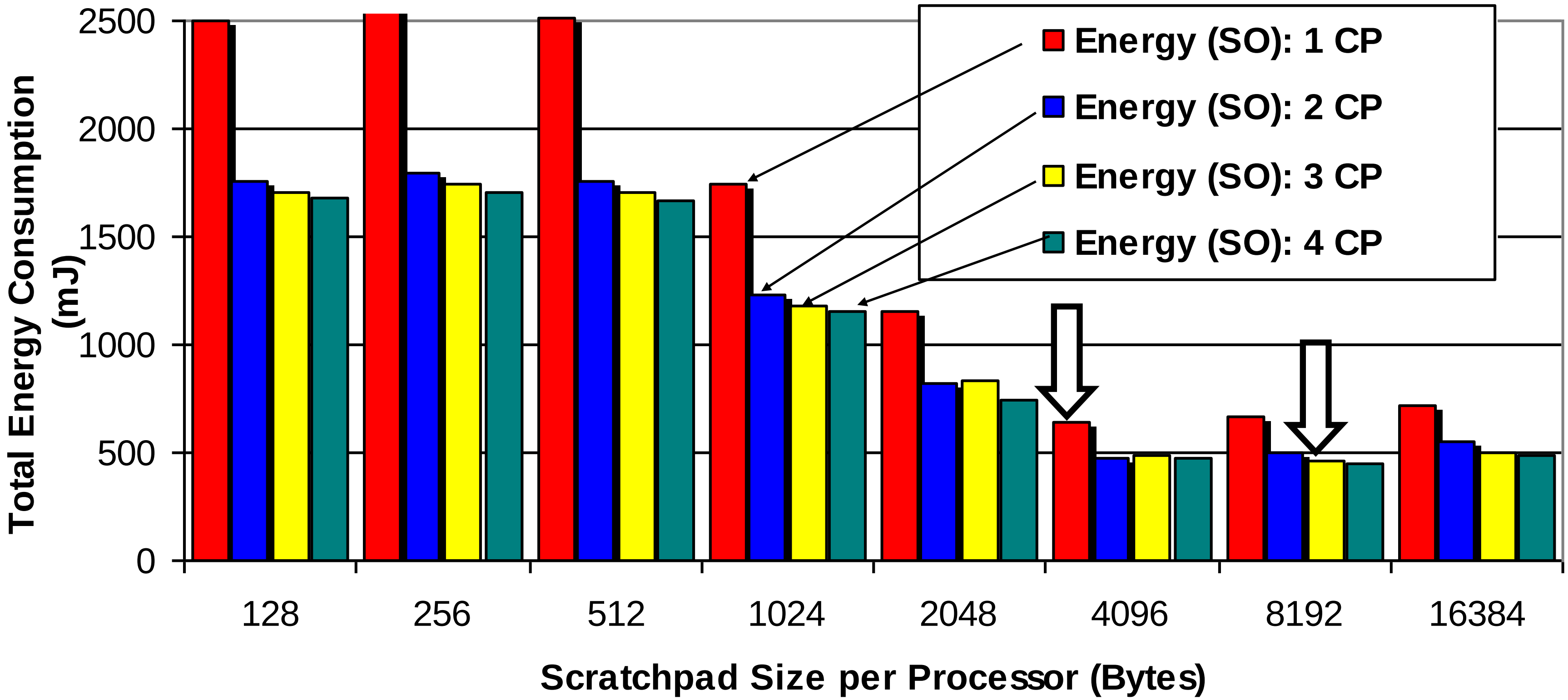
- Homogenous SMP ~ CELL processor
- Processing Unit : ARM7T processor
- Shared Coherent Main Memory
- Private Memory: Scratchpad Memory

Application Example:
Multi-Processor Edge Detection



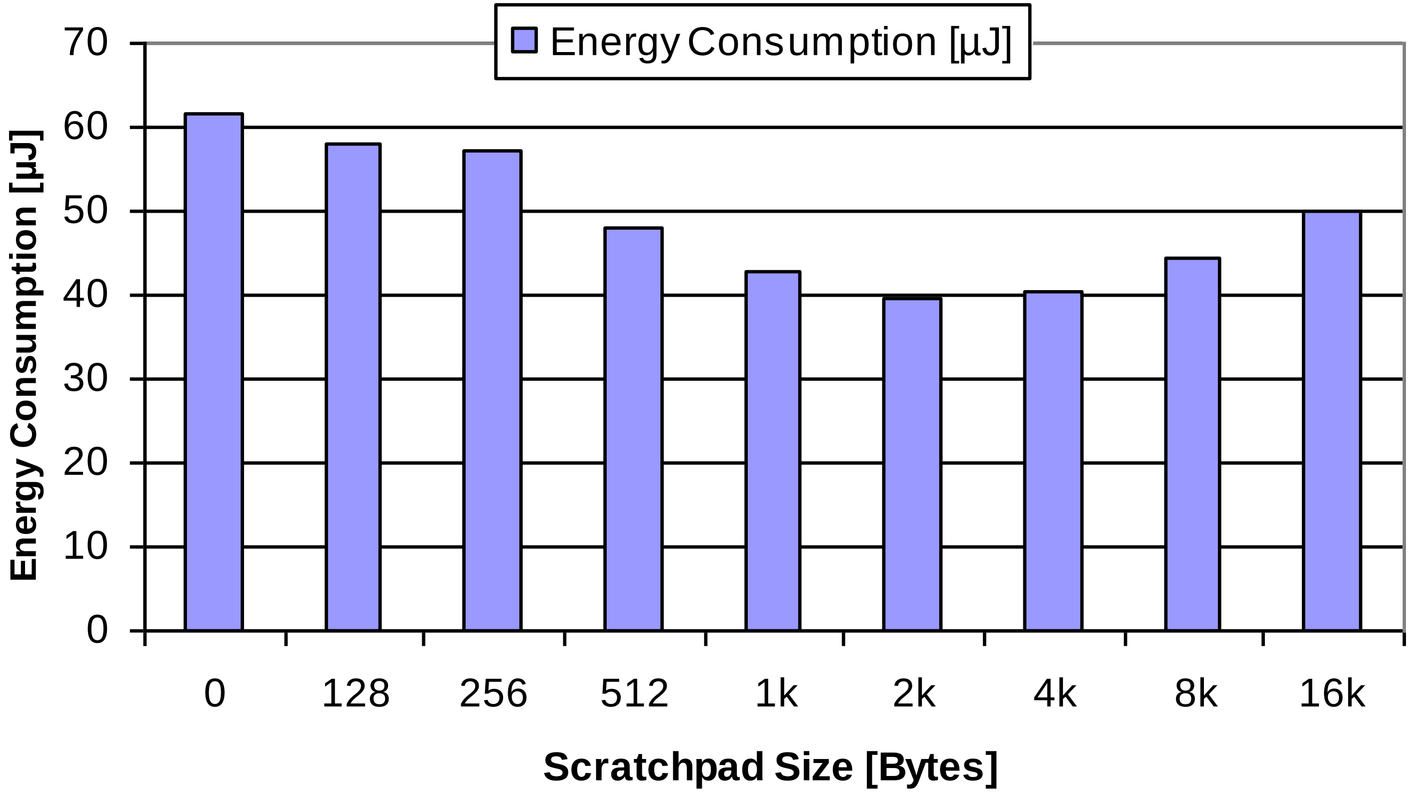
- Source, sink and n compute processors
- Each image is processed by an independent compute processor
  - Communication overhead is minimized.

Results:
Scratchpad Overlay for Edge Detection



- 2 CPs are better than 1 CP, then energy consumption stabilizes
- Best scratchpad size: 4kB (1CP& 2CP) 8kB (3CP & 4CP)

Results
DES-Encryption



DES-Encryption: 4 processors: 2 Controllers+2 Compute Engines

Energy values from ST Microelectronics

Result of cooperation between U. Bologna and TU Dortmund supported by ARTIST2 network of excellence.

Compilers for embedded systems

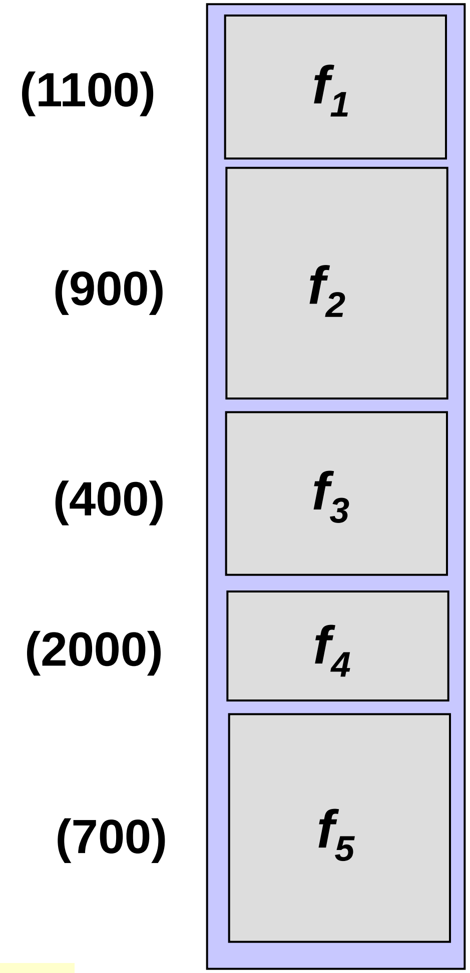
Book section 7.3

- Introduction
- Energy-aware compilation
- Memory-architecture-aware compilation
- Reconciling compilers and timing analysis
- Compilation for digital signal processors
- Compilation for VLIW processors
- Compiler generation, retargetable compilers, design space exploration

Code Layout Transformations (1)

Execution counts based approach:

- Sort the functions according to execution counts
  $f_4 > f_1 > f_2 > f_5 > f_3$
- Place functions in decreasing order of execution counts



[S. McFarling: Program optimization for instruction caches, 3*rd International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS),* 1989]

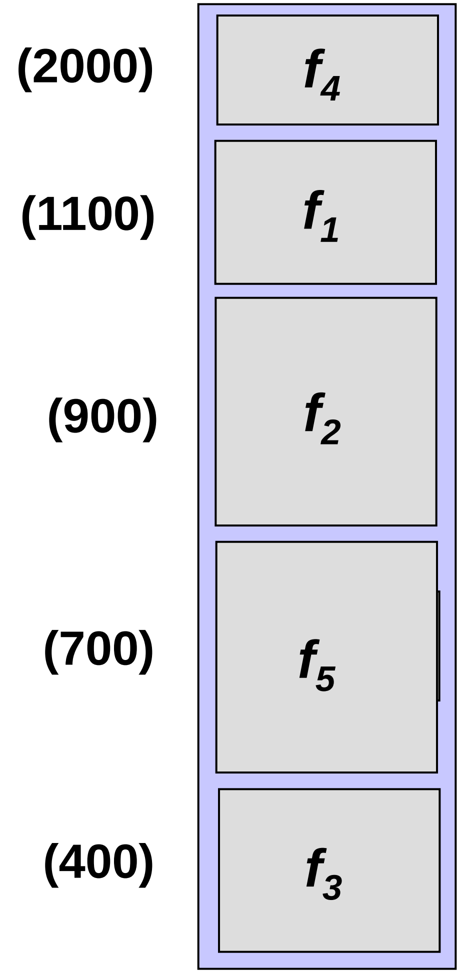
Code Layout Transformations (2)

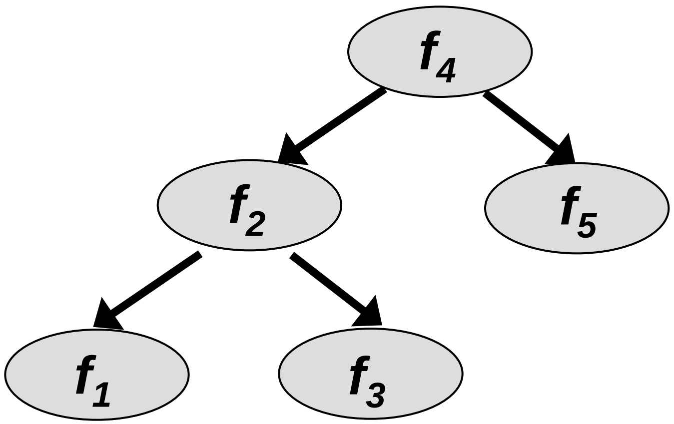
Execution counts based approach:

- Sort the functions according to execution counts
  $f_4 > f_1 > f_2 > f_5 > f_3$
- Place functions in decreasing order of execution counts

Transformation increases spatial locality.

Does not take in account calling order





Code Layout Transformations (3)

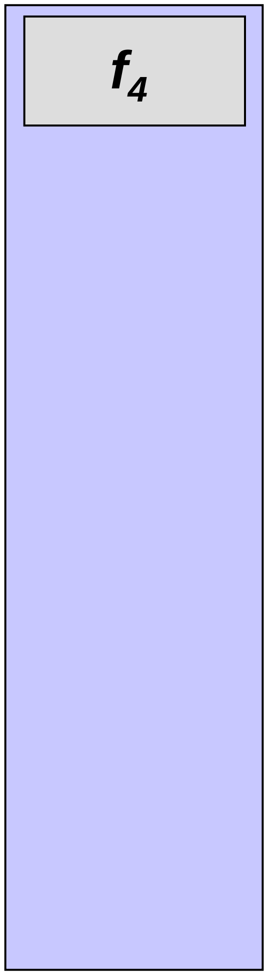
Call-Graph Based Algorithm:

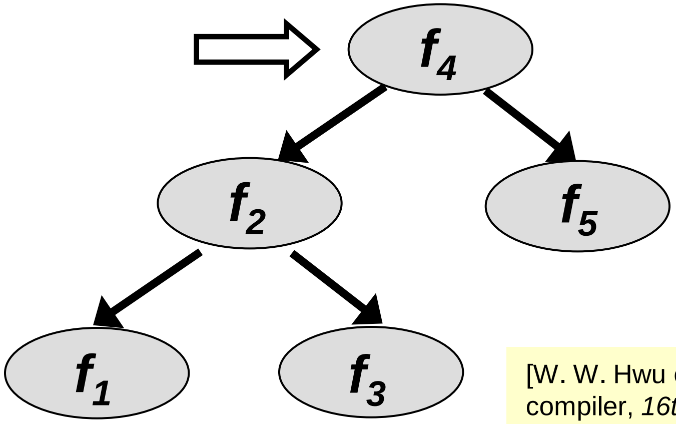
- Create weighted call-graph.
- Place functions according to weighted depth-first traversal.

  $f_4 > f_2 > f_1 > f_3 > f_5$

Increases spatial locality.

(2000)





[W. W. Hwu et al.: Achieving high instruction cache performance with an optimizing compiler, *16th Annual International Symposium on Computer Architecture,* 1989]

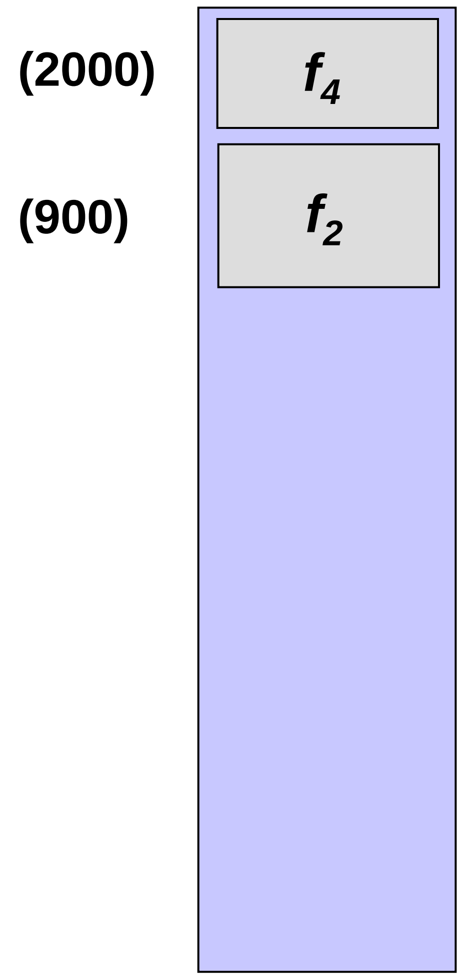
Code Layout Transformations (3)

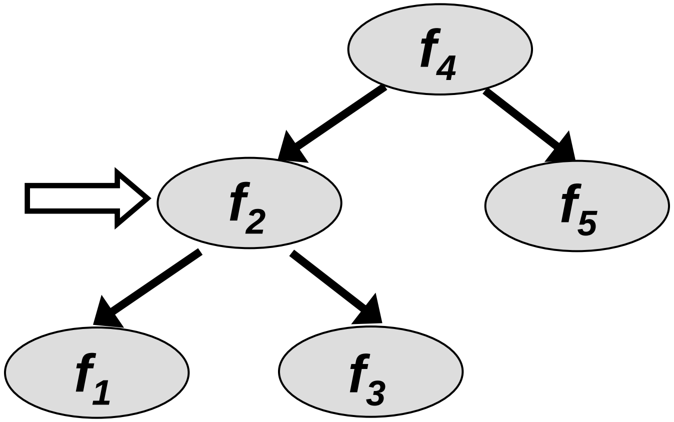
Call-Graph Based Algorithm:

- Create weighted call-graph.
- Place functions according to weighted depth-first traversal.

  $f_4 > f_2 > f_1 > f_3 > f_5$

Increases spatial locality.





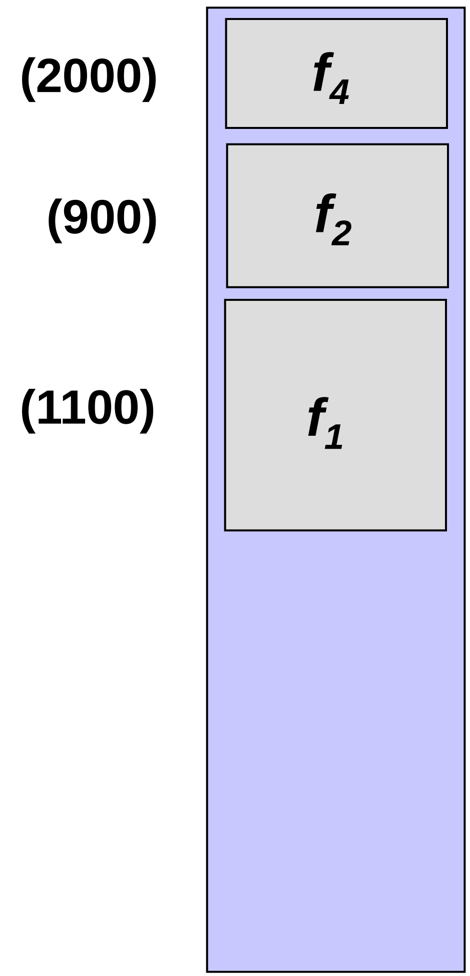
Code Layout Transformations (4)

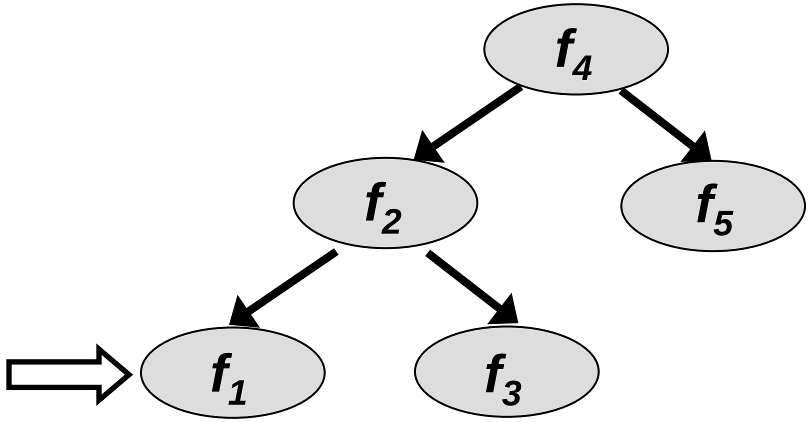
Call-Graph Based Algorithm:

- Create weighted call-graph.
- Place functions according to weighted depth-first traversal.

  $f_4 > f_2 > f_1 > f_3 > f_5$

Increases spatial locality.





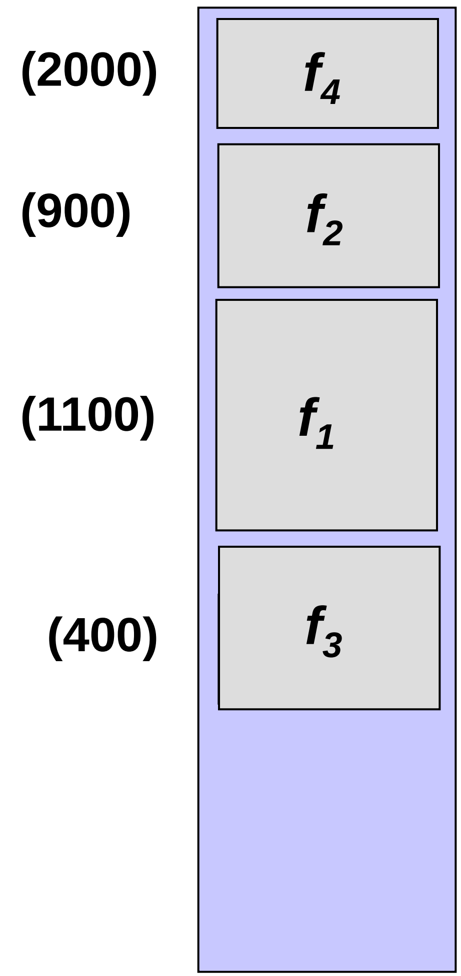
Code Layout Transformations (5)

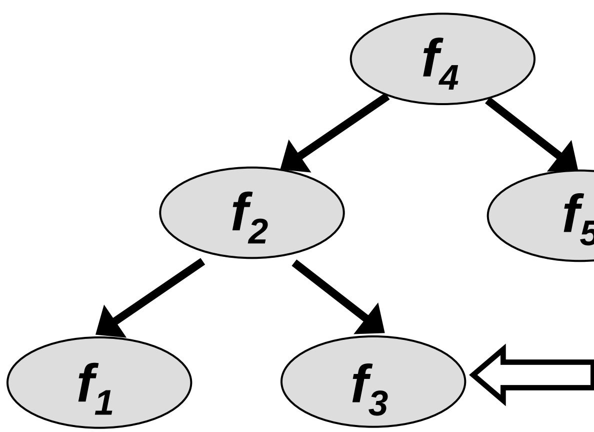
Call-Graph Based Algorithm:

- Create weighted call-graph.
- Place functions according to weighted depth-first traversal.

  $f_4 > f_2 > f_1 > f_3 > f_5$

Increases spatial locality.





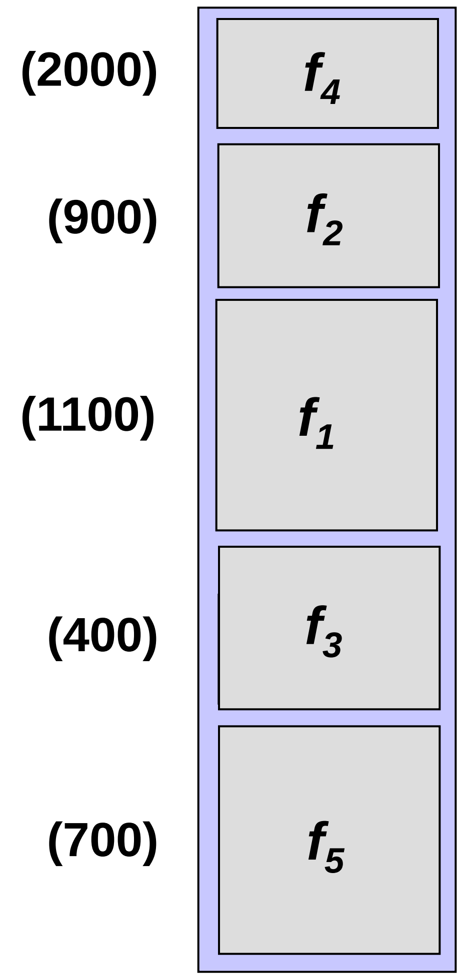
Code Layout Transformations (6)

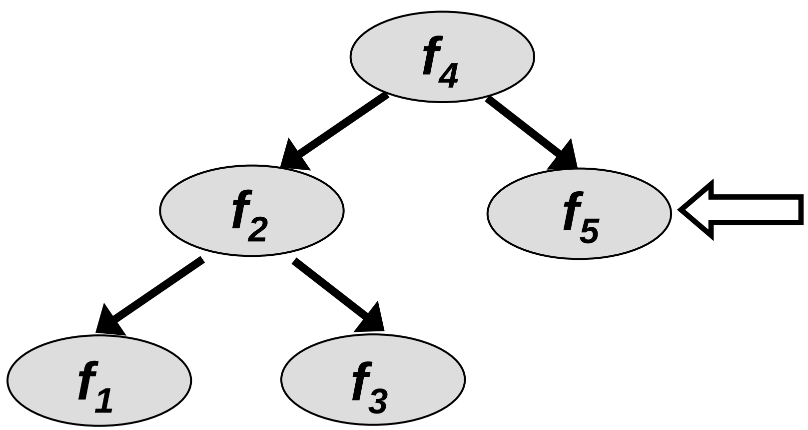
Call-Graph Based Algorithm:

- Create weighted call-graph.
- Place functions according to weighted depth-first traversal.

  $f_4 > f_2 > f_1 > f_3 > f_5$
- Combined with placing frequently executed traces at the top of the code space for functions.

Increases spatial locality.



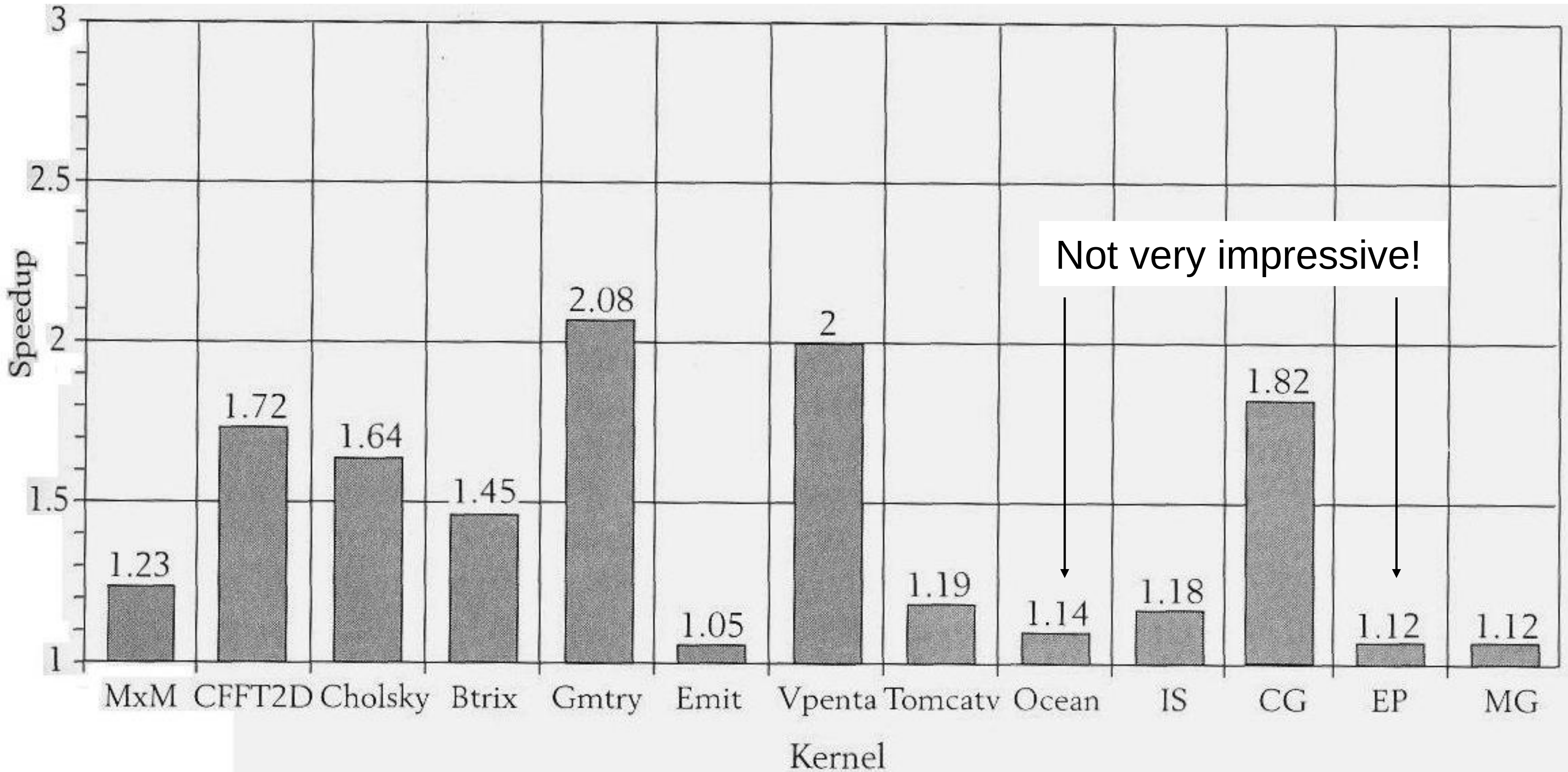


Prefetching

- 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!)

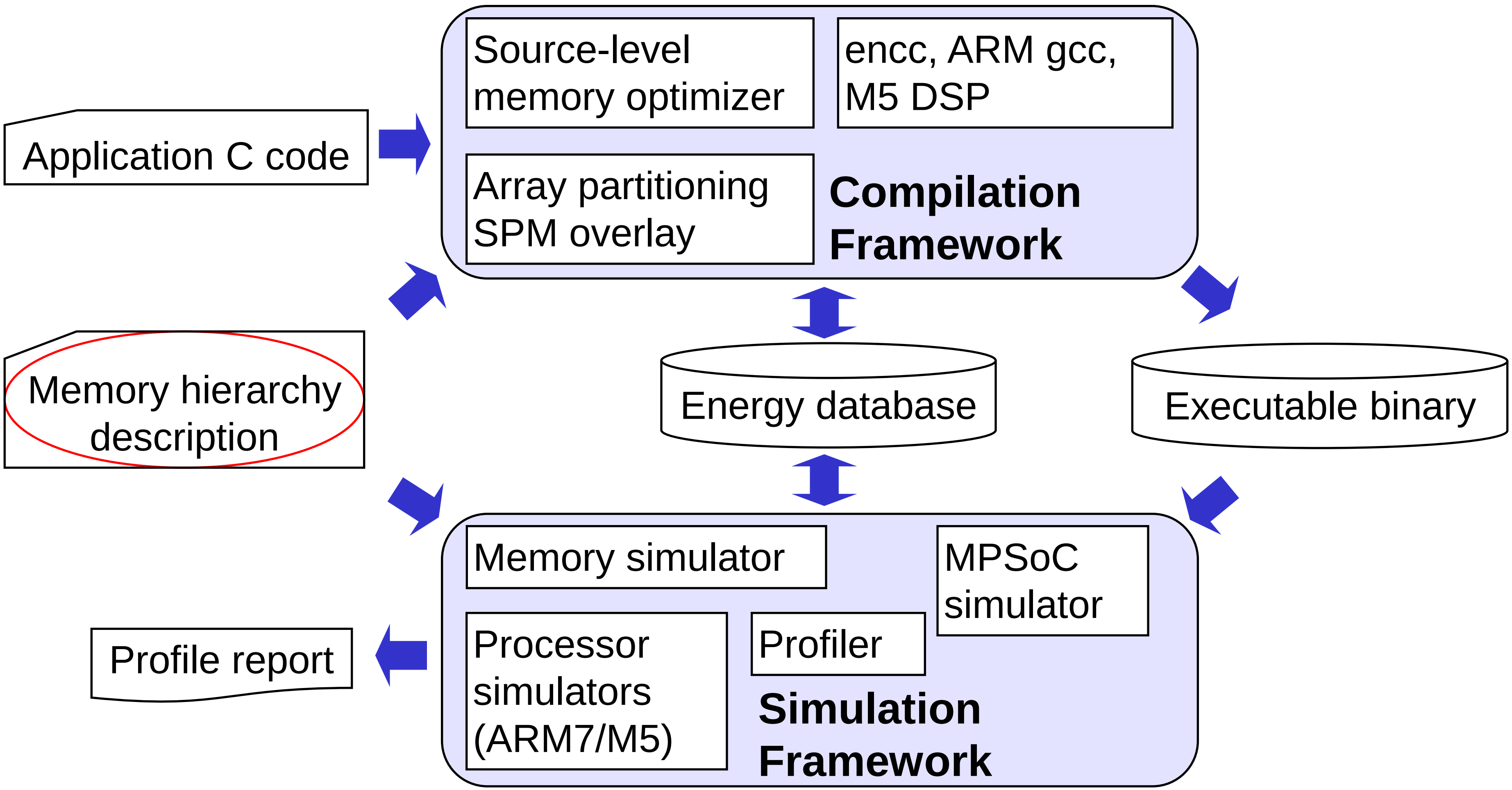
R. Allen, K. Kennedy: Optimizing Compilers for Modern Architectures, *Morgan-Kaufman*, 2002

Results for prefetching



Mowry, as cited by R. Allen & K. Kennedy

Memory Aware Compilation and Simulation Framework (for C) MACC

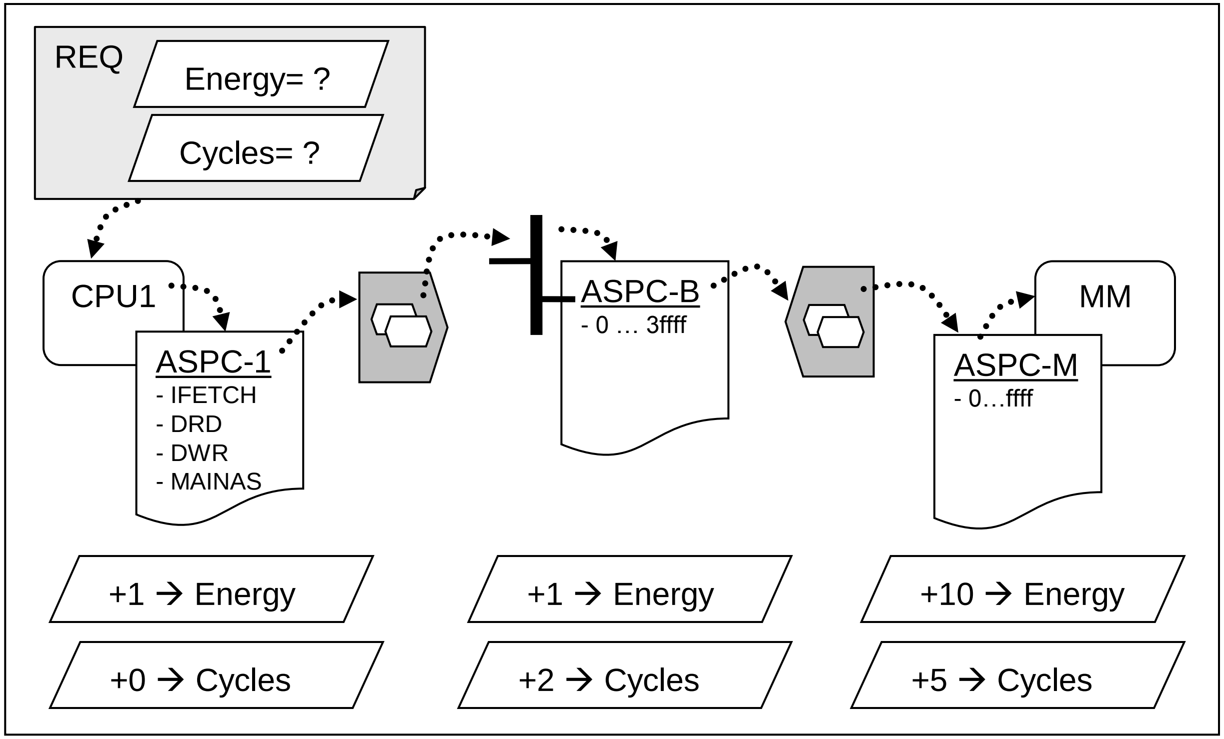


M. Verma, L. Wehmeyer, R. Pyka, P. Marwedel, L. Benini: Compilation and Simulation Tool Chain for Memory Aware Energy Optimizations, *Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS VI)*, 2006

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

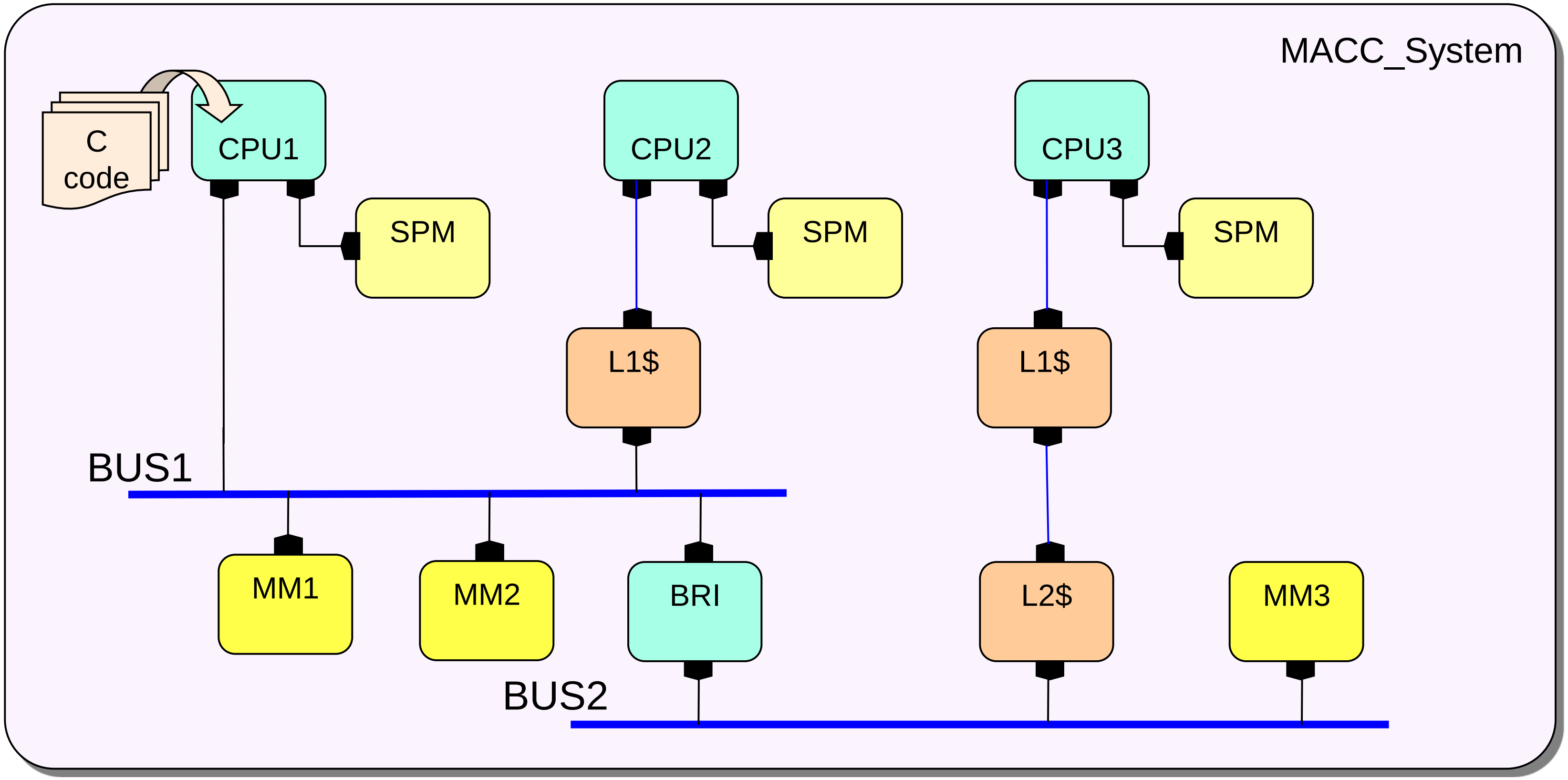
- API query to model simplifies integration into compiler
- External XML representation



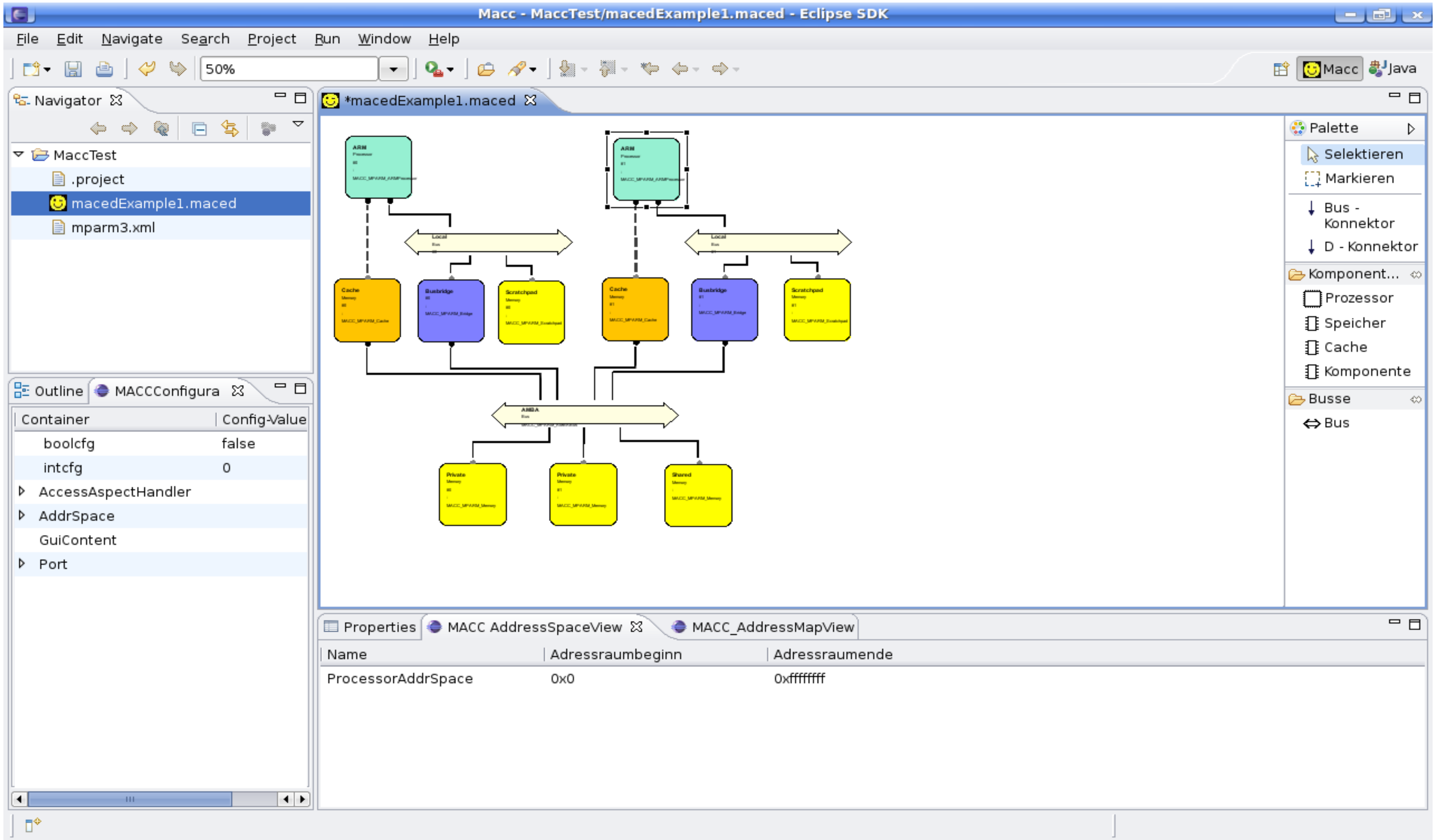
R. Pyka et al.: Versatile System level Memory Description Approach for embedded MPSoCs, *University of Dortmund, Informatik 12*, 2007

Introduction of Memory Architecture-Aware Optimization
The MACC PMS (Processor/ Memory/Switch) Model

- Explicit memory architecture
- API provides access to memory information



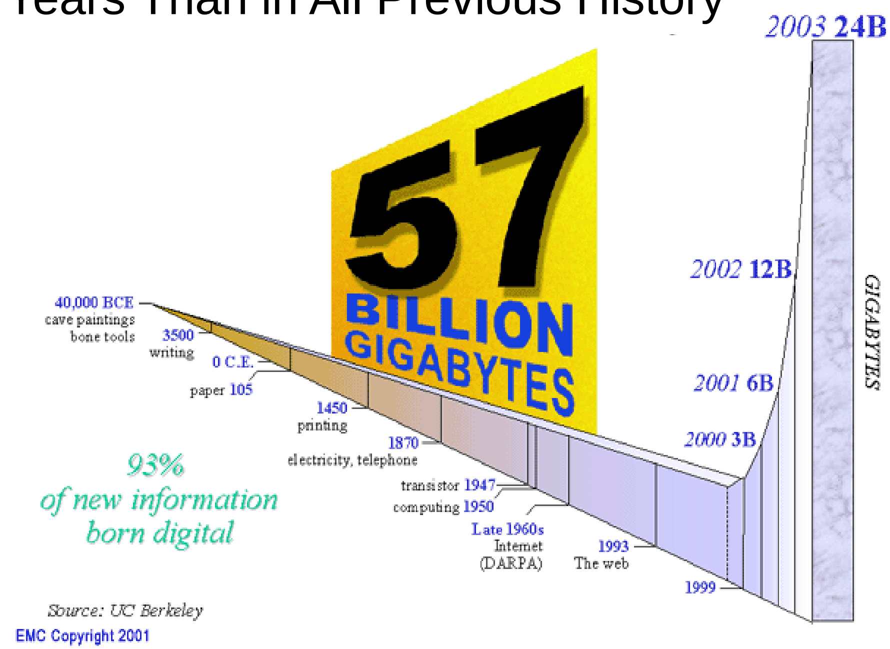
MaCC Modeling Example via GUI



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

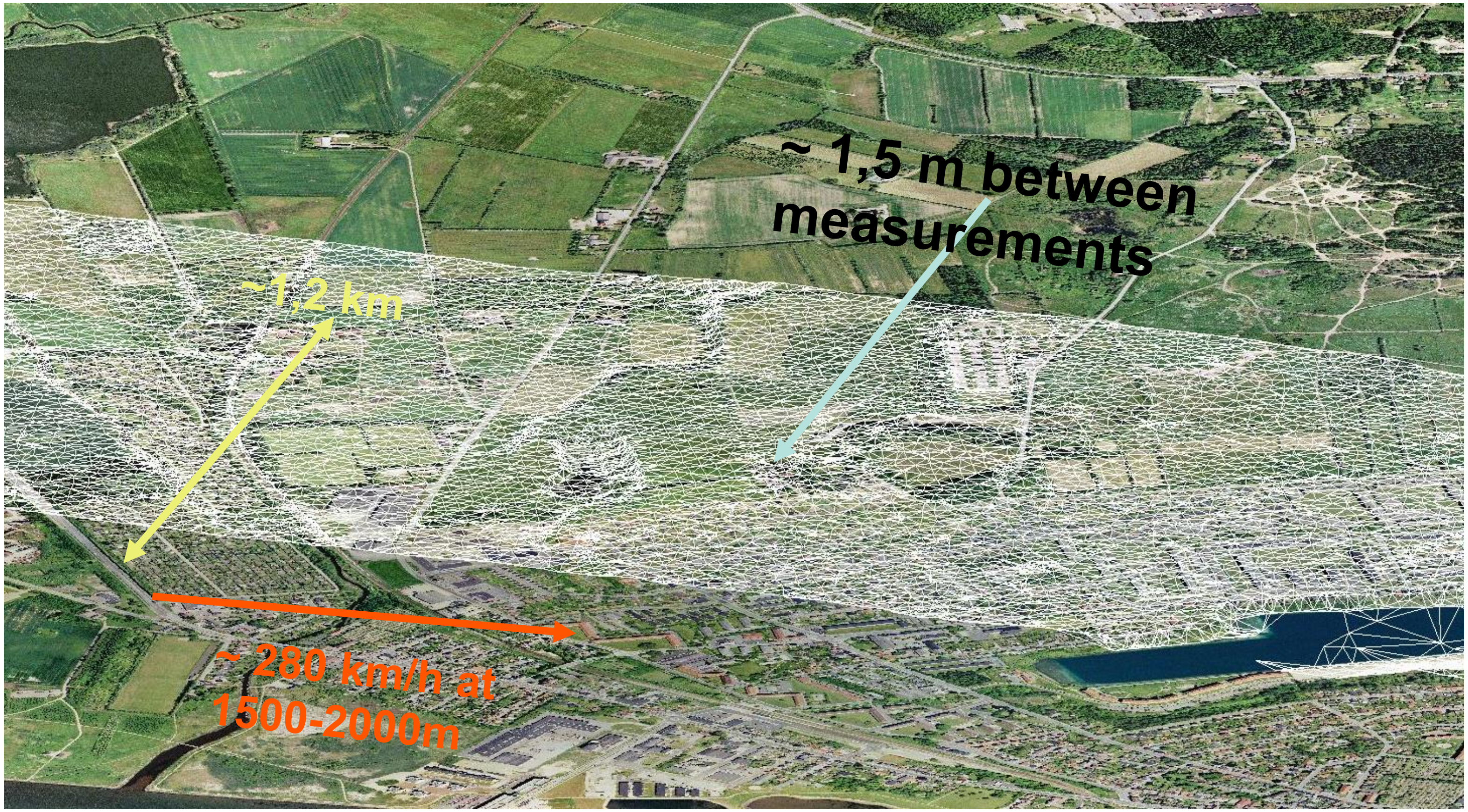
WEB: Web crawl of 200M pages and 2000M links, Akamai stores 7 billion clicks per day

Geography: NASA satellites generate 1.2TB per day

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

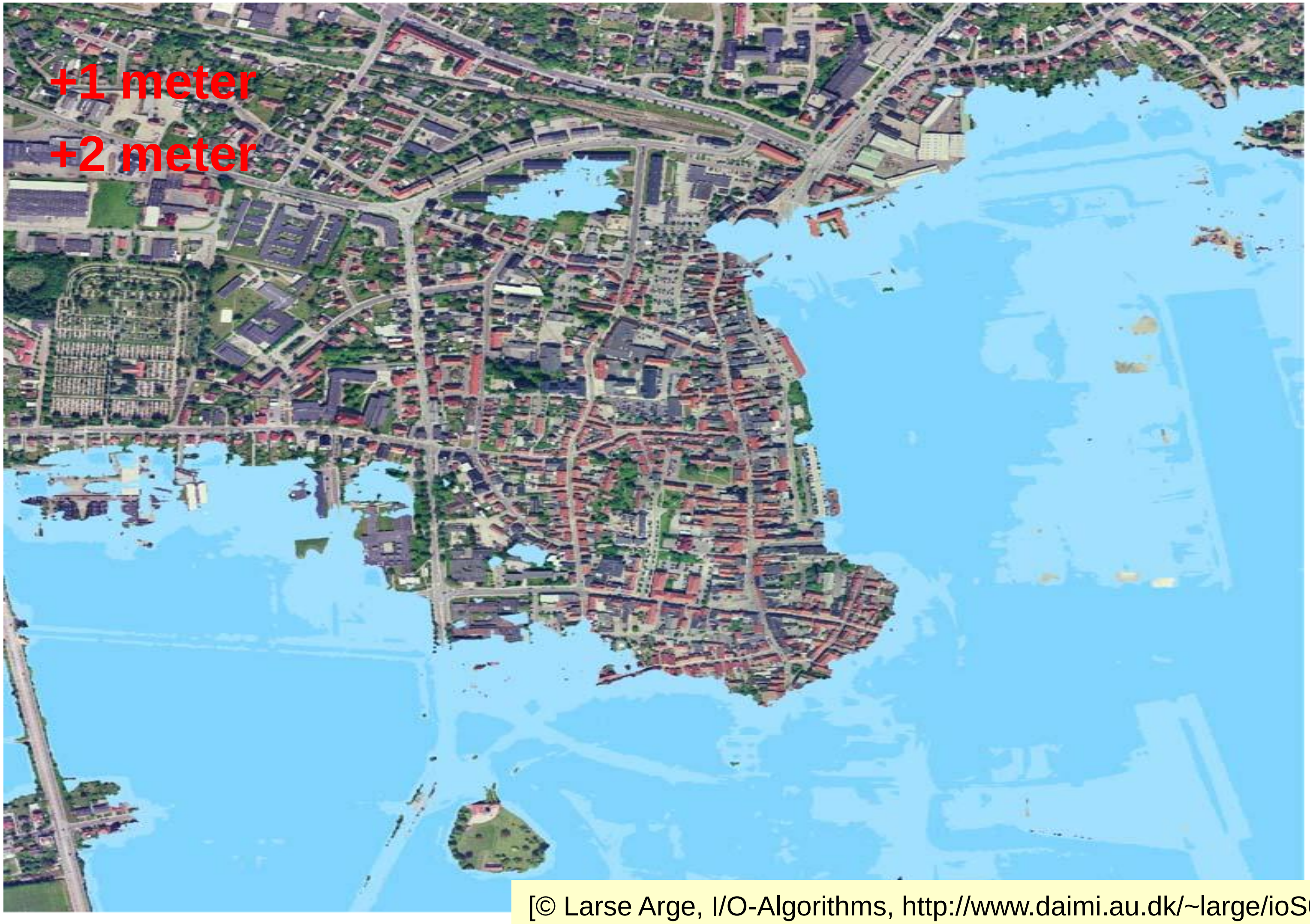
Example: LIDAR Terrain Data

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/]

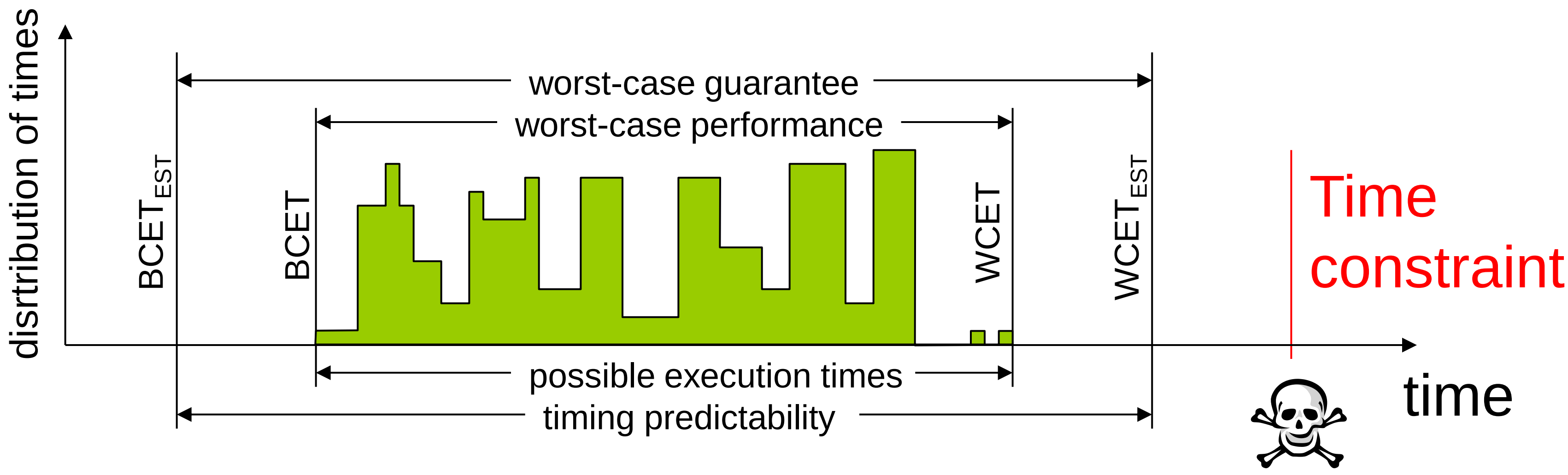
Compilers for embedded systems

Book section 7.3

- Introduction
- Energy-aware compilation
- Memory-architecture-aware compilation
- ➡ Reconciling compilers and timing analysis
- Compilation for digital signal processors
- Compilation for VLIW processors
- Compiler generation, retargetable compilers, design space exploration

Worst case execution time (1)

Definition of worst case execution time:



$WCET_{EST}$ must be

1. safe (i.e. ≥ WCET) and
2. tight ($WCET_{EST}$-WCET≪$WCET_{EST}$)

© Graphics: adopted from R. Wilhelm + Microsoft Cliparts

Current Trial-and-Error Based Development

1. Specification of CPS/ES system
2. Generation of Code (ANSI-C or similar)
3. Compilation of Code
4. Execution and/or simulation of code, using a (e.g. random) set of input data
5. Measurement-based computation of "estimated worst case execution time" ($WCET_{meas}$)
6. Adding safety margin m on top of $WCET_{meas}$: $WCET_{hypo} := (1 + m) * WCET_{meas}$
7. If "$WCET_{hypo}$" > deadline: change some detail, go back to 1 or 2.

Problems with this Approach

Dependability

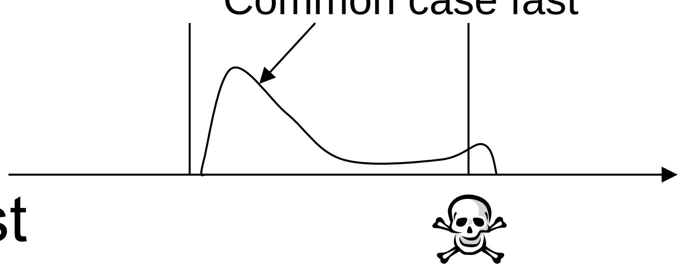
- Computed "WCET$_{hypo}$" not a safe approximation
- ☞ Time constraint may be violated

Design time

- How to find necessary changes?
- How many iterations until successful?

"Make the common case fast" a wrong approach for RT-systems

- Computer architecture and compiler techniques focus on average speed
- Circuit designers know it's wrong
- Compiler designers (typically) don't



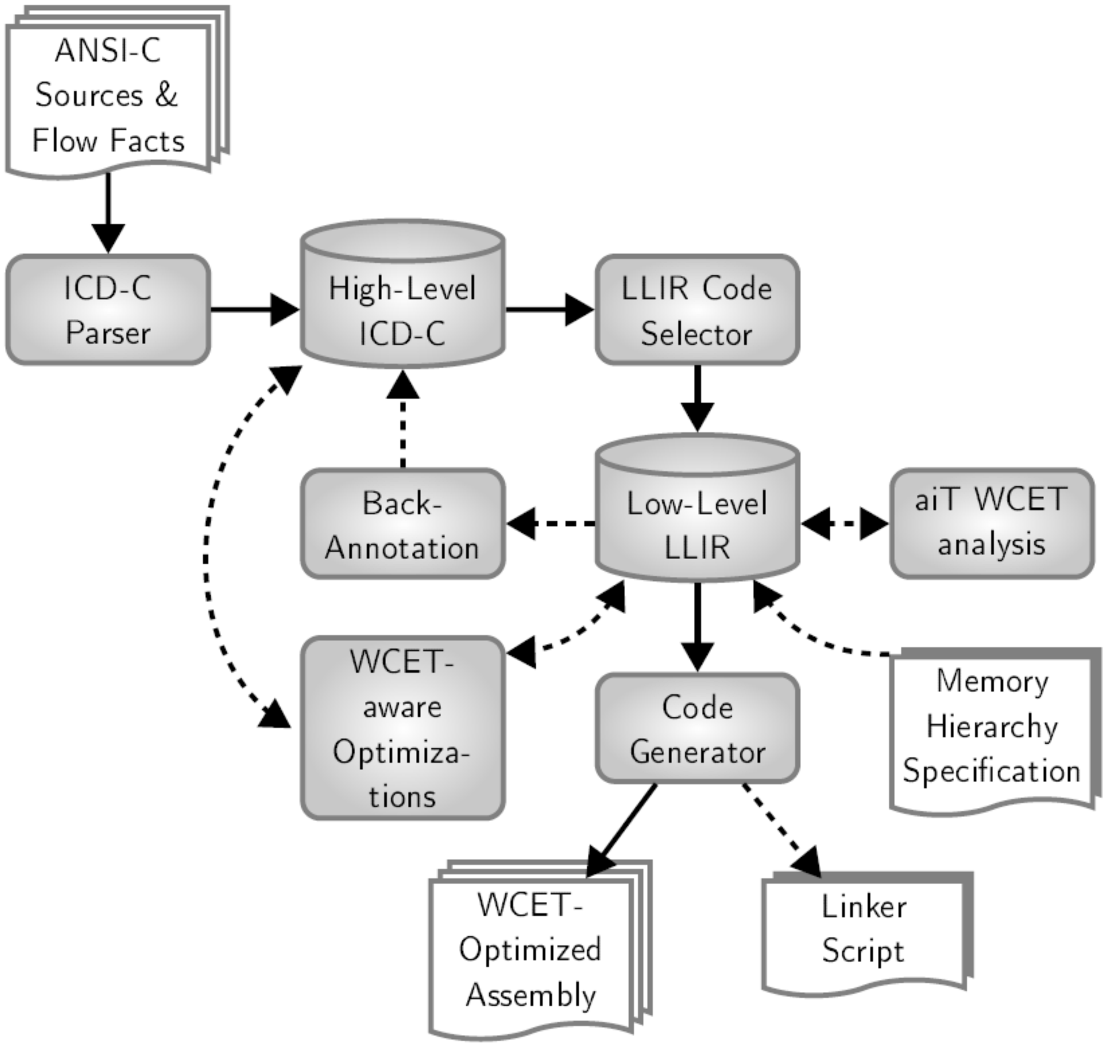
"Optimizing" compilers unaware of cost functions other than code size

Integration of WCET estimation and compilation

Computing $WCET_{EST}$ **after** code generation is too late.

Why not consider $WCET_{EST}$ as an objective function already in the compiler?

☞ Integration of aiT and compiler



Challenges for WCET_{EST}-Minimization

Worst-Case Execution Path (WCEP)

- WCET_{EST} of a program = Length of longest execution path (WCEP) in that program
- WCET_{EST}-Minimization:
  Reduction of the longest path
- Other optimizations do not result in a reduction of WCET_{EST}

☞ Optimizations need to know the WCEP

WCET-oriented optimizations

- Extended loop analysis (CGO 09)
- Instruction cache locking (CODES/ISSS 07, CGO 12)
- Cache partitioning (WCET-Workshop 09)
- Procedure cloning (WCET-WS 07, CODES 07, SCOPES 08)
- Procedure/code positioning (ECRTS 08, CASES 11 (2x))
- Function inlining (SMART 09, SMART 10)
- Loop unswitching/invariant paths (SCOPES 09)
- Loop unrolling (ECRTS 09)
- Register allocation (DAC 09, ECRTS 11))
- Scratchpad optimization (DAC 09)
- Extension towards multi-objective optimization (RTSS 08)
- Superblock-based optimizations (ICESS 10)
- Surveys (Springer 10, Springer 12)

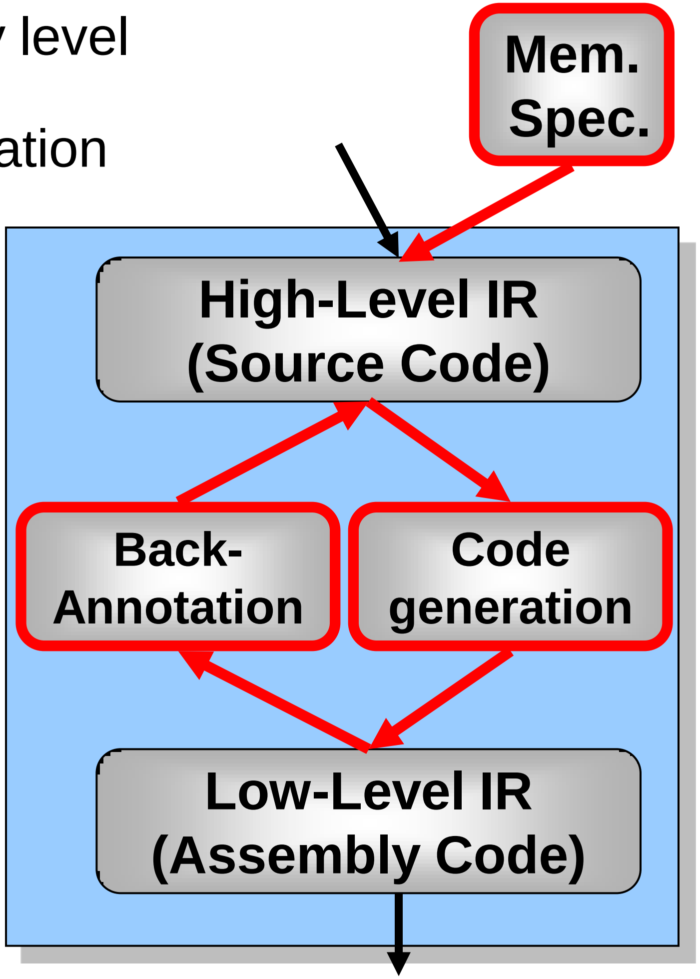
Loop Unrolling as an Example

- Unrolling replaces the original loop with several instances of the loop body
- Positive Effects
  - Reduced overhead for loop control
  - Enables instruction level parallelism (ILP)
  - Offers potential for following optimizations
- Unroll *early* in optimization chain
- Negative Effects
  - Aggressive unrolling leads to I-cache overflows
  - Additional spill code instructions
  - Control code may cancel positive effects

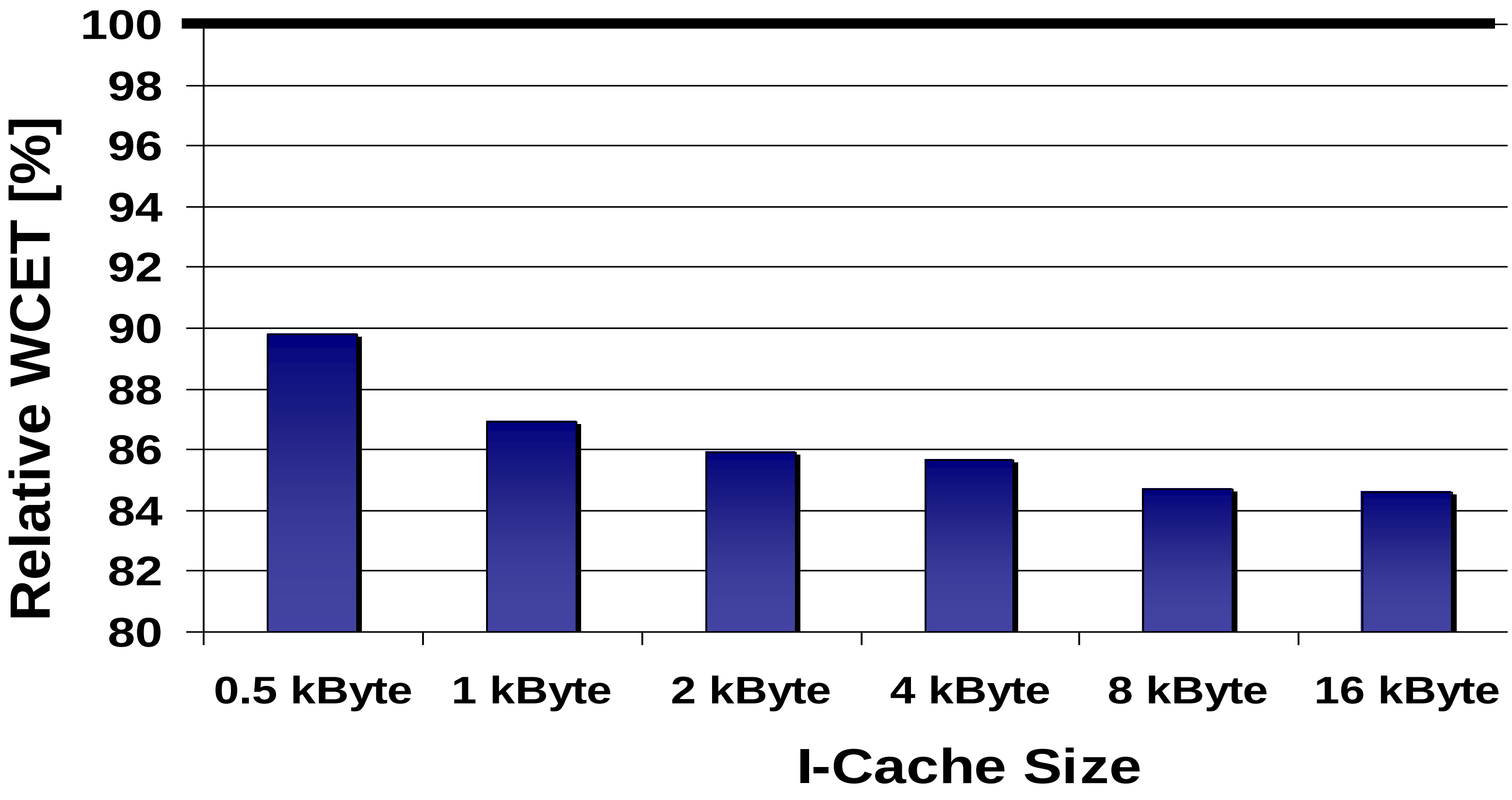
Consequences of transformation hardly known

WCET_{EST}-aware Loop Unrolling via Back-annotation

- WCET_{EST}-information available at assembly level
- Unrolling to be applied at internal representation of source code
- Solution: Back-annotation:
  Experimental WCET_{EST}-aware compiler WCC allows copying information:
  assembly code ☞ source code
  - WCET_{EST} data
  - Assembly code size
  - Amount of spill code
- Memory architecture info available



Results for unrolling: $WCET_{EST}$



100%: Avg. $WCET_{EST}$ for all benchmarks with –O3 & no unrolling

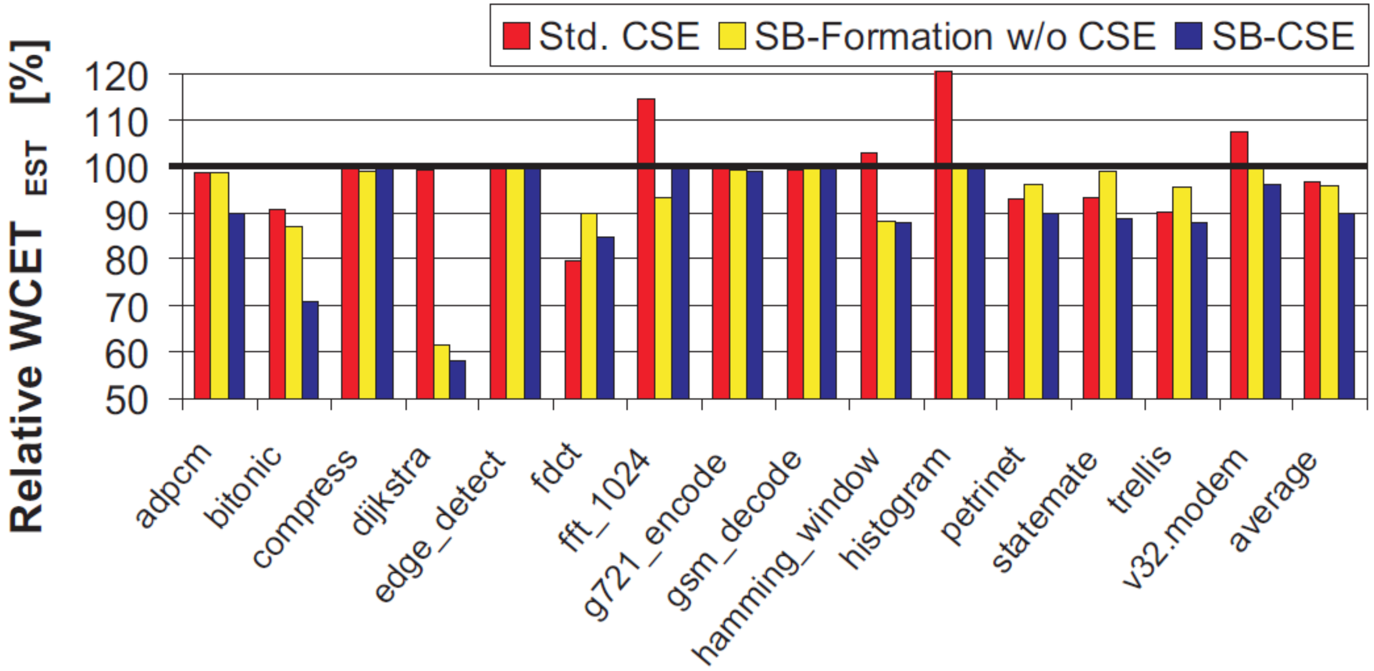
$WCET_{EST}$ reduction between 10.2% and 15.4%

$WCET_{EST}$-driven unrolling outperforms standard unrolling by 13.7%

WCET_{EST}-aware superblock optimizations

WCET-aware superblock optimizations

- ***WCC's superblocks:*** proposed 1st time at ANSI-C code level, rely on WCET_{EST} timing data
- ***WCET_{EST}-aware superblock optimizations:*** Common Subexpression Elimination (CSE) and Dead Code Elimination (DCE) ported to WCC

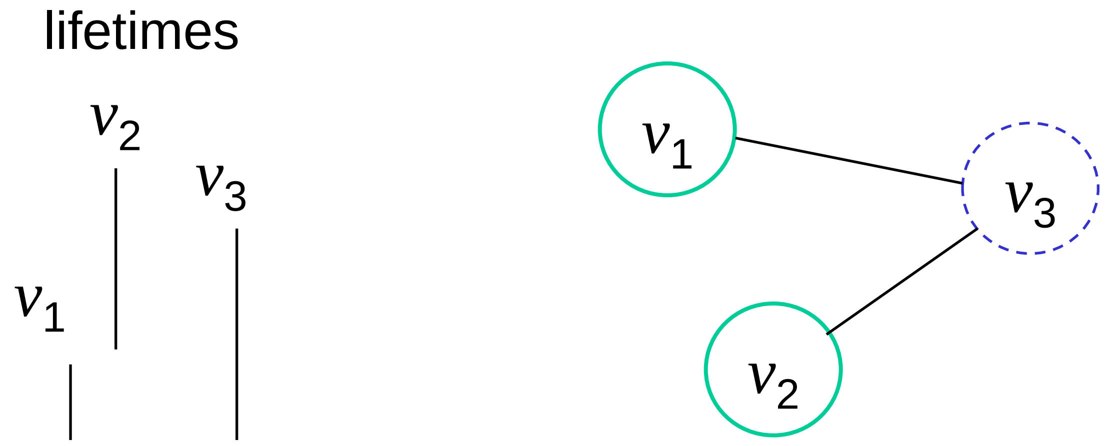


Results: Register allocation

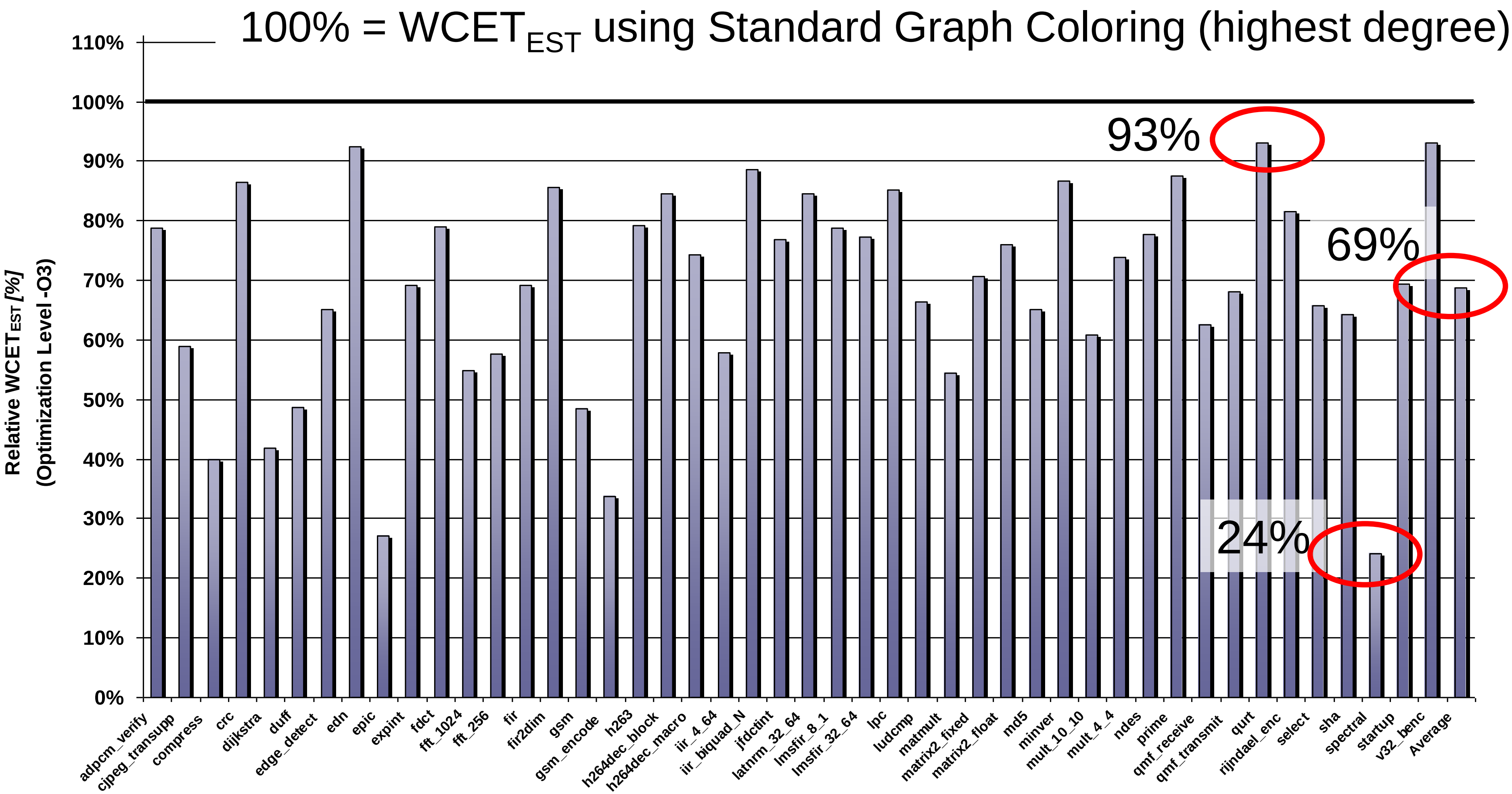
- Registers = fastest level in the memory hierarchy

☞ Interest in good global register allocation techniques

- Frequently based on coloring of interference graph



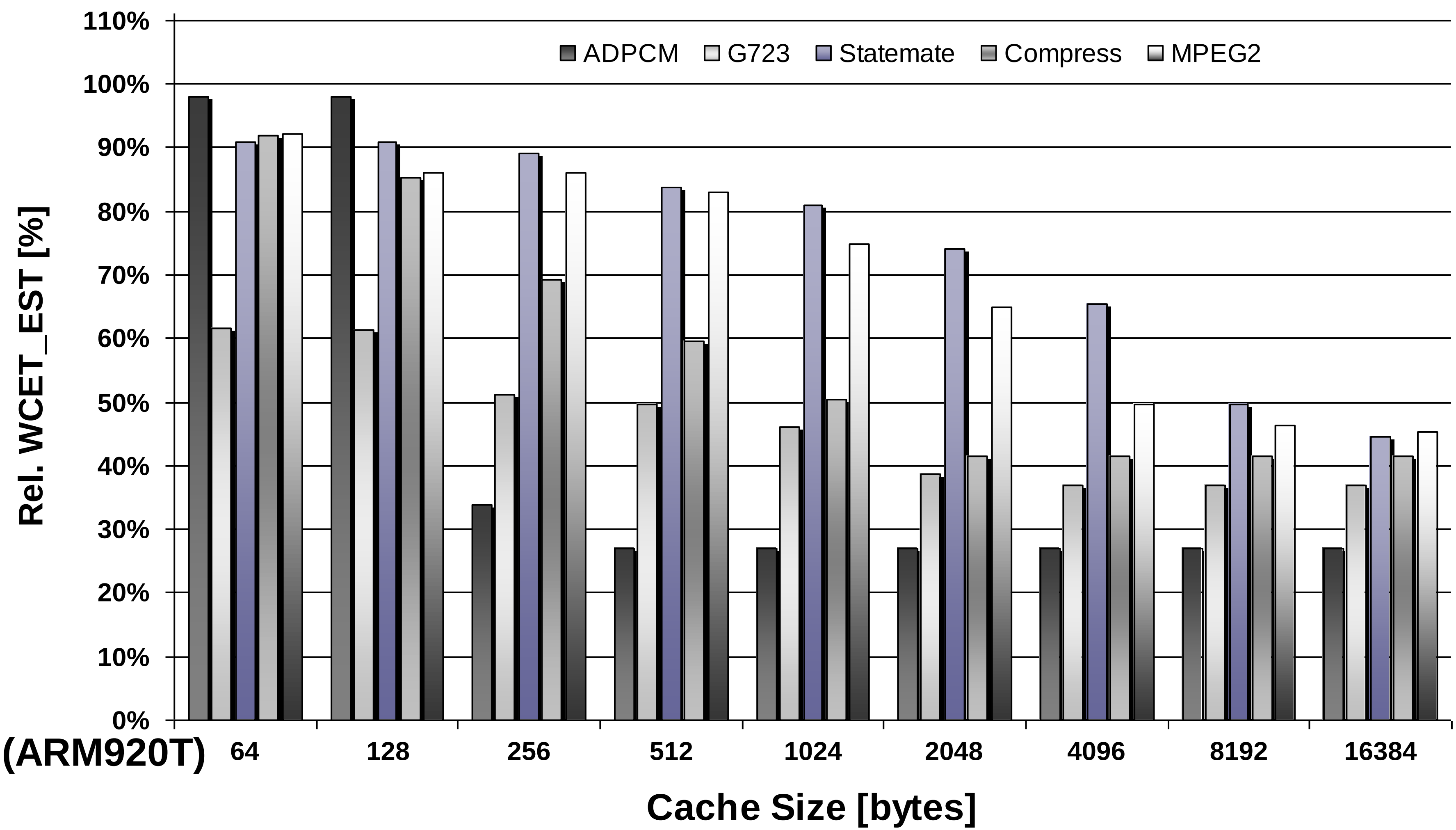
Register Allocation



Improving predictability for caches

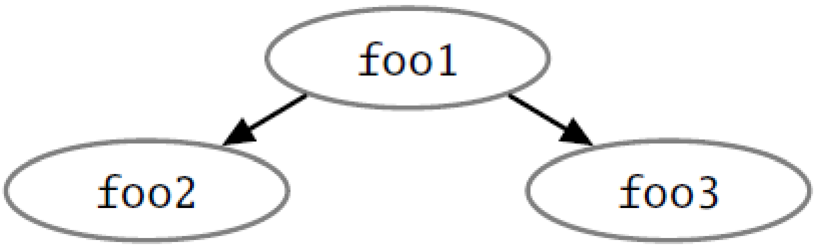
- Loop caches
- Mapping code to less used part(s) of the index space
- Cache locking/freezing
- Changing the memory allocation for code or data
- Mapping pieces of software to specific ways
  Methods:
  - Way prediction in hardware
  - Generating appropriate way in software
  - Allocation of certain parts of the address space to a specific way
  - Including way-identifiers in virtual to real-address translation

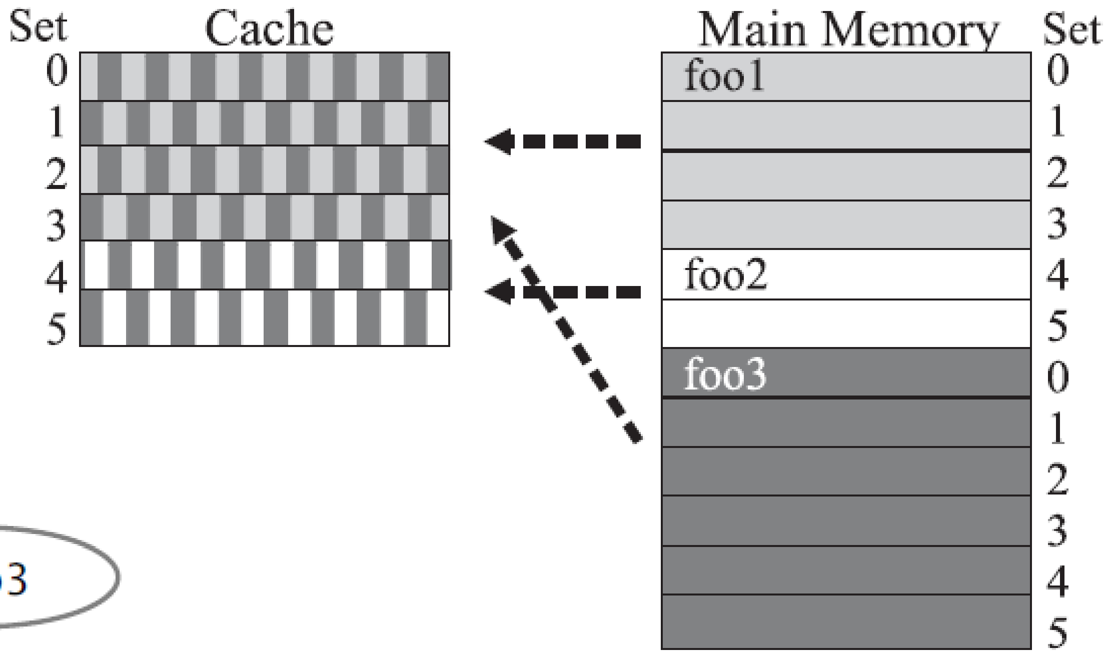
Relative $WCET_{EST}$ with I-*Cache Locking*
5 Benchmarks/ARM920T/Postpass-Opt



Potential cache thrashing

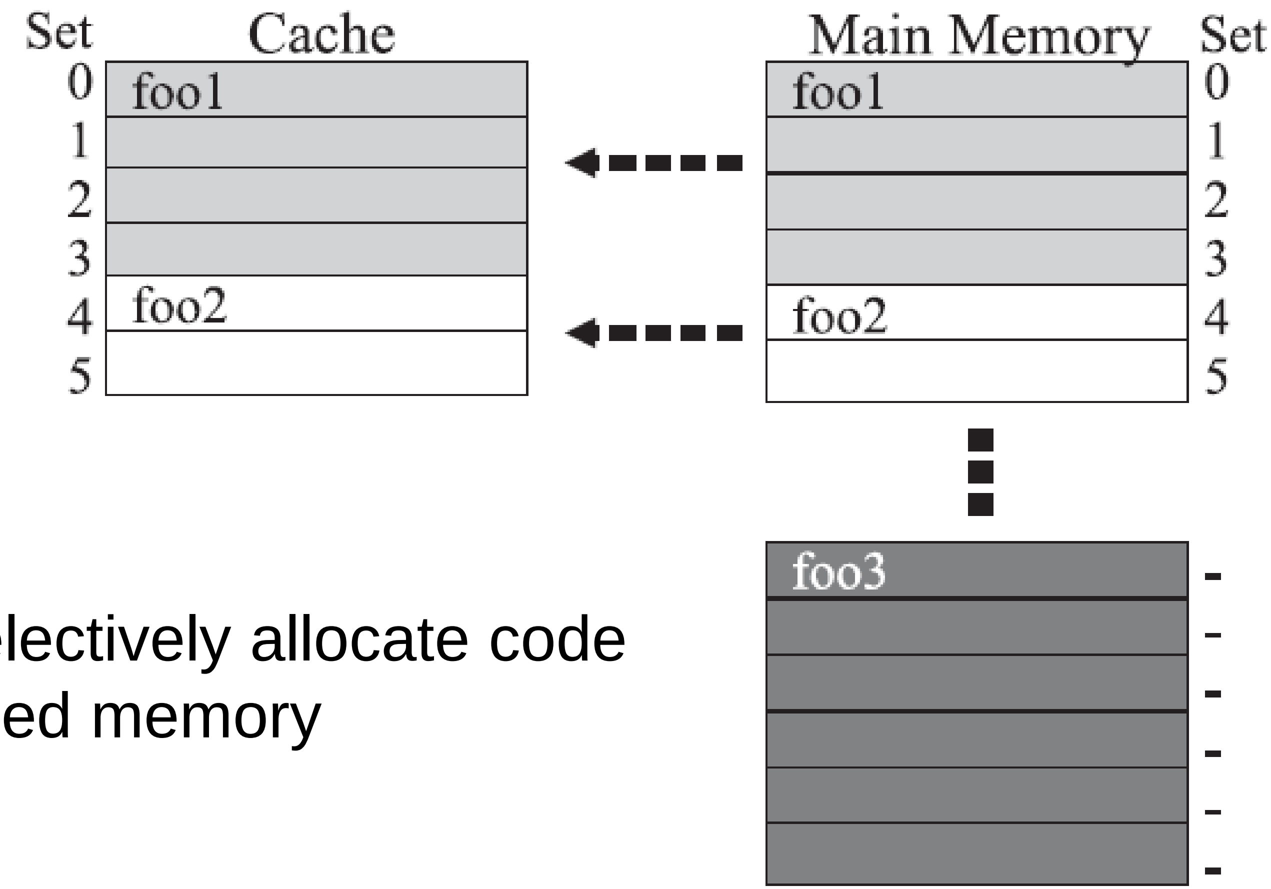
```
void foo1() {
 for(i=0; i<10; i++) {
  foo2();
  foo3();
 }
 ...
}
```





[S. Plazar: Memory-based Optimization Techniques for Real-Time Systems, PhD thesis, TU Dortmund, June 2012]

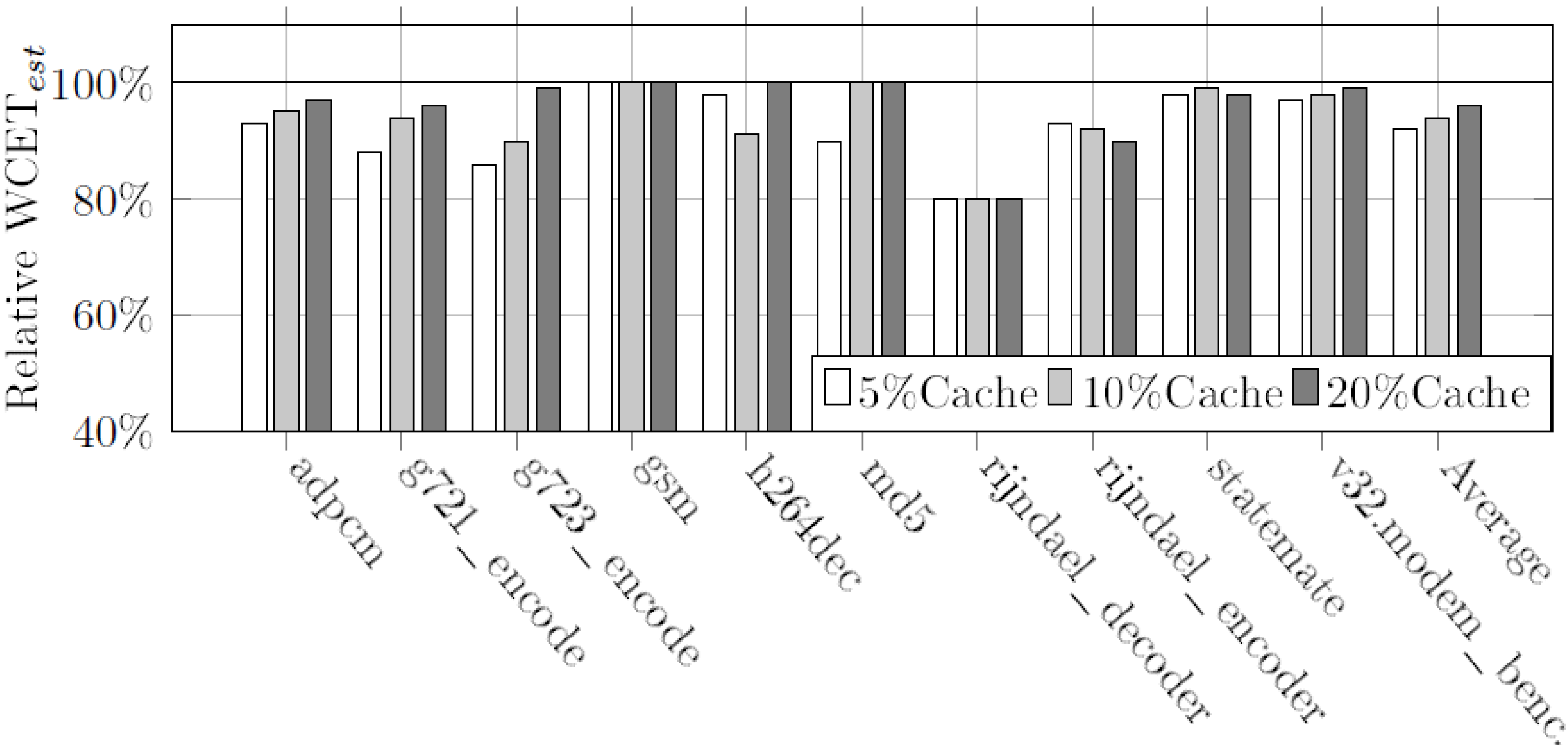
Avoiding cache thrashing



Key idea: selectively allocate code to uncached memory

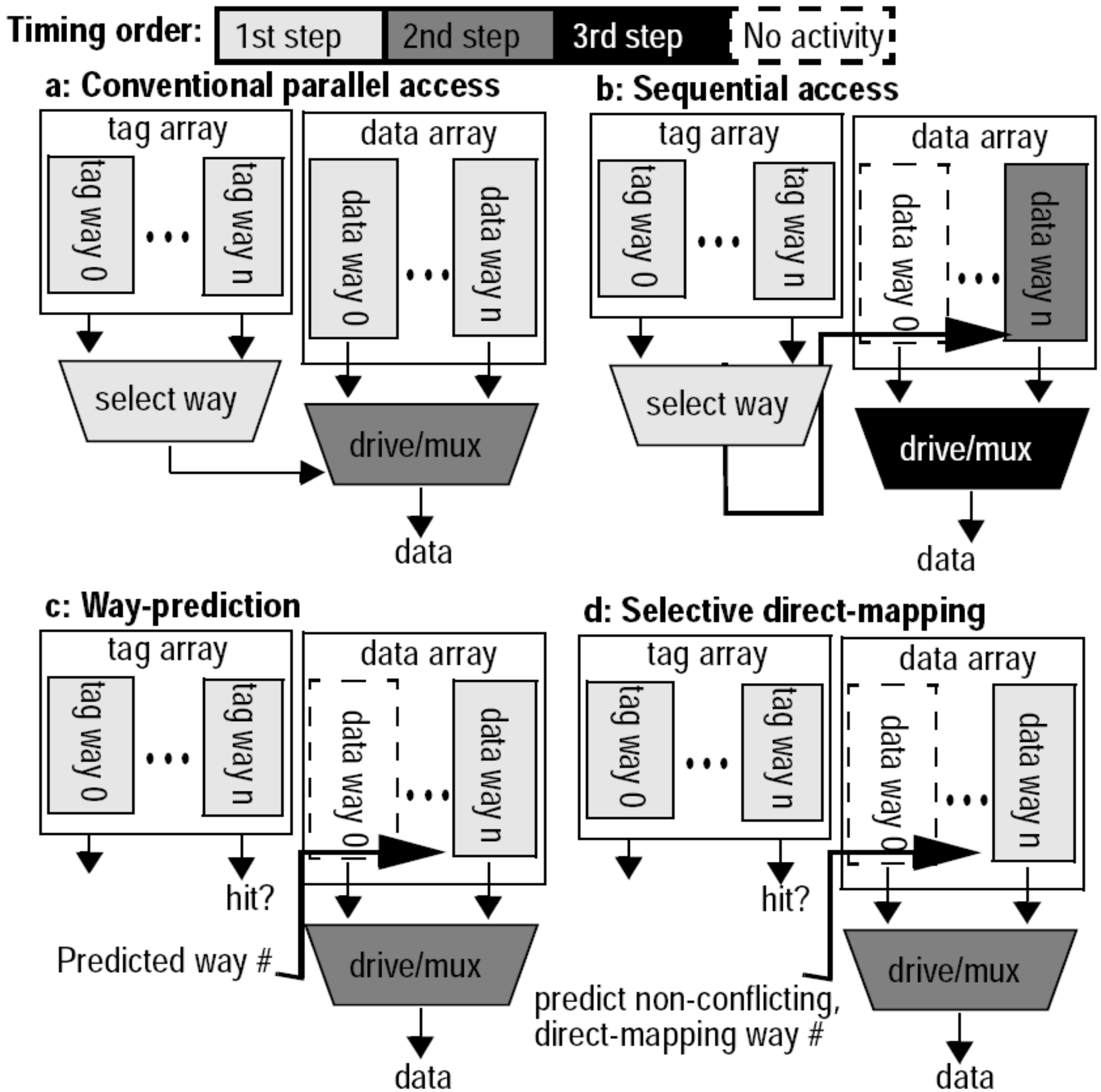
S. Plazar: Memory-based Optimization Techniques for Real-Time Systems, PhD thesis, TU Dortmund, June 2012

Results



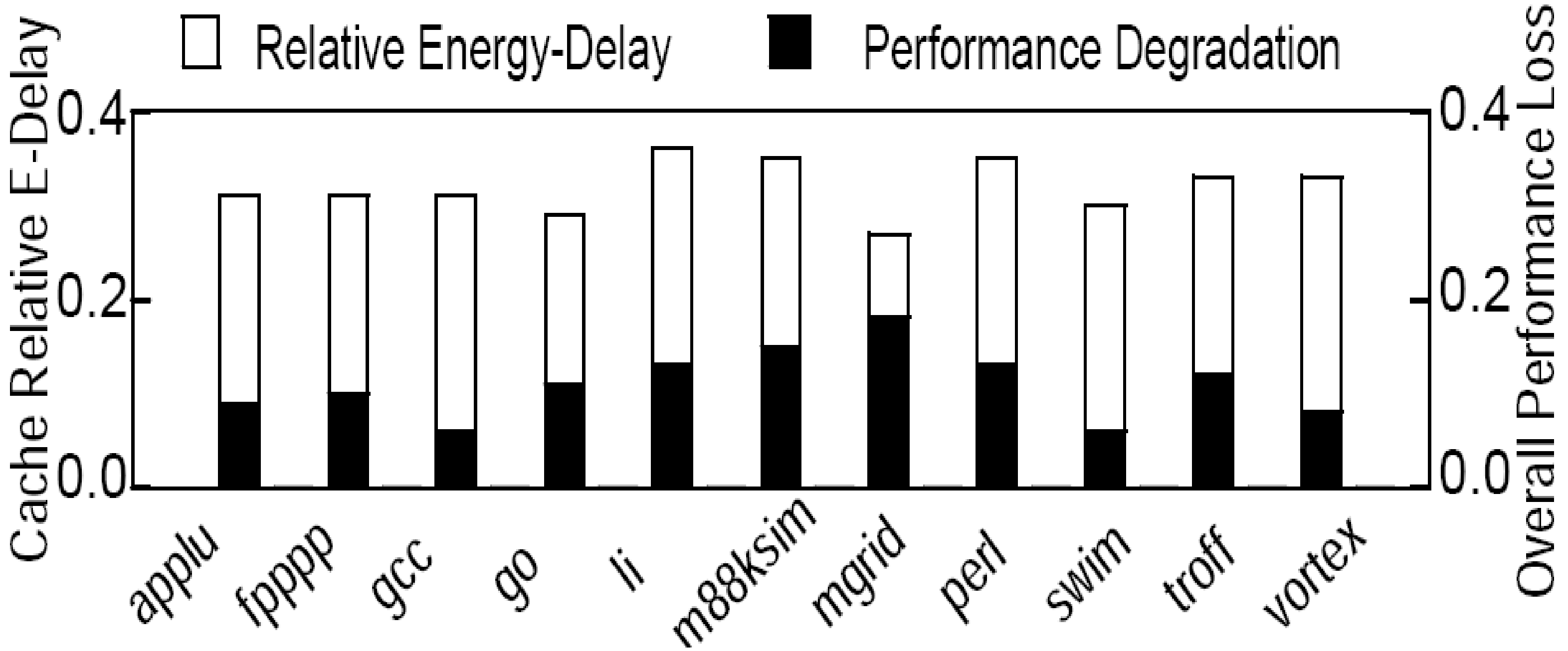
S. Plazar: Memory-based Optimization Techniques for Real-Time Systems, PhD thesis, TU Dortmund, June 2012

Way prediction/selective direct mapping



[M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, K. Roy: Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping, *MICRO-34*, 2001]

Results for the paper on way prediction (2)



Conclusion

- SPM allocation:
  - MMUs
  - multi-cores
- Code-Layout-Transformations
- Prefetching
- Reconciling Compilers and Timing Analysis
- Improving Timing Predictability