ARTIST2 Summer School 2008 in Europe
Autrans (near Grenoble), France
September 8-12, 2008

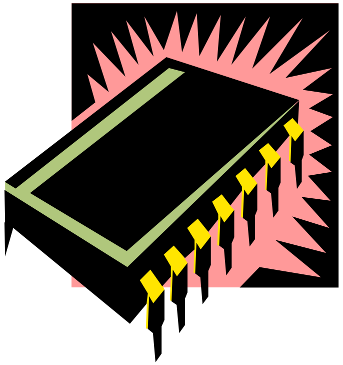
Memory-architecture aware compilation

Lecturers: Peter Marwedel, Heiko Falk
Informatik 12
TU Dortmund, Germany

The Problem with Memories

Memories?

Oops!
Memories!

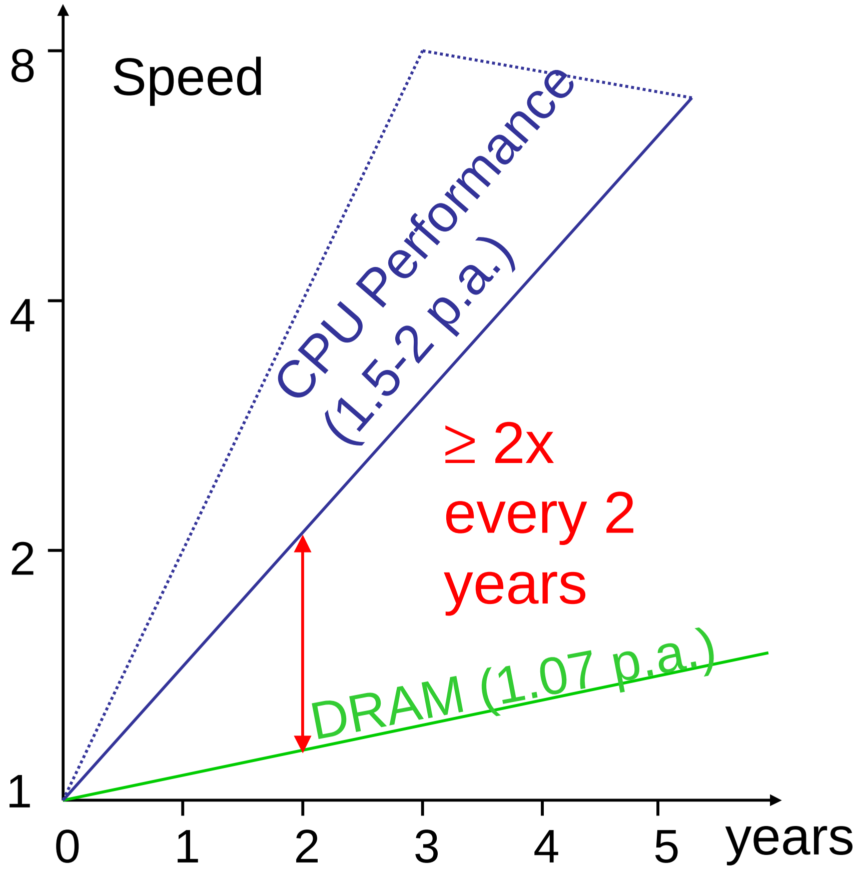


1. Increasing speed gap
2. Major consumer of electrical energy
3. Timing predictability difficult to achieve
4. …

Or: Why work on processors if memory is where the bottleneck is?

Trends for the speeds

Speed gap between processor and main DRAM increases



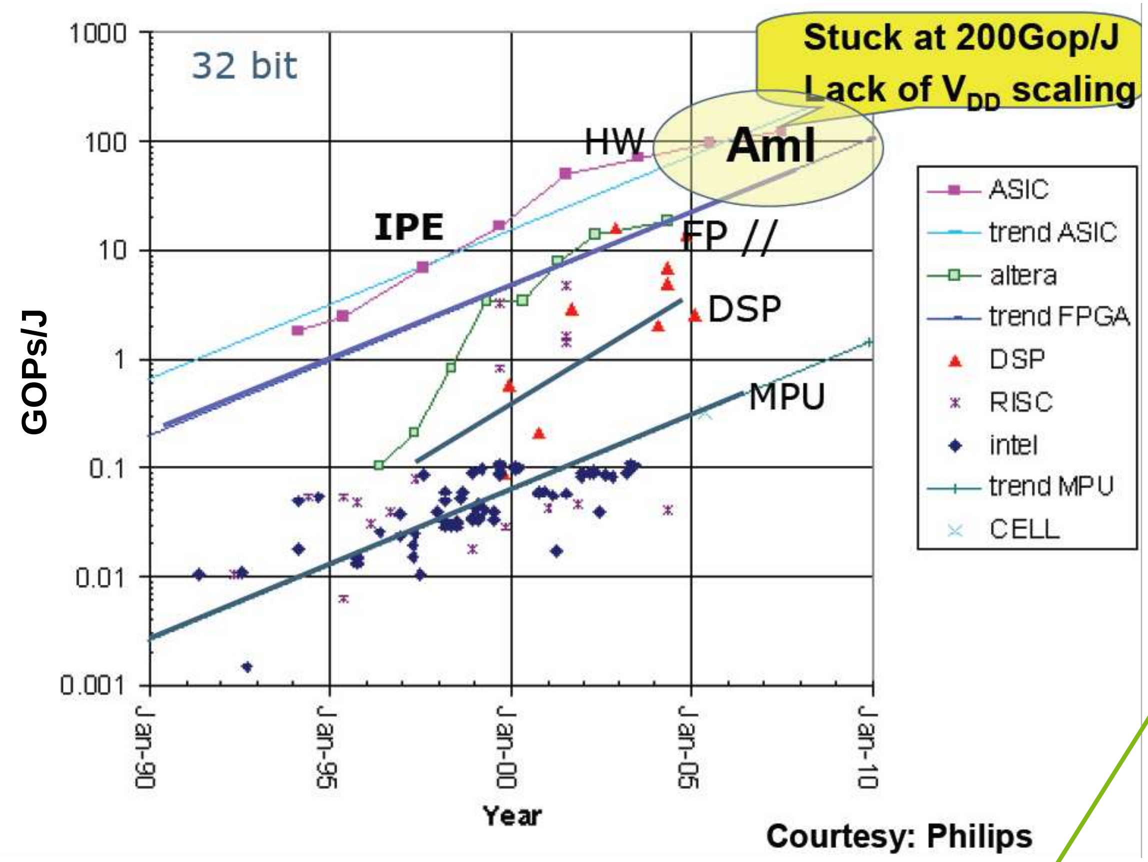
Similar problems also for embedded systems & MPSoCs

☞ Memory access times >> processor cycle times (today: e.g. 100 x)

☞ "Memory wall" problem; uniform memory access a myth

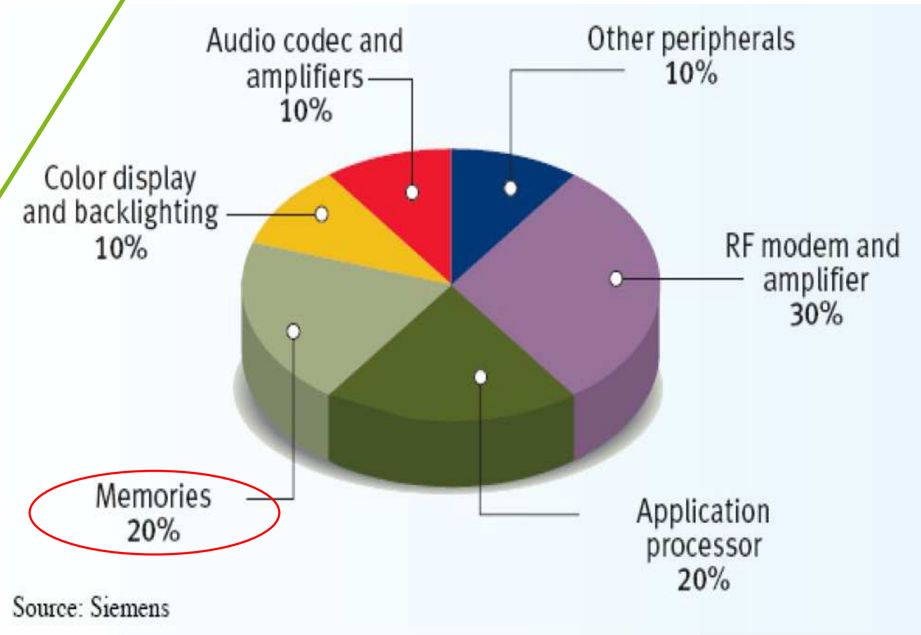
[P. Machanik: Approaches to Addressing the Memory Wall, TR Nov. 2002, U. Brisbane]

Importance of Energy Efficiency



Courtesy: Philips

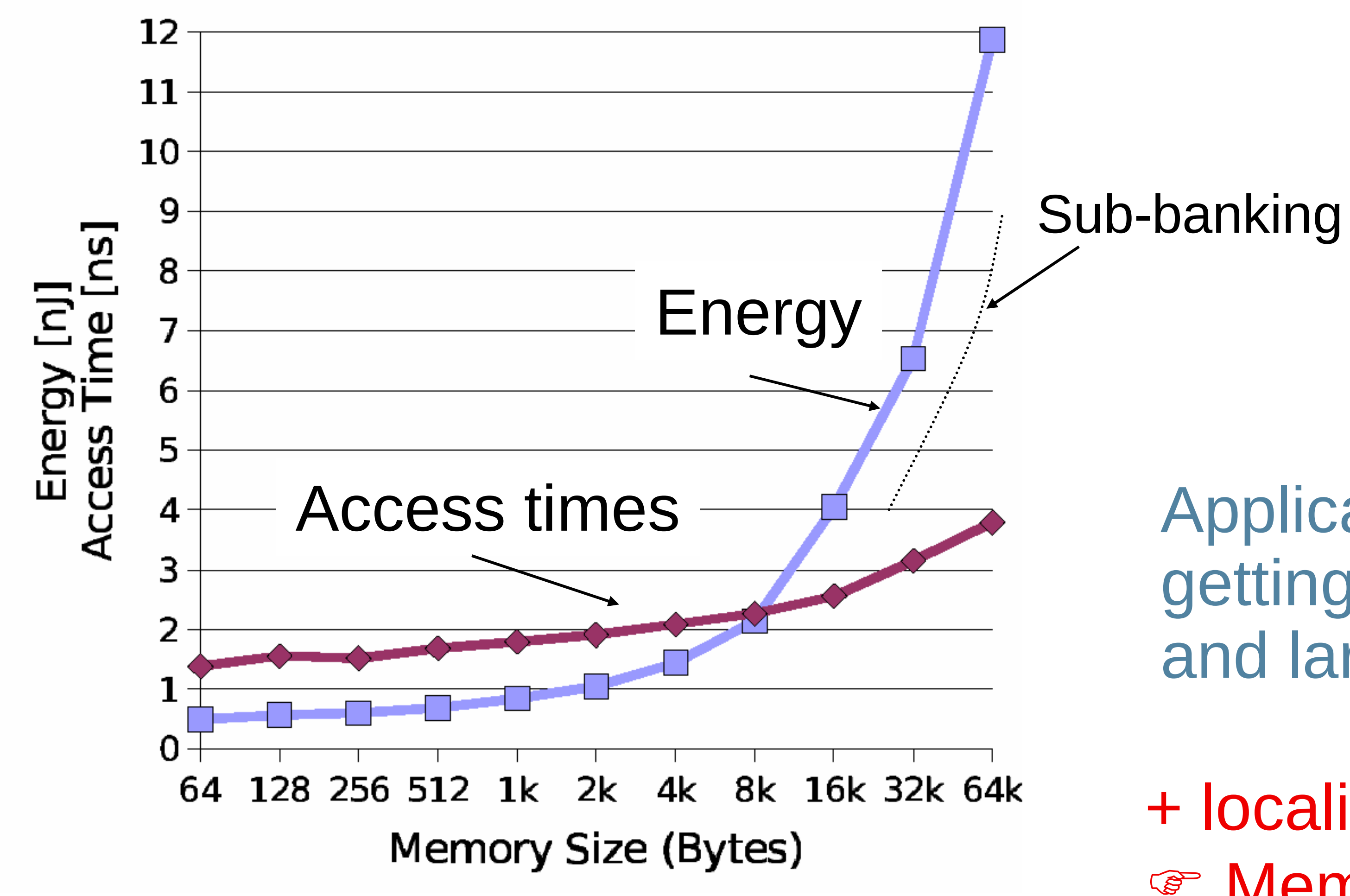
IPE=Inherent power efficiency;
AmI=Ambient Intelligence



Source: Siemens

O. Vargas (Infineon Technologies): Minimum power consumption in mobile-phone memory subsystems; Pennwell Portable Design - September 2005;

Dependency on the size



Applications are getting larger and larger …

+ locality of references

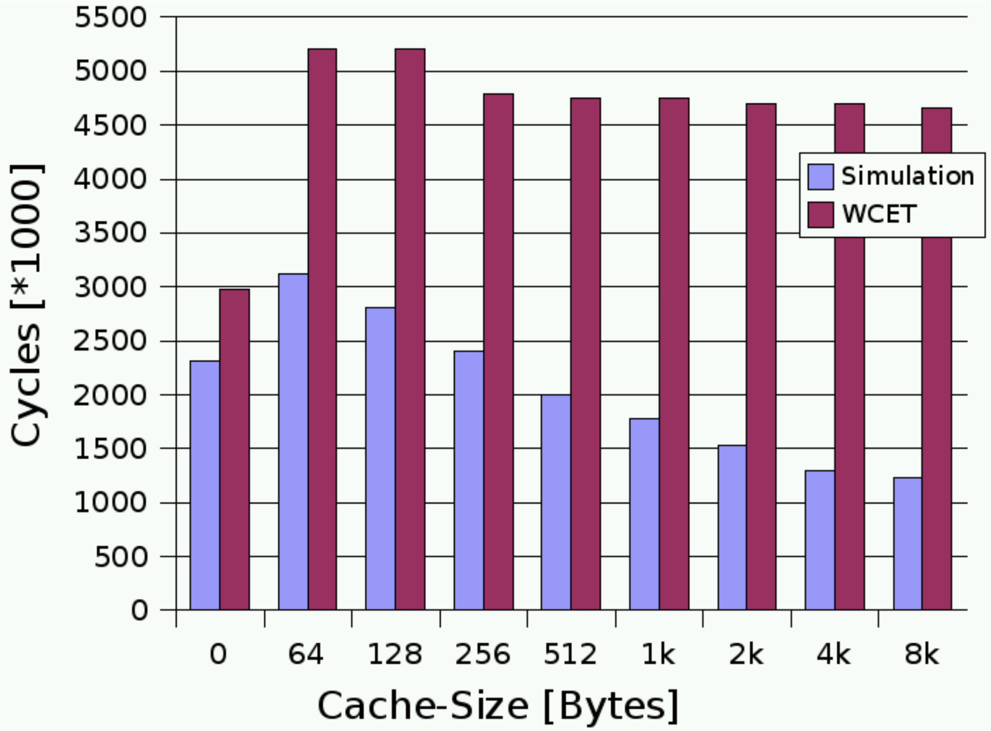
☞ Memory hierarchies

Timing Predictability

Many embedded systems are real-time systems

☞ computations to be finished in a given amount of time

Most memory hierarchies (e.g. caches) for PC-like systems designed for good average case, not for good worst case behavior.



Worst case execution time (WCET) larger than without cache

G.721: using unified Cache@ARM7TDMI

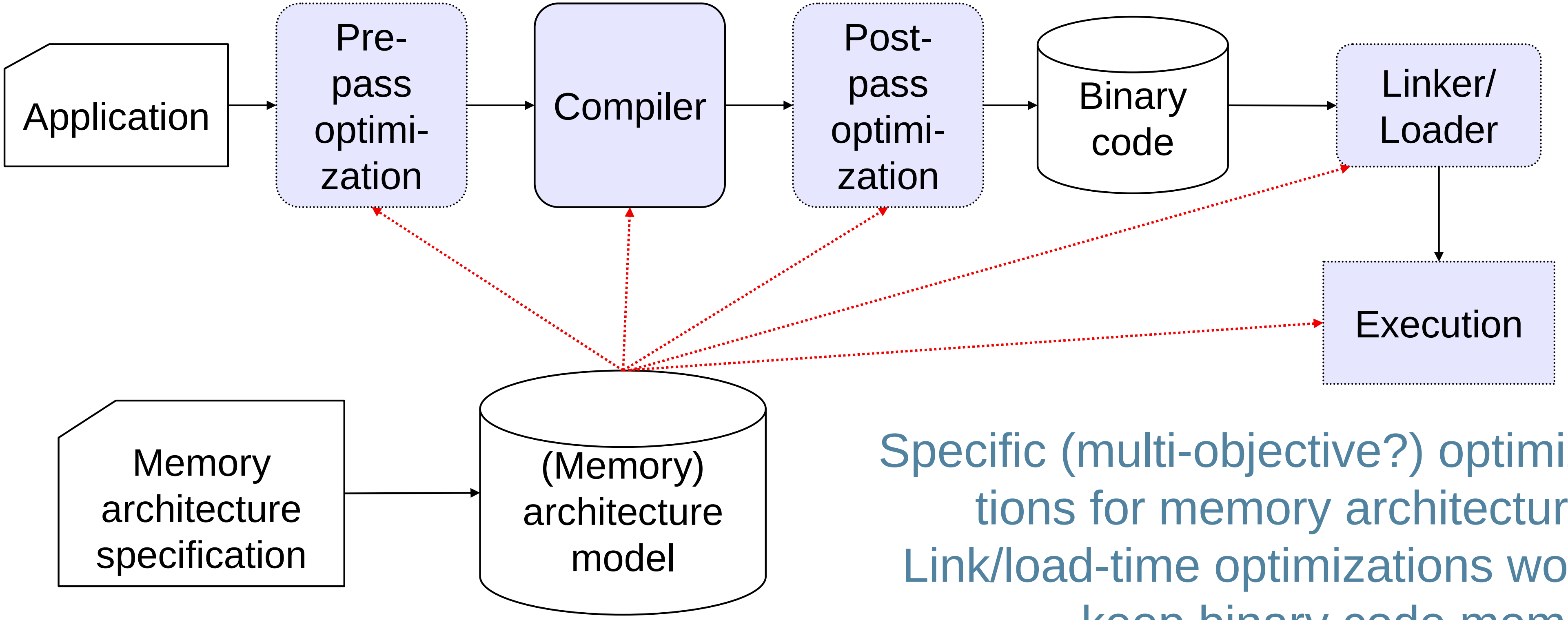
See later slide for experimental setup

Vision

☞ Multiple objectives.

What are optimizing compilers actually optimizing for?

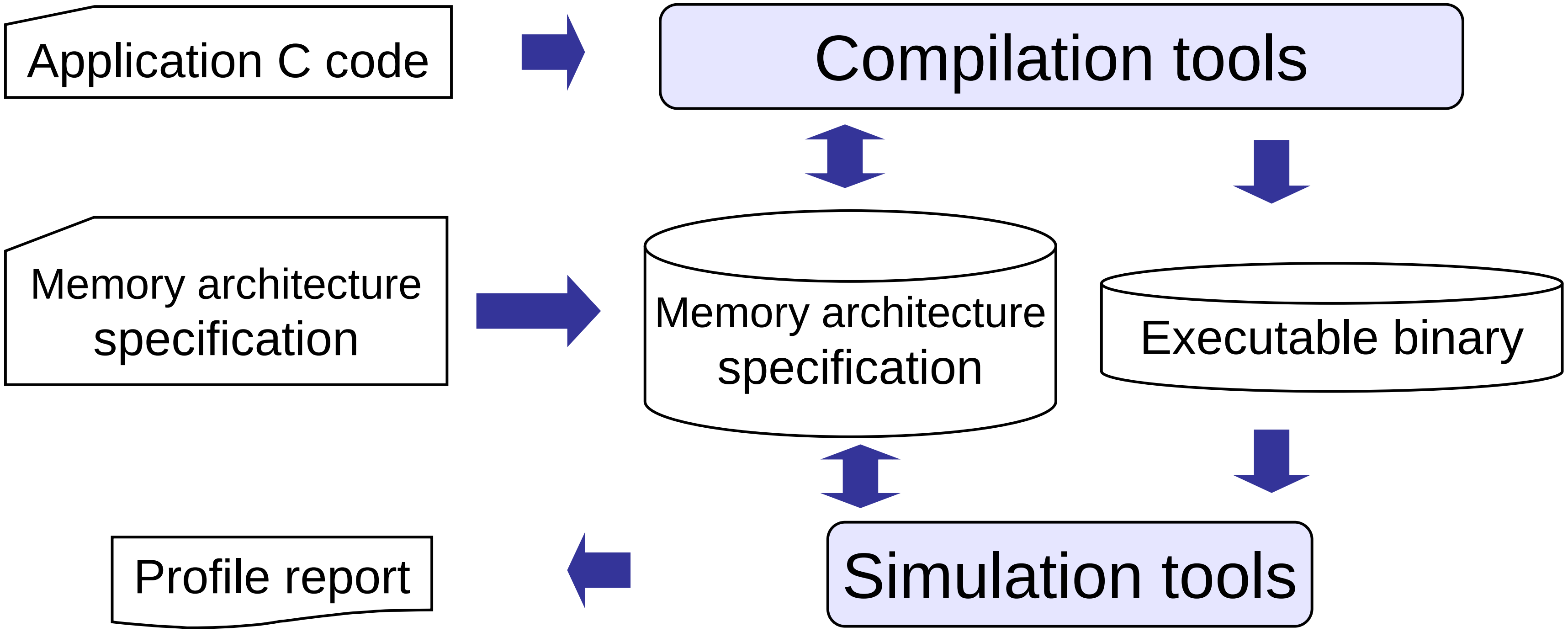
What is their cost model? ☞



Specific (multi-objective?) optimizations for memory architectures; Link/load-time optimizations would keep binary code memory architecture independent

☞ Integration of various optimizations into framework

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].

MACC still under development;
Tools for specific optimizations exist ☞

Related Work

- Optimizations exploiting burst mode for SDRAMs, using loop unrolling and architecture description language EXPRESSION (Dutt, Srivastava; UC Irvine)
- Smart linker (K. De Bosschere et al., U. Ghent)
- Architecture description language ArchC (G. Araujo, U. Campinas)
- Work on scratchpad optimizations (M. Kandemir, Penn State U.; R. Barua, U. Maryland; Egger+Lee, SNU; IMEC; Marwedel et al., TU Dortmund)
- ..

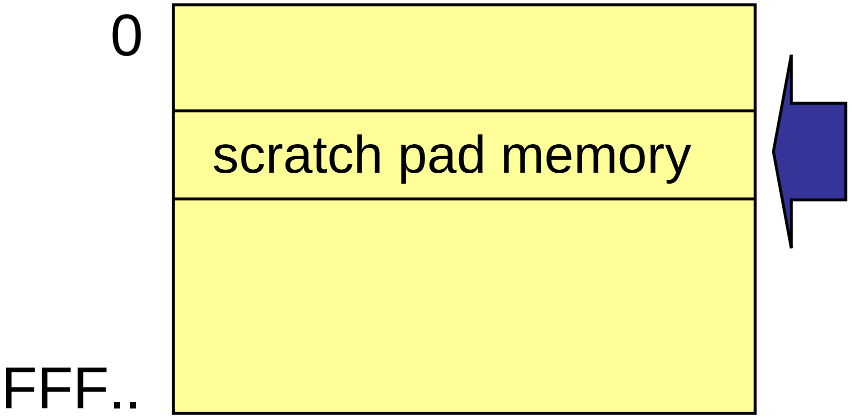
Existing work covers only some of the aspects

What we have done: Optimizations for Scratch Pad Memories (SPM)

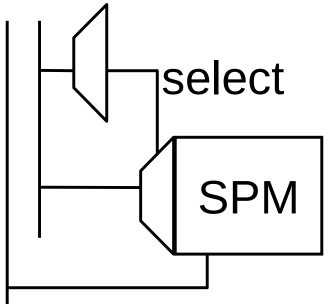
SPMs are small, physically separate memories mapped into the address space;

Selection is by an appropriate address decoder (simple!)

Address space

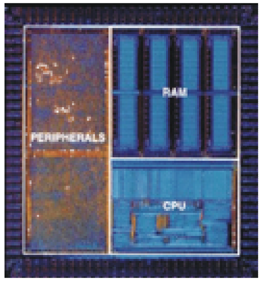


Small; no tag memory



Example

ARM7TDMI cores, well-known for low power consumption



SPMs are fast, energy-efficient, timing-predictable

Predictability and scratch-pad memories

… ***pre-run-time scheduling*** *is often the only practical means of providing predictability in a complex system.*

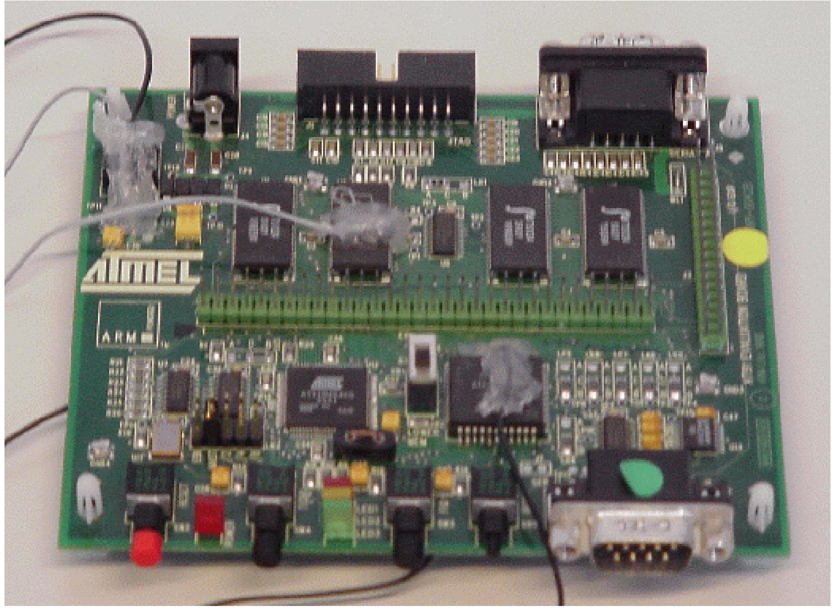
[J. Xu, D. Parnas: On satisfying timing constraints in hard real-time systems, *IEEE Trans. Soft. Engineering*, 1993, p. 70–84]

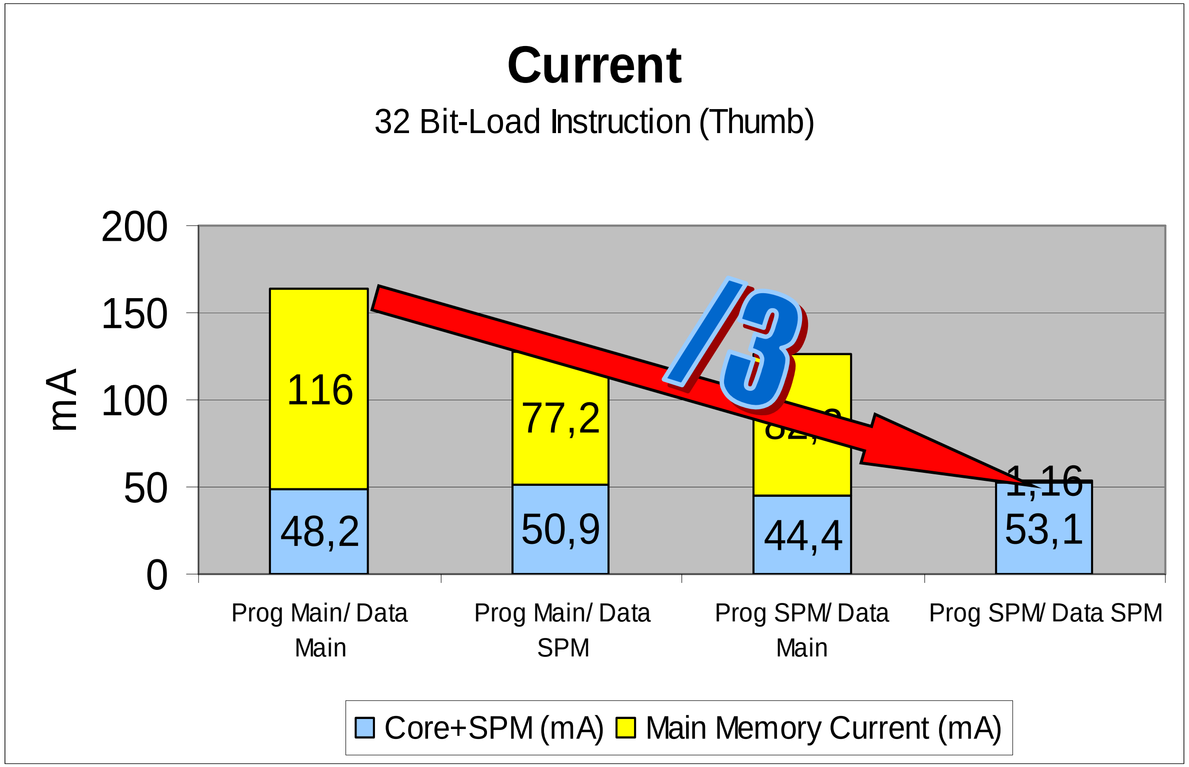
… In essence, we must reinvent computer science. Fortunately, we have quite a bit of knowledge and experience to draw upon. Architecture techniques such as ***software-managed caches*** *promise to deliver much of the benefit of memory hierarchy without the timing unpredictability.*

[Ed Lee: Absolutely Positively on Time: What would it take?, *IEEE Computer*, 2005]

Comparison of currents using measurements

E.g.: ATMEL board with ARM7TDMI and ext. SRAM

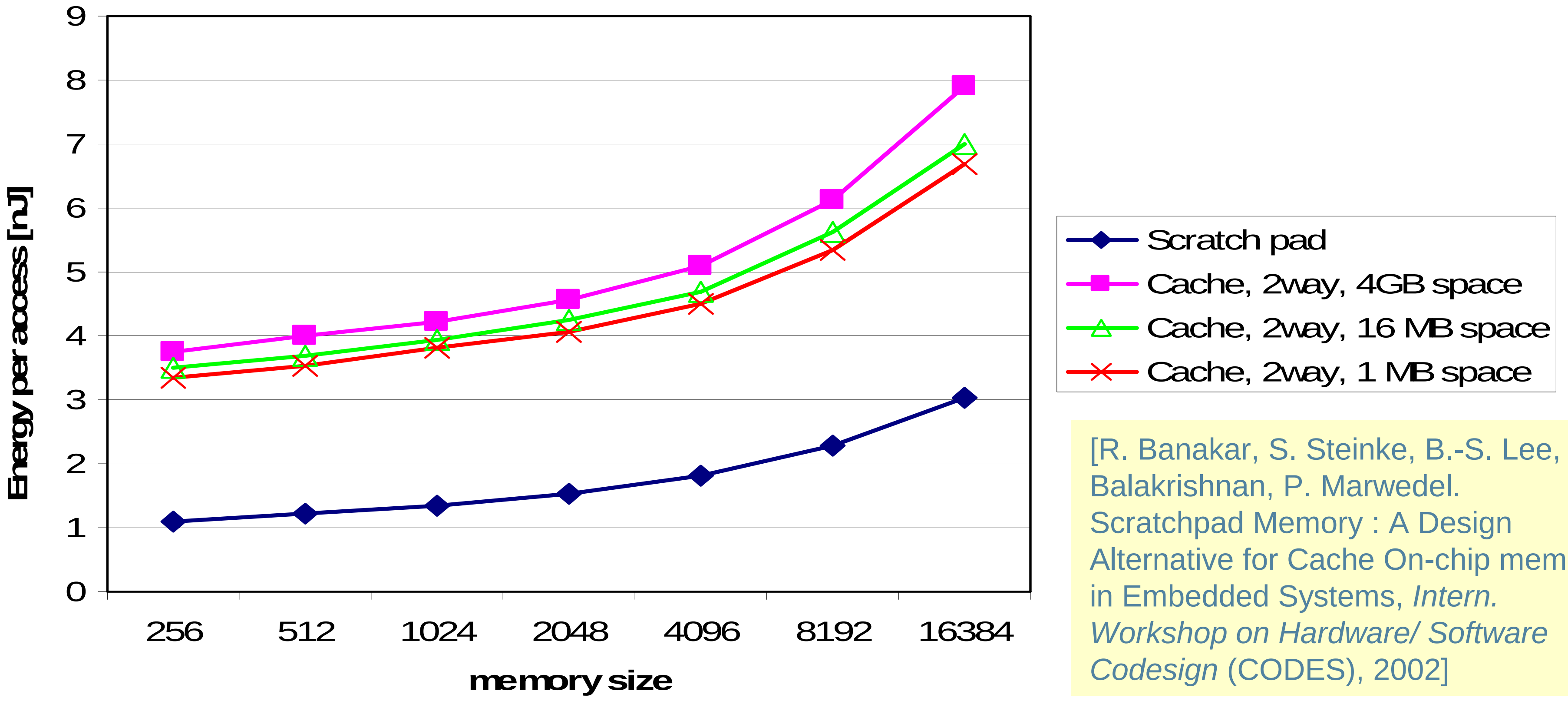




Even larger savings in terms of energy.

Why not just use a cache ?

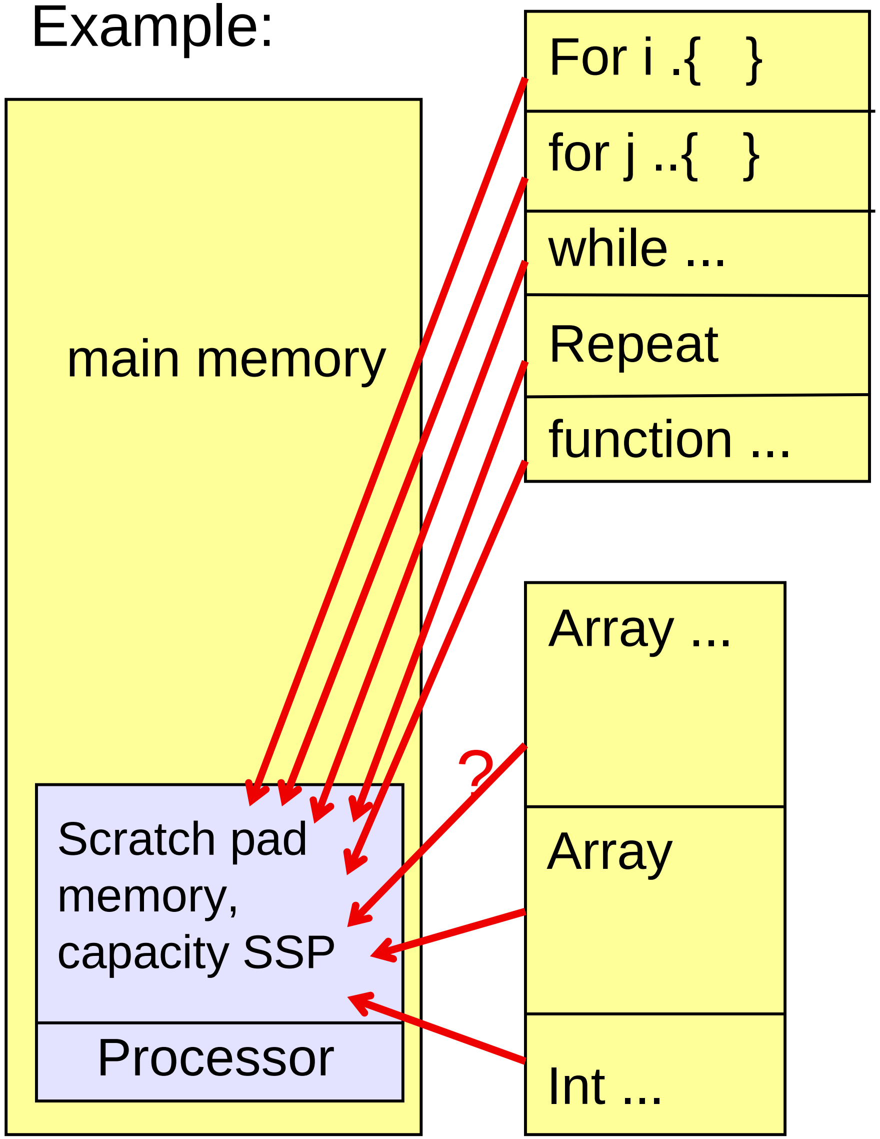
1. Timing predictability
2. Hardware complexity
3. Energy consumption (in tags, comparators and muxes)



[R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, P. Marwedel. Scratchpad Memory : A Design Alternative for Cache On-chip memory in Embedded Systems, *Intern. Workshop on Hardware/ Software Codesign* (CODES), 2002]

Migration of data and instructions - Global optimization model -

Example:



main memory

For i .{ }

for j ..{ }

while ...

Repeat

function ...

Array ...

Array

Int ...

?

Scratch pad memory, capacity SSP

Processor

Which object (array, loop, etc.) to be stored in SPM?

Non-overlaying memory allocation:

Gain g_k & size s_k for each object k.

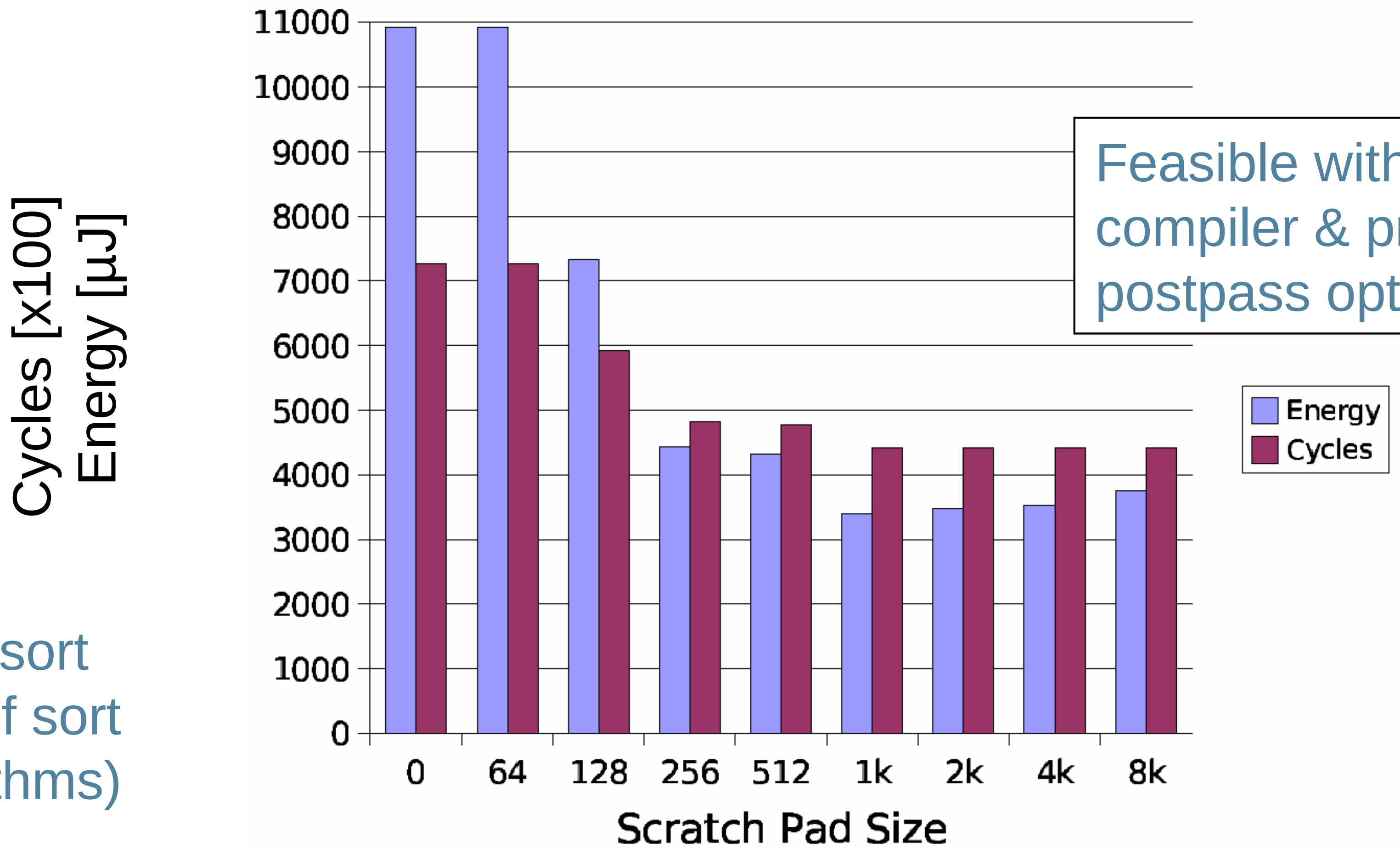
Maximise gain $G = \Sigma g_k$, respecting size of SPM $SSP \geq \Sigma\ s_k$.

Solution: Knapsack algorithm.

Overlaying allocation:

Moving objects back and forth between hierarchy levels

A first, non-overlaying approach for functions and global variables



Feasible with standard compiler & pre- or postpass optimization

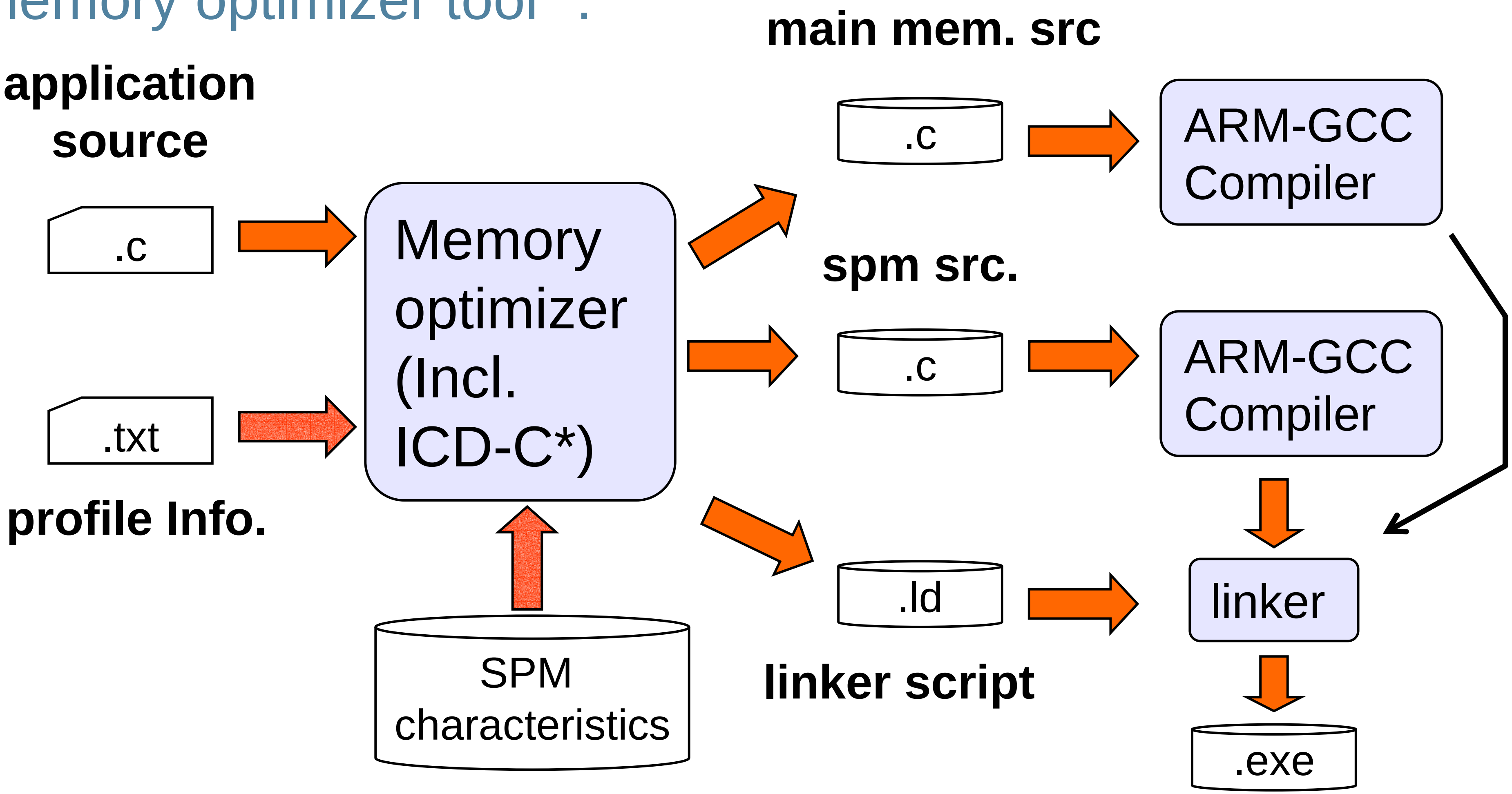
Multi_sort (mix of sort algorithms)

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

Extensions to smaller code blocks, stacks and heaps exist

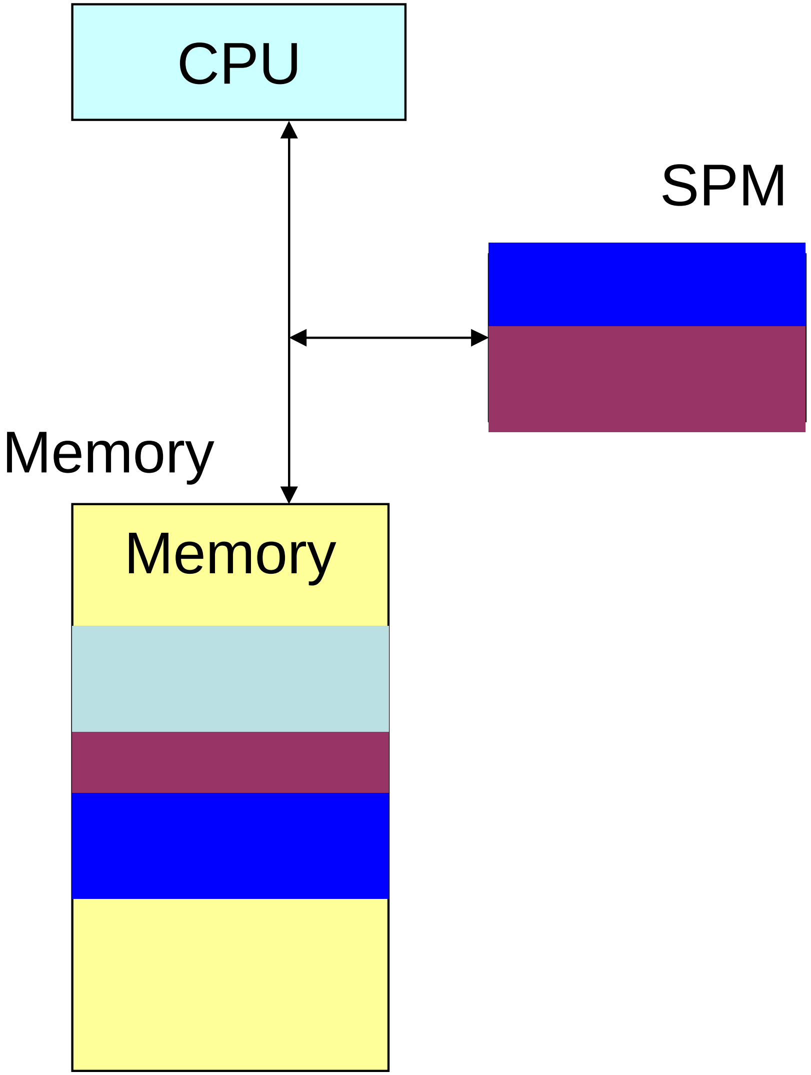
Using these ideas in a pre-pass tool

Source is split into 2 different files by specially developed memory optimizer tool *.



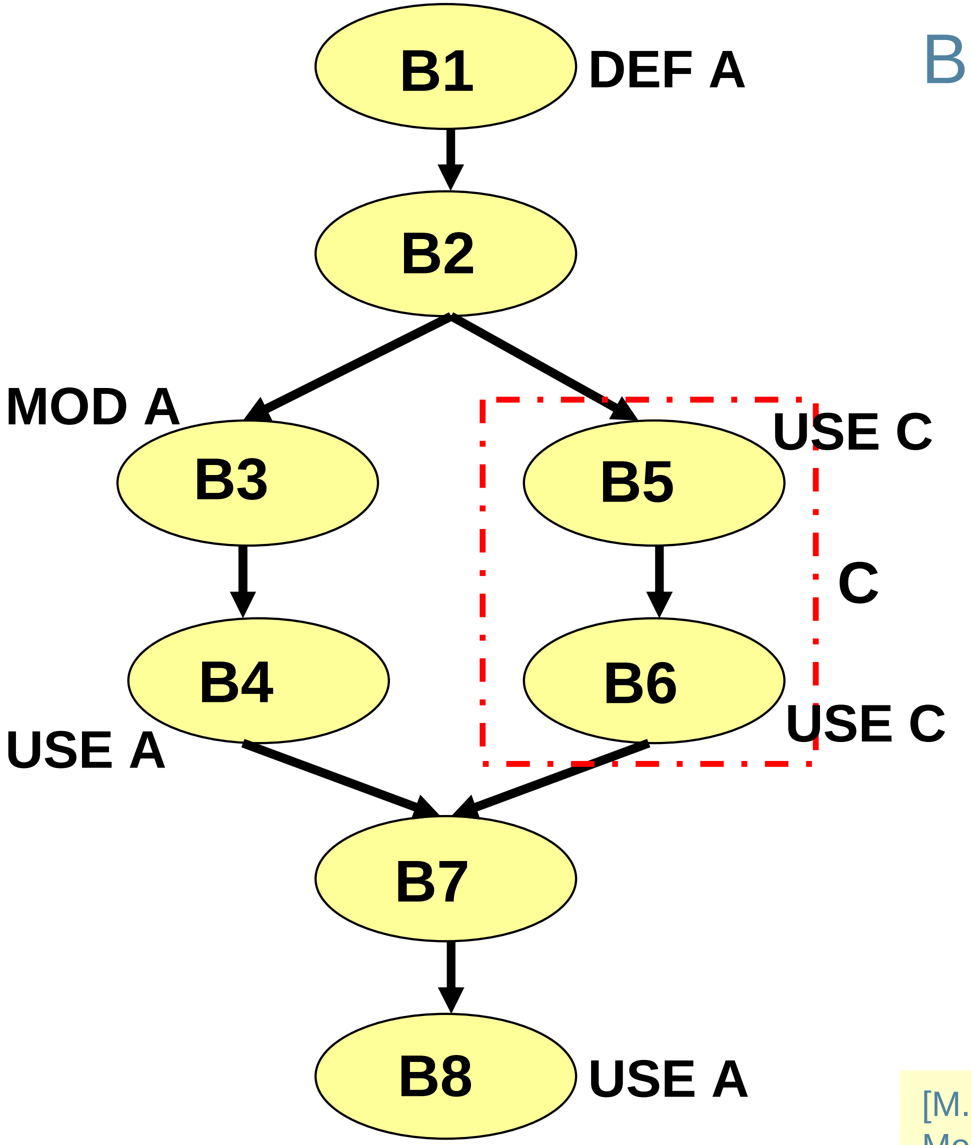
*Built with tool design suite ICD-C available from ICD (see www.icd.de/es)

Non-overlaying allocation problematic for multiple hot spots ☞Overlaying allocation



- Effectively results in a kind of **compiler-controlled overlays** for SPM
- Address assignment within SPM required

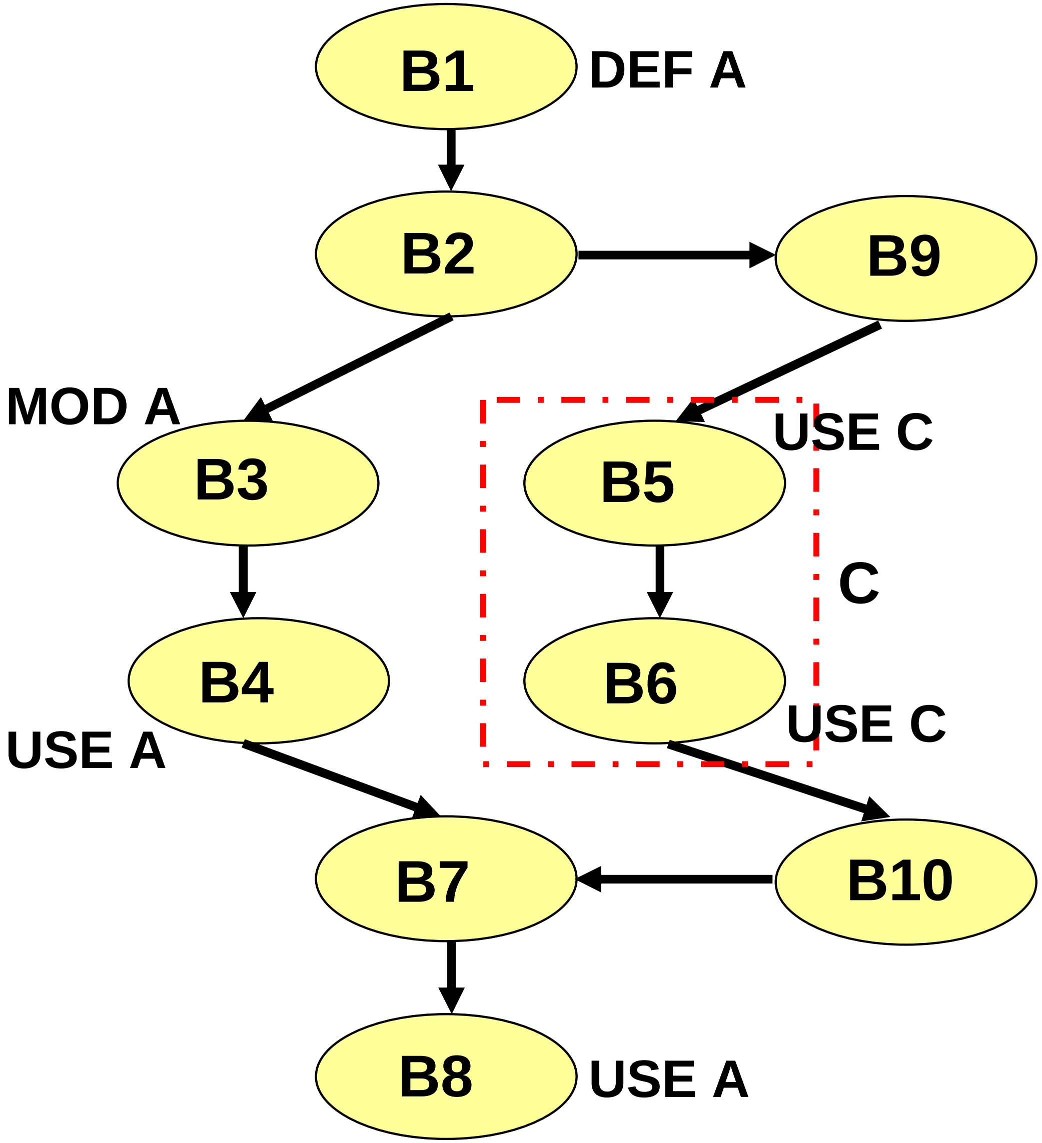
Overlaying allocation by Verma et al. (1)



Based on control flow graph.

[M.Verma, P.Marwedel: Dynamic Overlay of Scratchpad Memory for Energy Minimization, *ISSS,* 2004]

Overlaying allocation by Verma et al. (2)



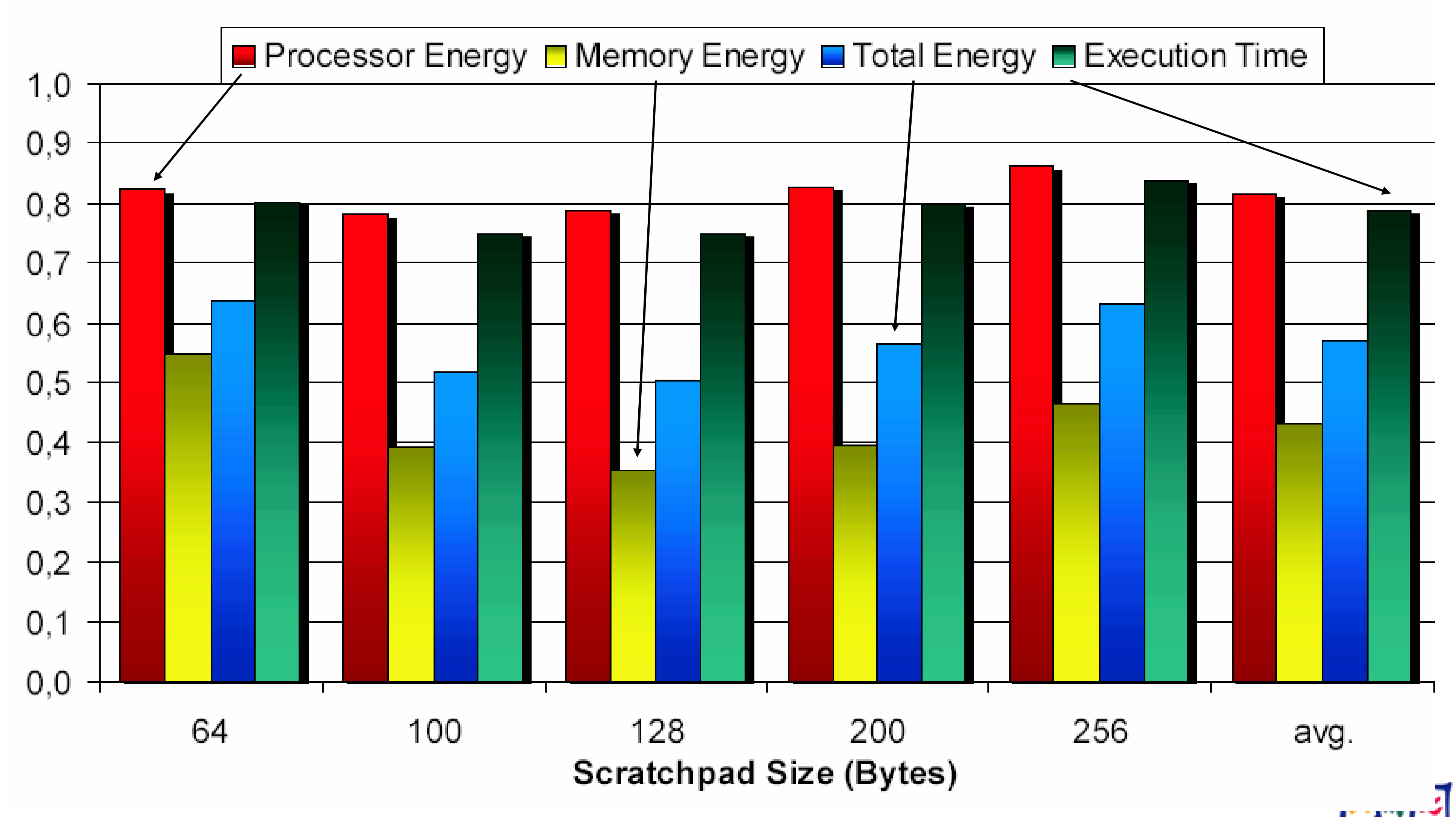
SPILL_STORE(A);
SPILL_LOAD(C);

Global set of ILP equations reflects cost/benefit relations of potential copy points

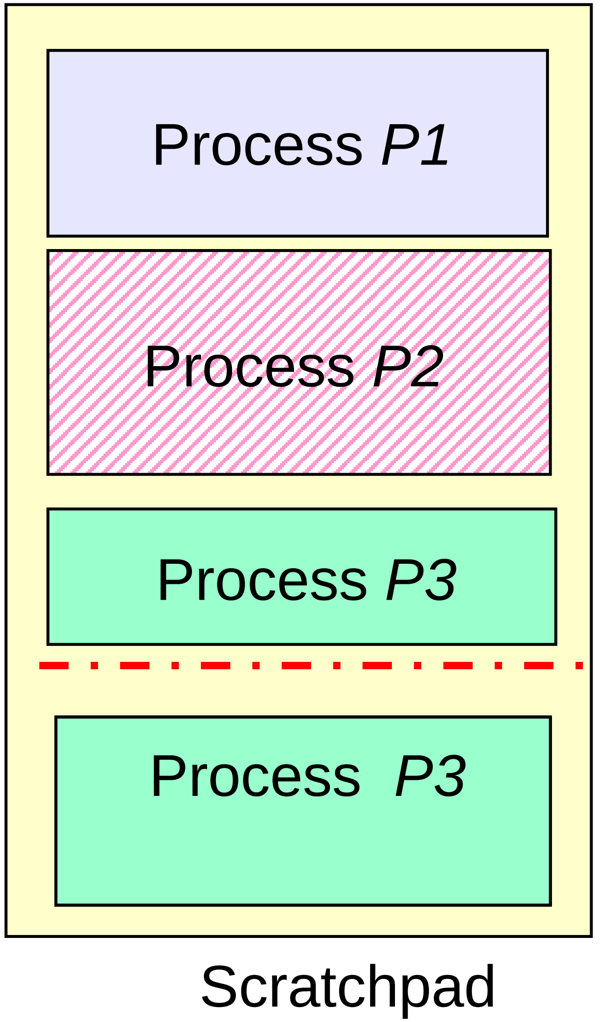
SPILL_LOAD(A);

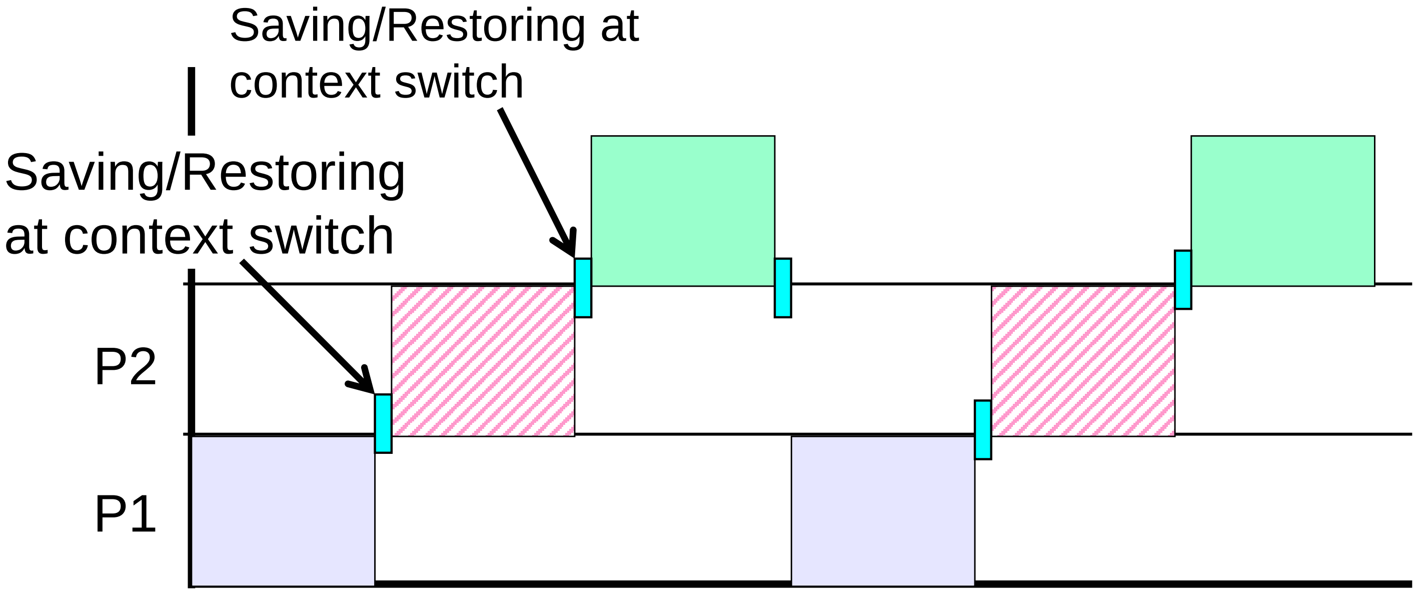
Code handled like data

Runtime/energy reduction with respect to non-overlaying (“static”) allocation



Multi-process Scratchpad Allocation: Hybrid Context Switch



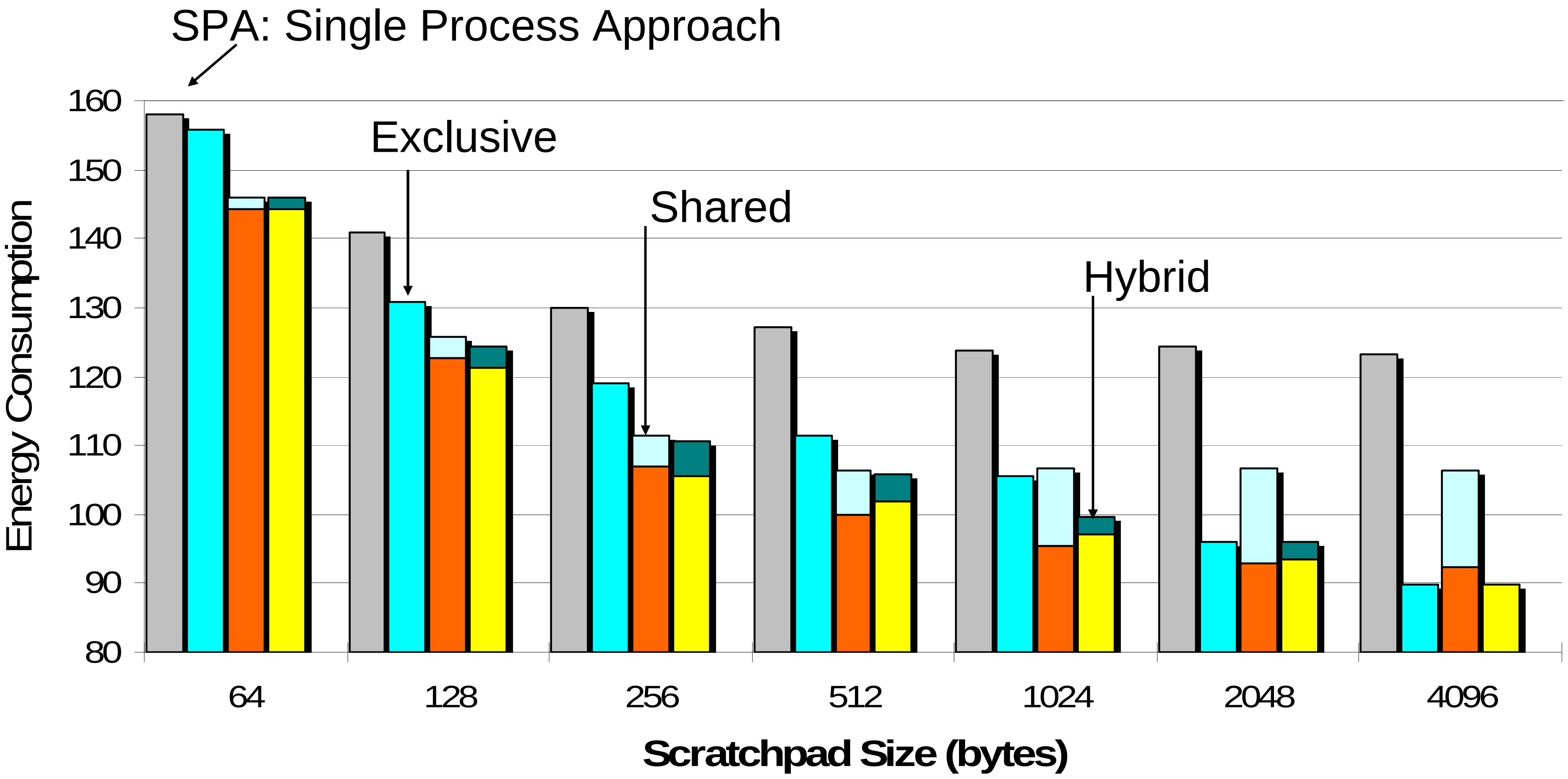

Saving/Restoring at context switch
Saving/Restoring at context switch
P2
P1

- Processes can use *shared* and *exclusively allocated* areas
- Saving/restoring required for shared area
- Optimization of sizes published by Verma

Multi-process Scratchpad Allocation: Results

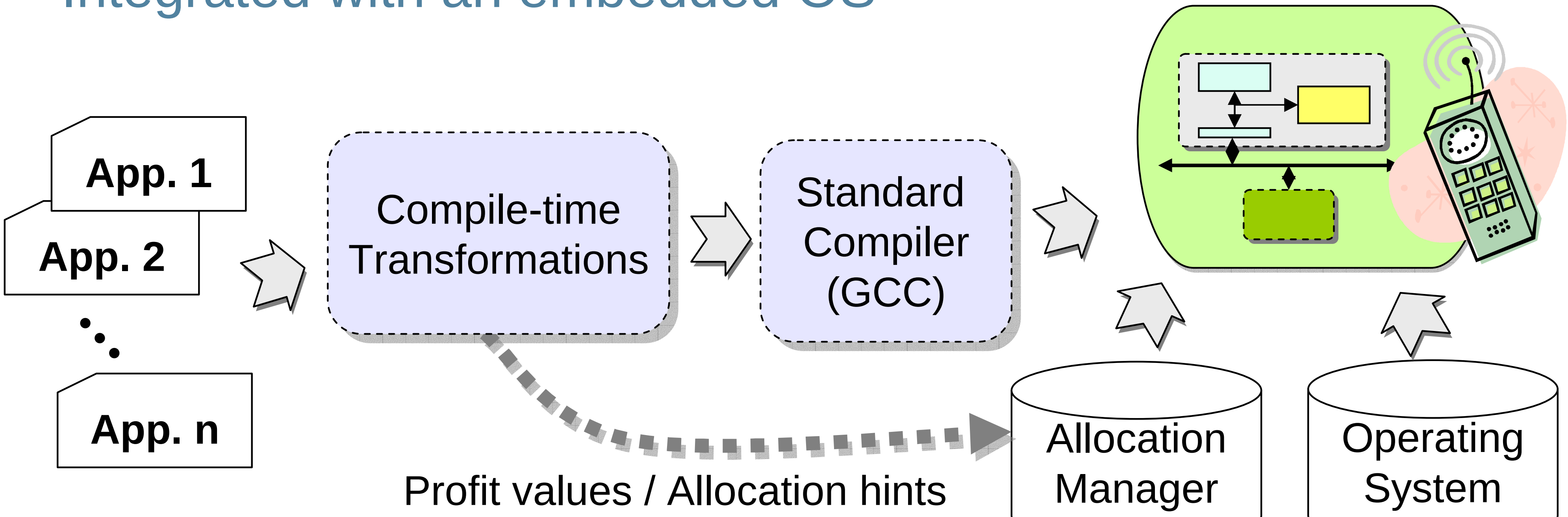
Hybrid approach superior to using only exclusively allocated or only shared areas

edge detection, adpcm, g721, mpeg



Dynamic set of applications

- 2 steps: compile-time analysis & runtime decisions
- No need to know all applications at compile-time
- Capable of managing runtime allocated memory objects
- Integrated with an embedded OS



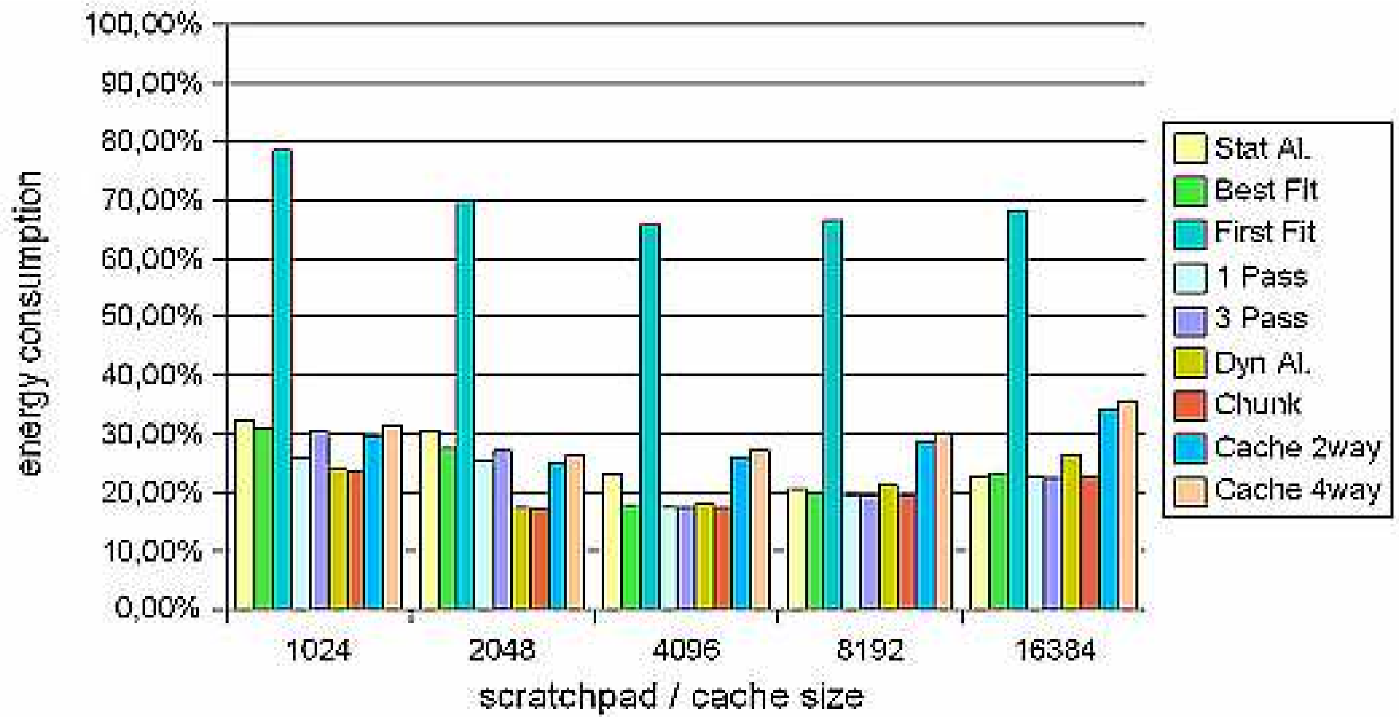
Using MPArm simulator from U. Bologna

Dynamic set of applications: Comparison of SPMM to Caches

- Baseline: Main memory only
- SPMM peak energy reduction by 83% at 4k Bytes scratchpad
- Cache peak: 75% at 2k 2-way $
- Application: sorting

- SPMM outperforms caches
- OS and libraries not considered yet
- Chunk allocation results:

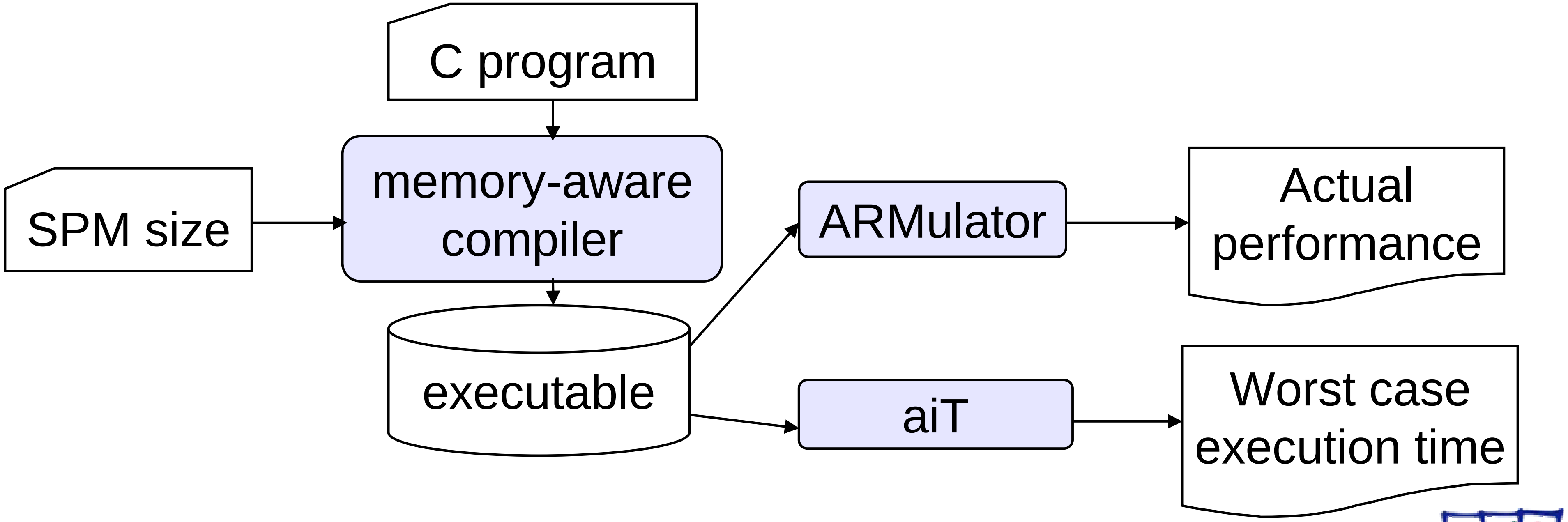
SPM Size	Δ 4-way
1024	74,81%
2048	65,35%
4096	64,39%
8192	65,64%
16384	63,73%



Scratch-pad/tightly coupled memory based predictability

Time-triggered, statically scheduled operating systems

- ☞ Let‘s do the same for the memory system
  - ☞ Are SPMs really more timing predictable?
  - ☞ Analysis using the aiT timing analyzer



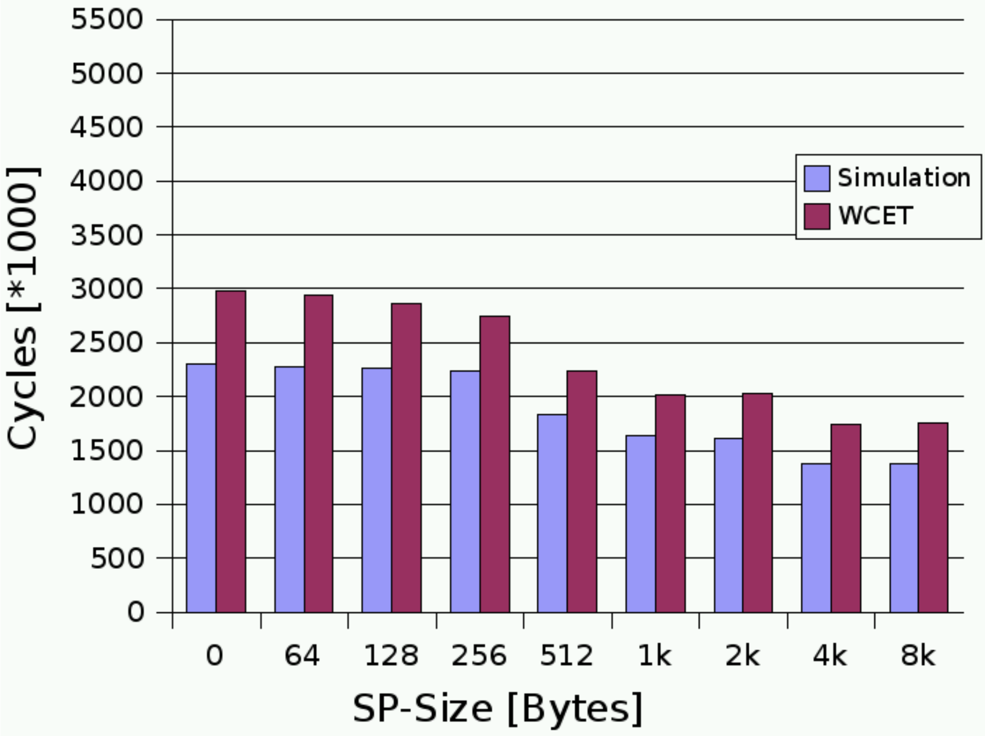
Architectures considered

ARM7TDMI with 3 different memory architectures:

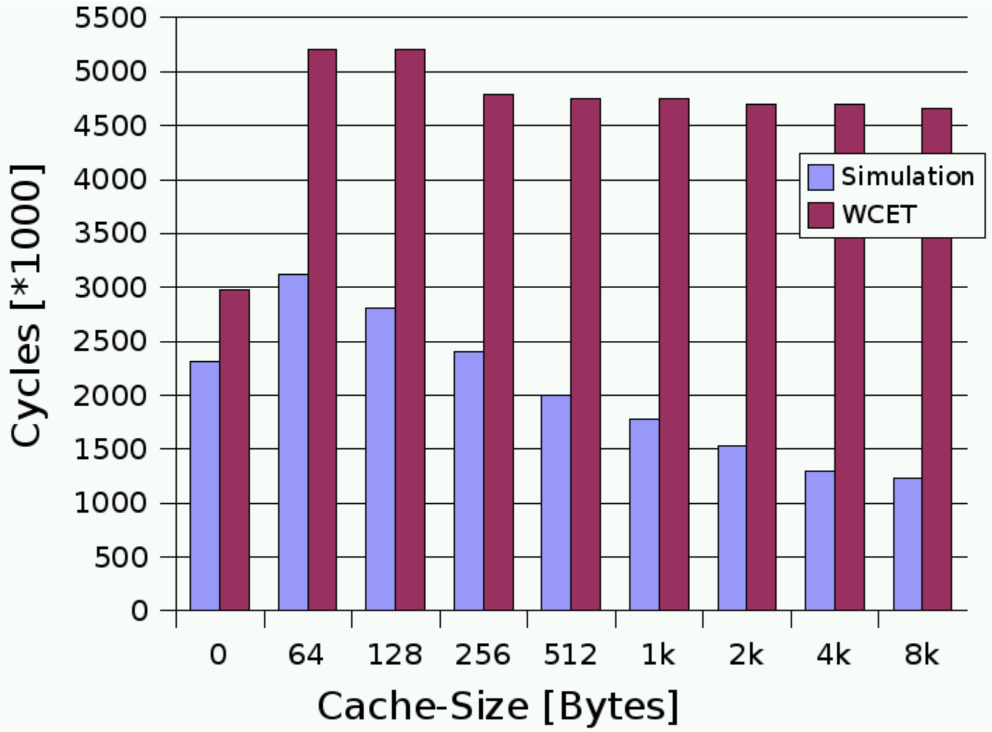
- **Main memory**
  LDR-cycles: (CPU,IF,DF)=(3,2,2)
  STR-cycles: (2,2,2)
  * = (1,2,0)
- **Main memory + unified cache**
  LDR-cycles: (CPU,IF,DF)=(3,12,6)
  STR-cycles: (2,12,3)
  * = (1,12,0)
- **Main memory + scratch pad**
  LDR-cycles: (CPU,IF,DF)=(3,0,2)
  STR-cycles: (2,0,0)
  * = (1,0,0)

Results for G.721

Using Scratchpad:



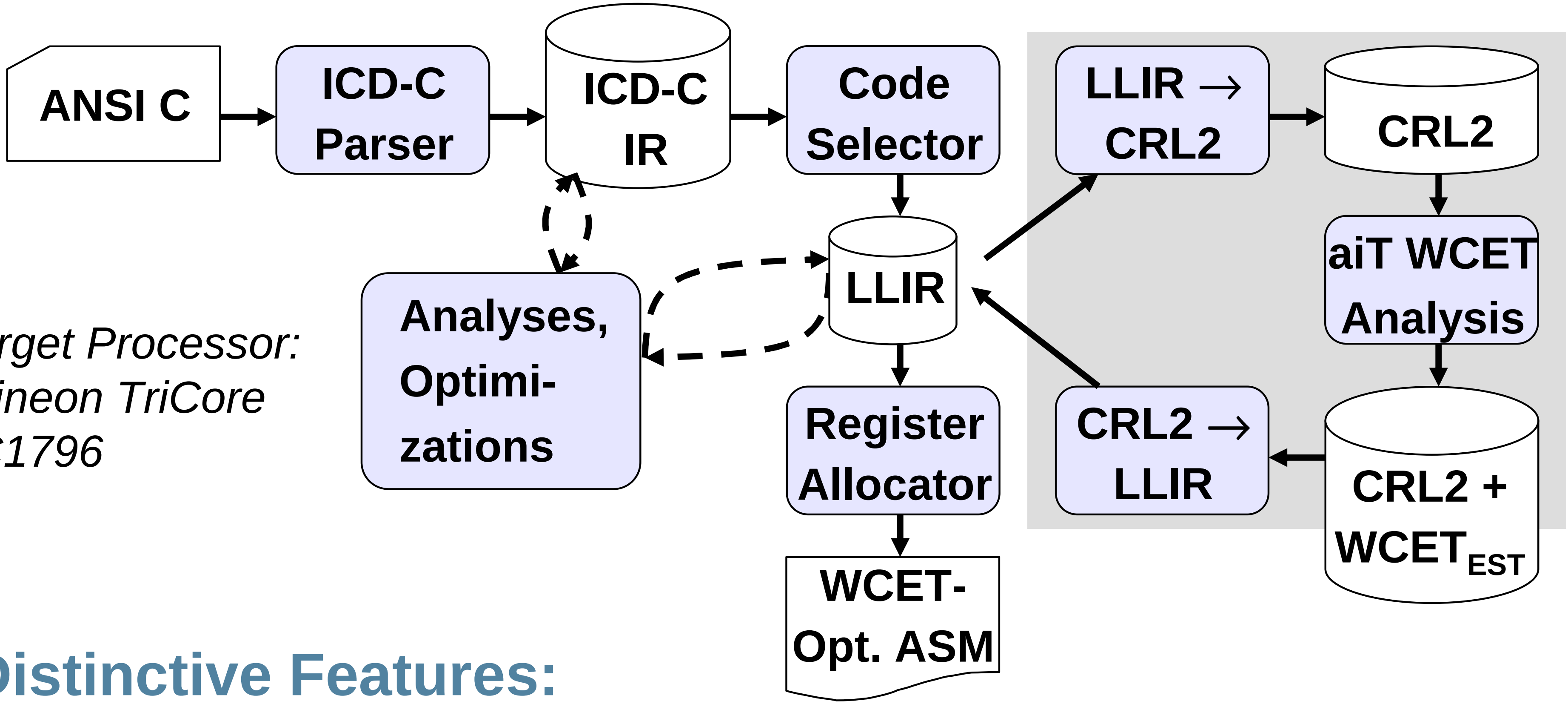
Using Unified Cache:



Yes, they are clearly more timing predictable!

- L. Wehmeyer, P. Marwedel: Influence of Onchip Scratchpad Memories on WCET: *4th Intl Workshop on worst-case execution time analysis, (WCET)*, 2004
- L. Wehmeyer, P. Marwedel: Influence of Memory Hierarchies on Predictability for Time Constrained Embedded Software, *Design Automation and Test in Europe (DATE)*, 2005

A WCET-Aware C-Compiler *(WCC)*



Target Processor: Infineon TriCore TC1796

Distinctive Features:

- Integration of timing models into compilation & optimization process.
- Tight coupling of WCET analyzer aiT into WCC's backend.

Imported WCET Data & Flow Facts in WCC

Most important WCET data imported into WCC:

- $WCET_{EST}$ of the entire program
- $WCET_{EST}$ of each single basic block
- Worst-Case execution frequencies of each CFG edge

Flow Fact Annotation within WCC:

- Annotation of e.g. loop iteration bounds directly in C source code: **`_Pragma( “loopbound min 10 max 10” );`**
- Since compiler optimizations may restructure loops and thus their annotated bounds, WCC automatically keeps Flow Facts consistent during all applied optimizations.

Problems during WCET_{EST} Minimization

The Worst-Case Execution Path *(WCEP)*:

- WCET of a program *P* = length of longest execution path of *P (WCEP)*
- To minimize *P*'s WCET_{EST}, optimizations must exclusively focus on those parts of *P* lying on the WCEP.

☞ *Optimization of parts not lying on the WCEP don't reduce WCET_{EST} at all!*

☞ *Optimization strategies for WCET_{EST} Minimization must have detailed knowledge about the WCEP.*

☞ *During optimization, the WCEP may switch within the CFG*

Graph Colouring Register Allocation

1. **Initialisation:** Build Interference Graph $G = (V, E)$ with
   $G = \{$*virtual registers*$\} \cup \{K$ *physical processor registers*$\}$,
   $e = \{v, w\} \in E \Leftrightarrow$ VREGs *v* and *w* may never share the same PHREG, i.e. *v* and *w* interfere
2. **Simplification:** Successively remove all $v \in V$ with deg. $< K$; push each *v* onto stack *S*
3. **Spilling:** After step 2, each node of *G* has degree $\geq K$. Select one $v \in V$; mark *v* as *potential spill*; remove *v* from *V*; push *v* onto stack *S*
4. Repeat steps 2 and 3 until $G = \emptyset$.

[A. W. Appel, Modern compiler implementation in C, 1998]

Problem of Standard Graph Colouring

3. **Spilling:** After step 2, each node of G has degree $\geq K$. Select one $v \in V$; mark v as *potential spill*; remove v from V; push v onto stack S

Which node v should be selected as potential spill?

Graph colouring implementations heuristically select …

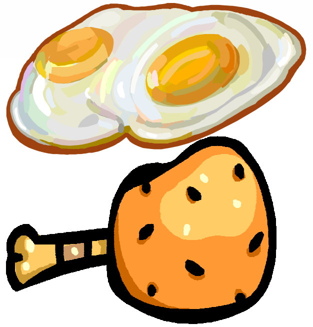
… either some arbitrary node or …

… the node with highest degree or …

… a node in some inner loop.

☞ ***Uncontrolled spill code generation – potentially along Worst-Case Execution Path (WCEP) defining the WCET!***

A Chicken-Egg Problem



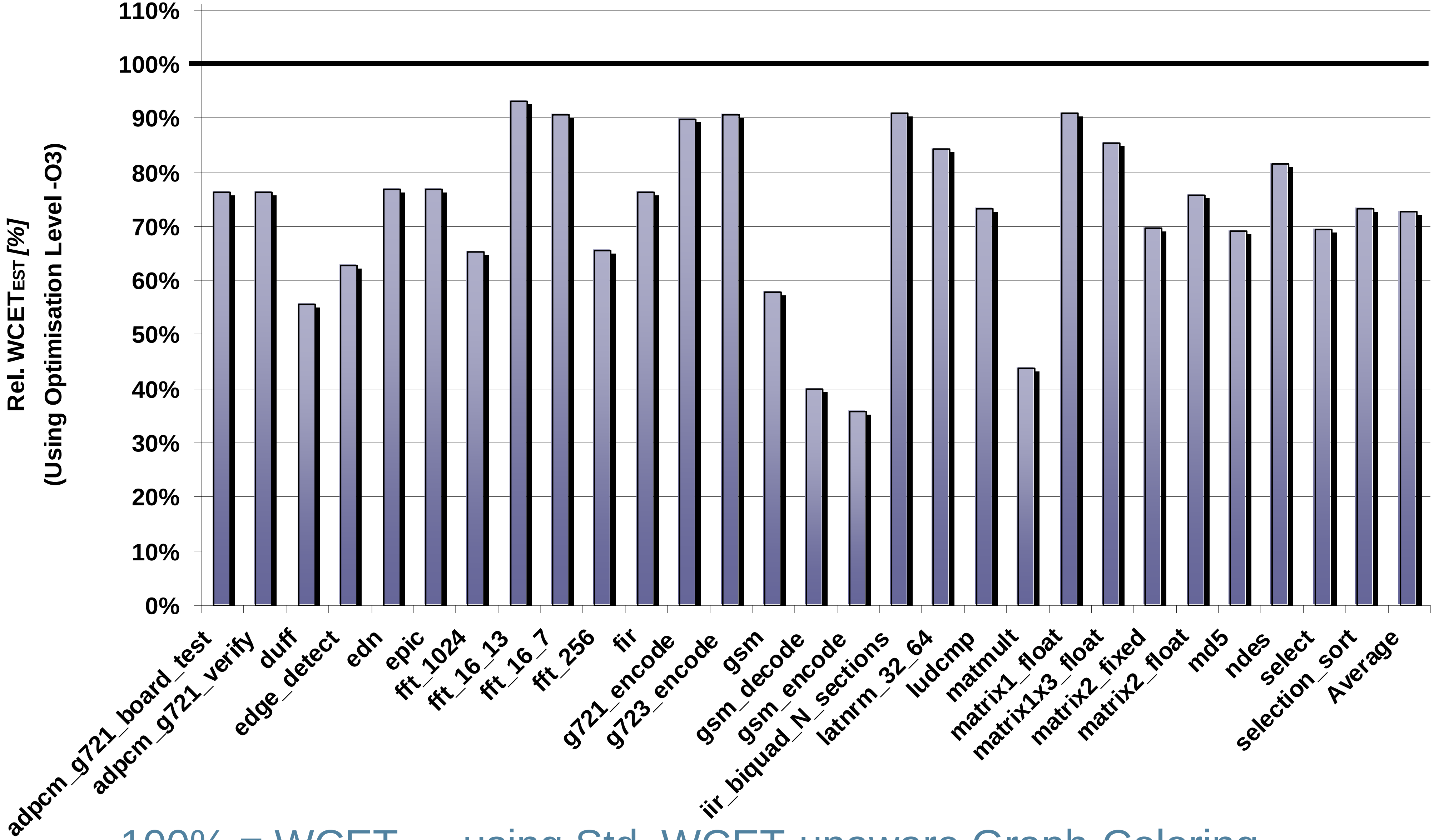
A WCET-aware Register Allocator…

- …relies on WCET data provided by WCET analysis
- …but can‘t obtain WCET data since code containing virtual registers is not analysable!

The Way out:

- Start by spilling each VREG onto stack
  ☞ code is fully analysable
- Perform WCET analysis, get WCEP *P*
- Allocate VREGs of that basic block $b \in P$ with most worst-case spill code executions to PHREGs using standard GC
- Recompute WCEP

Results – WCET$_{EST}$ Reductions



100% = WCET$_{EST}$ using Std. WCET-unaware Graph-Coloring

Other WCET-aware Optimizations

- WCET-aware Procedure Cloning & Positioning
  [Lokuciejewski, Falk et al., „*WCET-Driven, Code-Size Critical Procedure Cloning*“, SCOPES 2008.]
  [Lokuciejewski, Falk et al., „*WCET-driven Cache-based Procedure Positioning Optimizations*“, ECRTS 2008.]
- WCET-aware I-Cache Locking
  [Falk, Plazar et al., „*Compile Time Decided Instruction Cache Locking Using Worst-Case Execution Paths*”, CODES+ISSS 2007.]
- WCET-aware Scratchpad Memory Allocation
  - First simple approaches proposed by I. Puaut et al.
  - Integrated ILP strategies under development at Dortmund, supported by PREDATOR

Future work

- Tight integration of tools and representation of memory architectures
- Making these optimizations available to the “average” software engineer for embedded systems
- Creating comprehensive set of SPM optimizations in the form of pre-pass optimizations
- Extensions focusing on multi-processor based systems
- Analysis of standard optimizations from the viewpoint of WCET reduction
- Analysis of tradeoffs between multiple objectives

Conclusion

- Current compiler technology does not reflect non-uniform memory access costs well
- ☞ Proposal for an introduction of optimizations driven by models of memory access costs
- ☞ First approaches focus on exploitation of scratch-pad memories
    - ☞ Non-overlaying + overlaying approaches
    - ☞ Single + multiple applications
- ☞ Optimizations driven by an explicit WCET model
    - ☞ Significant WCET reductions even in well-established areas