UNIVERSITÄT DORTMUND

Memory-architecture aware compilation

Peter Marwedel
University of Dortmund
Informatik 12
44221 Dortmund, Germany

peter.marwedel@udo.edu

http://ls12-www.cs.uni-dortmund.de
phone: +49 (231) 755 6111
mobile: +49 (173) 5165815
fax: +49 (231) 755 6116

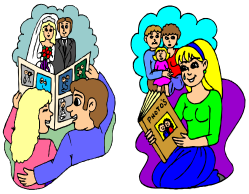
UNIVERSITÄT DORTMUND

Introduction

Universität Dortmund

The Problem with Memories

Memories?



Oops!
Memories!

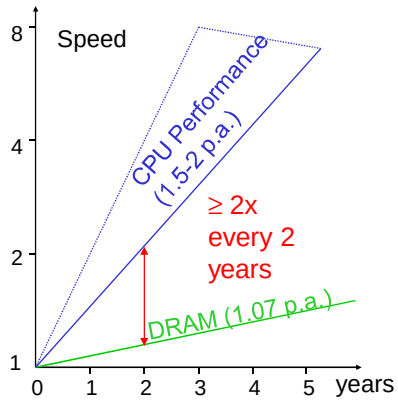
1. Increasing speed gap
2. Major consumer of electrical energy
3. Timing predictability difficult to achieve
4. …

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

Universität Dortmund

Trends for the speeds

Speed gap between processor and main DRAM increases



Similar problems also for embedded systems & MPSoCs

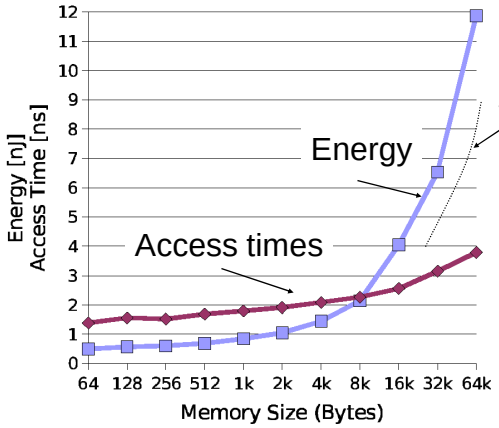
☞ In the future:
Memory access times >> processor cycle times

☞ "Memory wall" problem

References integrated into slides for modularity & context

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

Dependency on the size

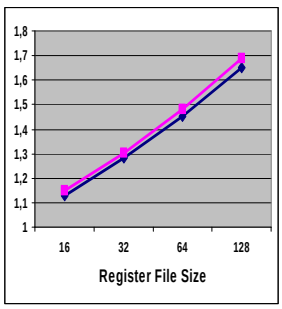


Applications are getting larger and larger …

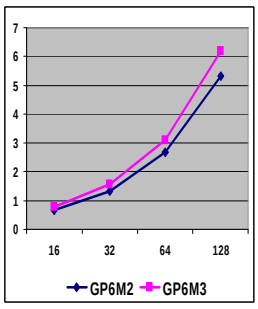
+ locality of references
☞ Memory hierarchies

„Everything“ is large for large memories

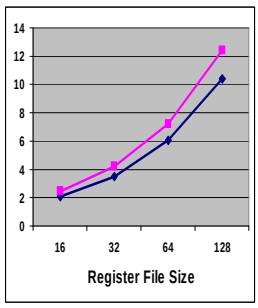
Cycle Time (ns)*



Area ($\lambda^2 x 10^6$)



Power (W)

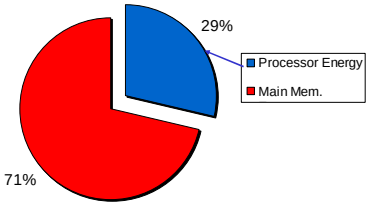


* Monolithic register file; Rixner's et al. model [HPCA'00], Technology of 0.18 mm; VLIW configurations for a certain number of ports („GPxMyREGz where: x={6}, y={2, 3} and z={16, 32, 64, 128"}; Based on slide by and ©: Harry Valero, U. Barcelona, 2001

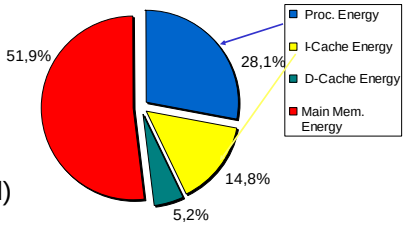
Universität Dortmund

Memory system frequently consumes >50 % of the energy used for processing

Cache ($)-less monoprocessor



Multiprocessor with cache ($)

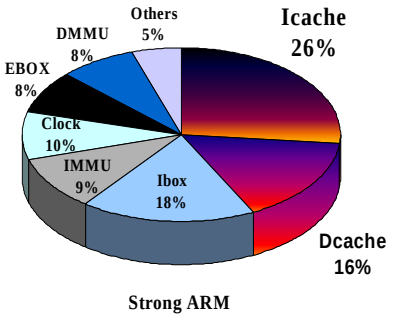


Average over 200 benchmarks analyzed by Verma (U. Dortmund)

[M. Verma, P. Marwedel: Advanced Memory Optimization Techniques for Low-Power Embedded Processors, Springer, 2007]

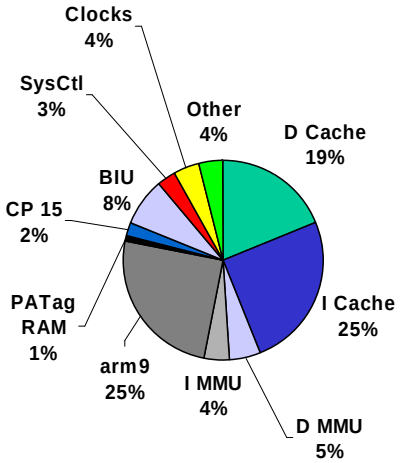
Universität Dortmund

Similar information according to other sources



Strong ARM

IEEE Journal of SSC Nov. 96

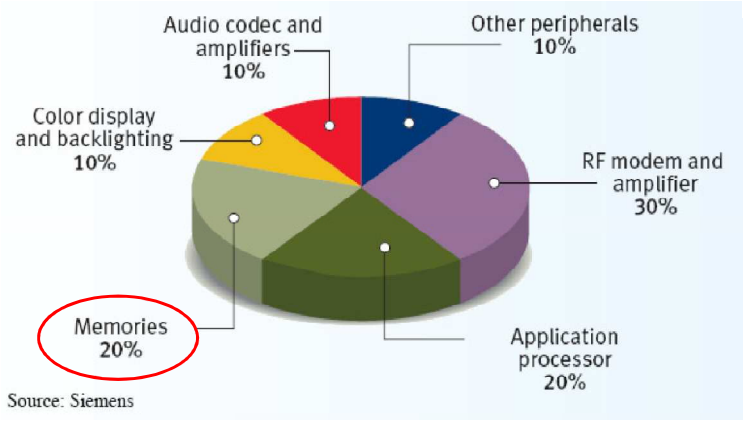


[Based on slide by and ©: Osman S. Unsal, Israel Koren, C. Mani Krishna, Csaba Andras Moritz, U. of Massachusetts, Amherst, 2001]

[Segars 01 according to Vahid@ISSS01]

Universität Dortmund

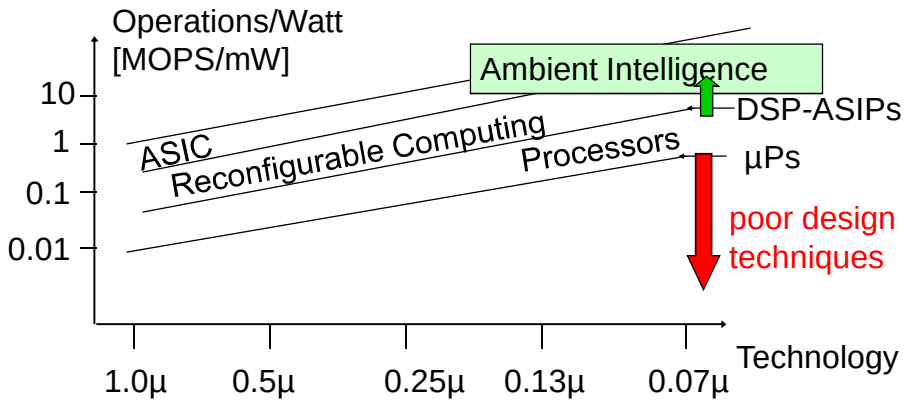
Energy consumption in mobile devices



[O. Vargas (Infineon Technologies): Minimum power consumption in mobile-phone memory subsystems; Pennwell Portable Design - September 2005;] Thanks to Thorsten Koch (Nokia/ Univ. Dortmund) for providing this source.

Universität Dortmund

Importance of Energy Efficiency



Necessary to optimize; otherwise the price for flexibility cannot be paid!

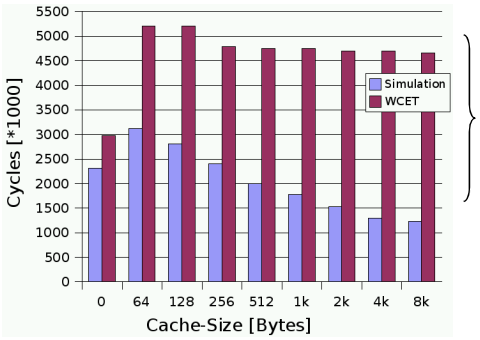
[H. de Man, Keynote, DATE'02; T. Claasen, ISSCC99]

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

☞ Multiple Objectives for Memory System Design

- (Average) Performance
  - Throughput
  - Latency
- Energy consumption
- Predictability, good worst case execution time bound (WCET)
- Size
- Cost
- ….





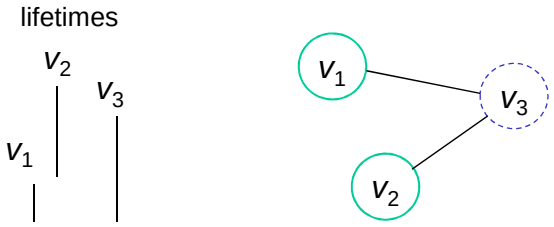

Universität Dortmund

Register allocation

- Registers = fastest level in the memory hierarchy

☞ Interest in good global register allocation techniques

- Frequently based on coloring of interference graph



- Registers neither suitable for |objects| » single words nor for code

☞ Register allocation not considered in the remainder

Universität Dortmund

Outline

- Introduction, Motivation
- ➡ Scratch-pad memories
  - Characteristics
  - Non-overlaying allocation, partitions, stack, heap, WCET
  - Overlaying (dynamic) allocation
  - Multiple processes & processors
- Caches
  - Predictability, loop transformations
  - Prefetching, layout transformations
- Other levels of the memory hierarchy
  - Array folding
  - ADL-driven compilation for hierarchies
  - Rewriting algorithms for memory hierarchies
- Conclusion, Additional references

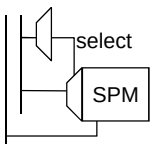
UNIVERSITÄT DORTMUND

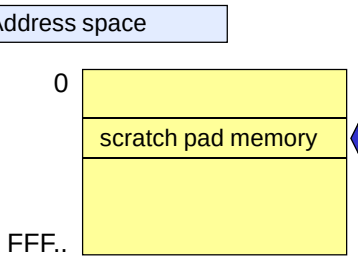
Scratch pad memories
- Characteristics
- Compilation (1)

Scratch pad memories (SPM): Fast, energy-efficient, timing-predictable

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

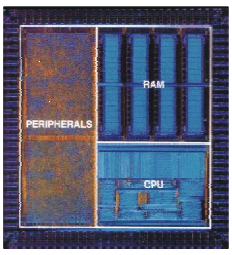
Selection is by an appropriate address decoder (simple!)





Example

ARM7TDMI cores, well-known for low power consumption



Universität Dortmund

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]

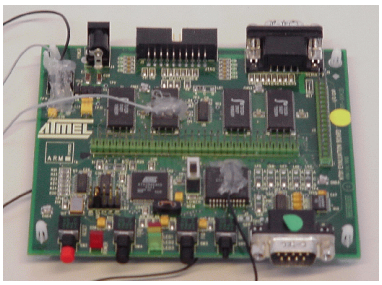
*… In essence, we must reinvent computer science. Fortunately, we have quite a bit of knowledge and experience to draw upon. Architecture techniques such as **software-managed caches** promise to deliver much of the benefit of memory hierarchy without the timing unpredictability.*

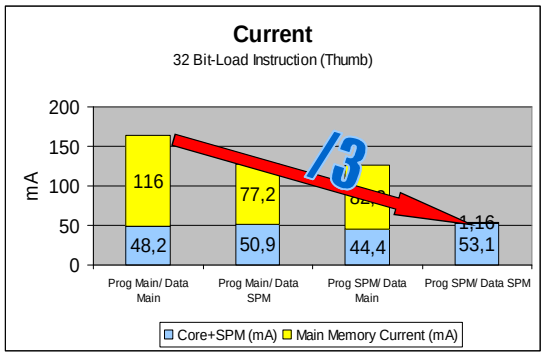
[Ed Lee: Absolutely Positively on Time: What would it take?, *IEEE Computer*, 2005]

Universität Dortmund

Comparison of currents using measurements

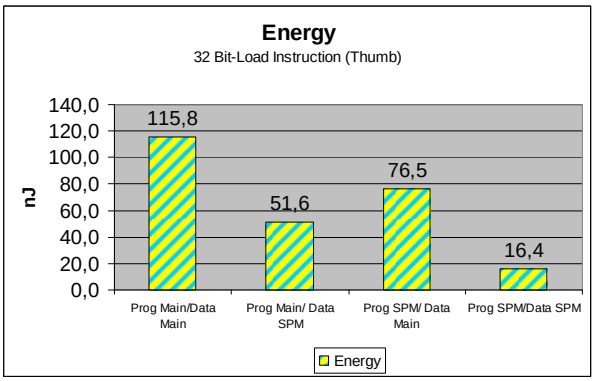
E.g.: ATMEL board with ARM7TDMI and ext. SRAM





Scratchpad vs. main memory energy

Example: Atmel ARM-Evaluation board



Savings (86%) even larger.

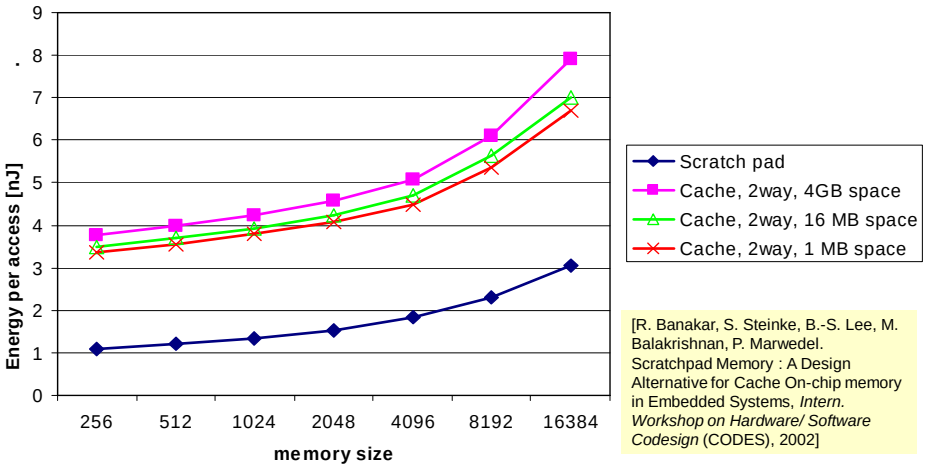
energy reduction: / 7.06

100% predictable

€

Why not just use a cache ?

Energy consumption in tags, comparators and muxes is significant.

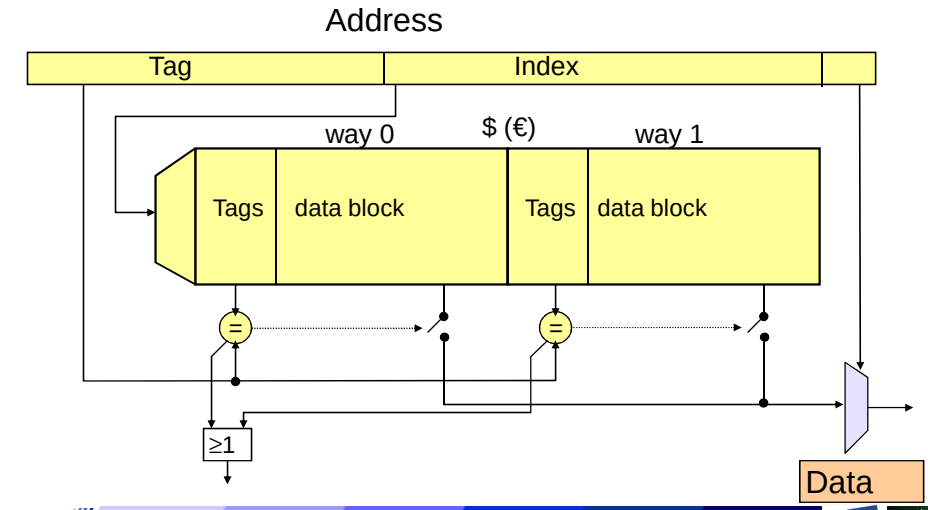


[R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, P. Marwedel. Scratchpad Memory : A Design Alternative for Cache On-chip memory in Embedded Systems, *Intern. Workshop on Hardware/ Software Codesign* (CODES), 2002]

Universität Dortmund

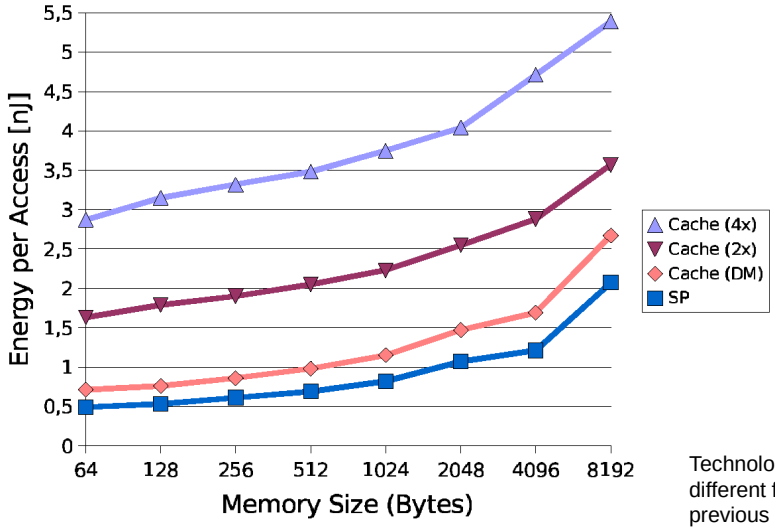
Set-associative cache
n-way cache

n = 2



Universität Dortmund

Influence of the associativity



Technology different from previous slide.

Availability of SPMs=("Tightly Coupled Memories")

ARM CPU Core	Caches	TCM Available
ARM 1026EJ-S	Variable	yes
ARM 1136J(F)-S	Variable	yes
ARM 1176JZ(F)-S	Variable	yes
ARM 926EJ-S	Variable	yes
ARM 1026EJ-S	Variable	yes
ARM 1156T2(F)-S	Variable	yes
ARM 946E-S	Variable	yes
ARM 966E-S	-	yes
ARM 968E-S	-	yes
All others		no

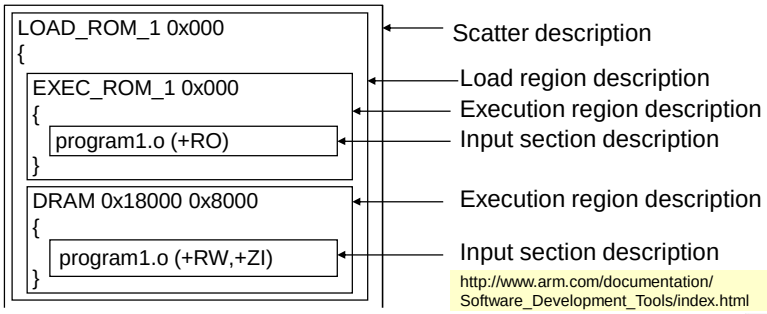
Source: http://www.arm.com/products/CPUs/core_selector.html

Current usage for ARM

1. **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
```

2. **Input scatter loading file to linker for allocating section to specific address range**



(2 different examples)

http://www.arm.com/documentation/Software_Development_Tools/index.html

Universität Dortmund

Popular among designers?

Information received from designers:

- Used for buffering speech in mobile phones (mobile phone company)
- Manual mapping of frequently accessed data
- Essentially no idea on how to exploit it (WLAN/Bluetooth specialists & major vendor)

Why not change this situation?

Universität Dortmund

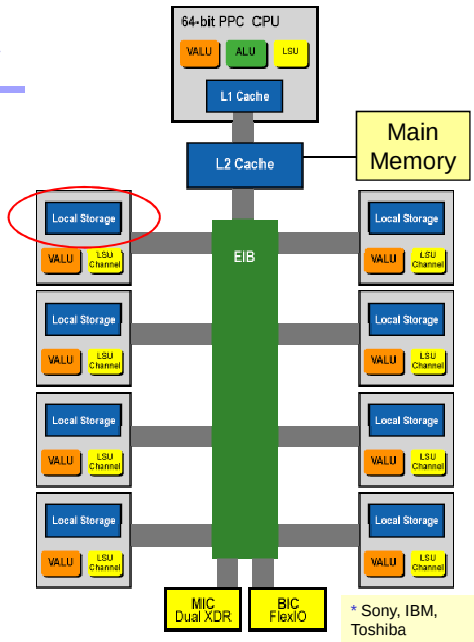
Similar Concept for the Cell processor *

Local SPE processors fetch instructions and data from local storage LS (256 kB). LS **not** designed as a cache. Separate DMA transfers required to fill and spill.

Motivation same as for this tutorial:

- Large memory latency
- Huge overhead for automatically managed caches

Similar for Infineon TriCore

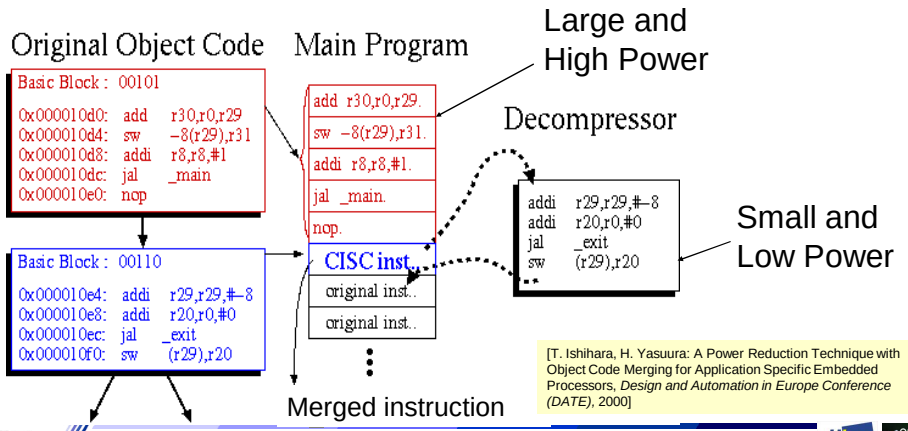


* Sony, IBM, Toshiba

Universität Dortmund

Migration of instructions to small memory Yasuura's architecture (Kyushu U.)

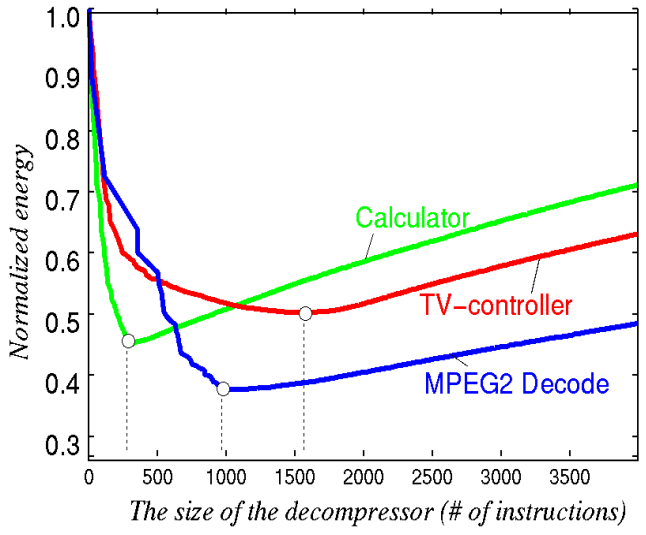
2 instruction memories: main memory + "decompressor" memory;
Compiler optimizing code allocation & size of the 2 instruction memories



[T. Ishihara, H. Yasuura: A Power Reduction Technique with Object Code Merging for Application Specific Embedded Processors, *Design and Automation in Europe Conference (DATE)*, 2000]

Universität Dortmund

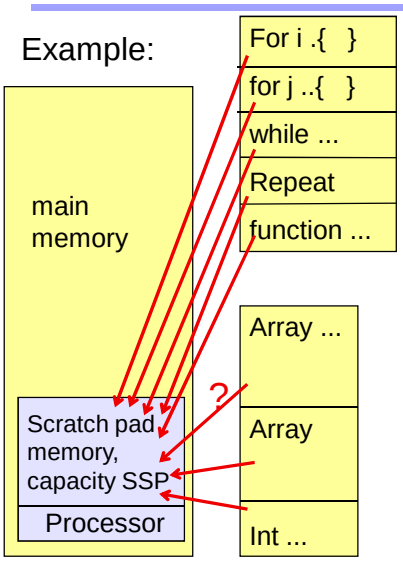
Energy savings



[Based on slide by & © : H. Yasuura, 2000]

Universität Dortmund

Migration of data and instructions, global optimization model (U. Dortmund)



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

Universität Dortmund

IP representation - migrating functions and variables-

Symbols:

$S(var_k)$ = size of variable k

n_k = number of accesses to variable k

$\Delta e(var_k)$ = energy ***saved*** per variable access, if var_k is migrated

$\Delta E(var_k)$ = energy saved if variable var_k is migrated (= $\Delta e(var_k)\ n(var_k)$)

$x(var_k)$ = decision variable, =1 if variable k is migrated to SPM,
=0 otherwise

K = set of variables

Similar for functions I

Integer programming formulation:

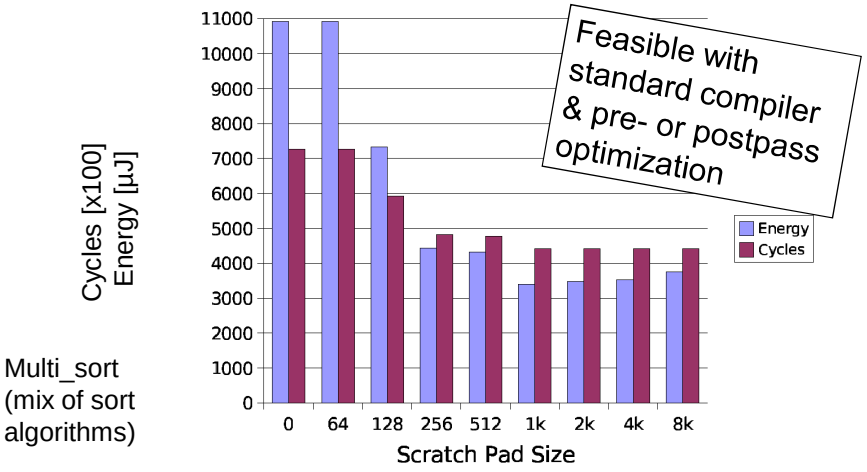
Maximize $\sum_{k \in K} x(var_k)\ \Delta E(var_k) + \sum_{i \in I} x(F_i)\ \Delta E(F_i)$

Subject to the constraint

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

Universität Dortmund

Reduction in energy and average run-time



Multi_sort (mix of sort algorithms)

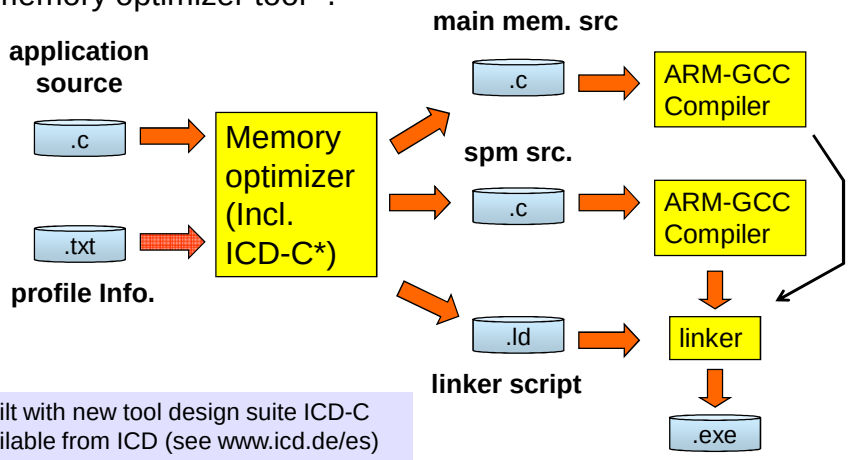
Measured processor / external memory energy + CACTI values for SPM (combined model)

Numbers will change with technology, algorithms remain unchanged.

Universität Dortmund

Using these ideas with an gcc-based tool flow

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



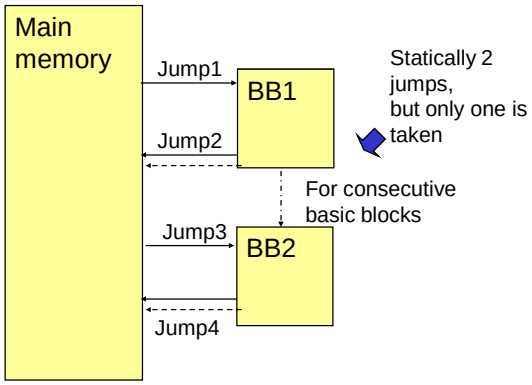
*Built with new tool design suite ICD-C available from ICD (see www.icd.de/es)

Universität Dortmund

Allocation of basic blocks

Fine-grained granularity smoothens dependency on the size of the scratch pad.

Requires additional jump instructions to return to "main" memory.



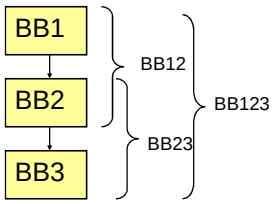
Universität Dortmund

Taking consecutive basic blocks into account

Approach:

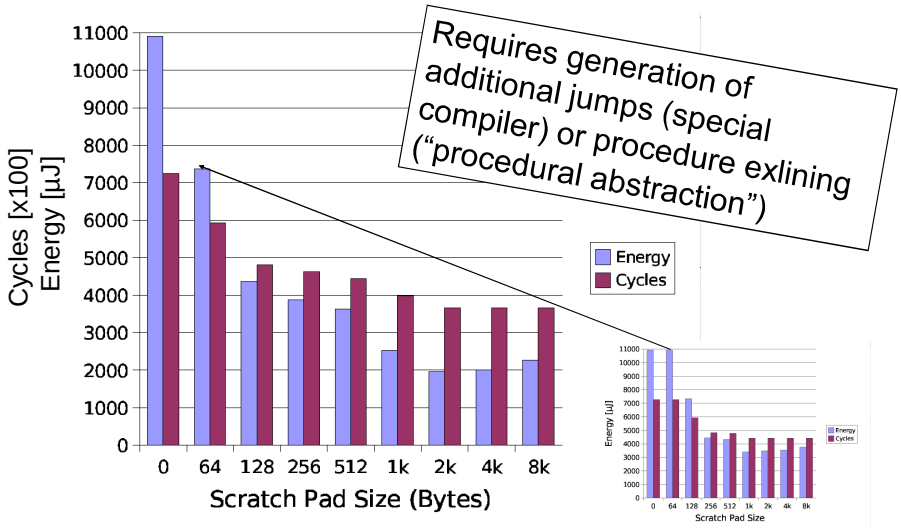
- Consider sets of consecutive BBs as a new kind of basic blocks (“multi blocks”)
- Add a constraint preventing the same block from being selected twice:
  $x(BB_b) + x(F_i) + \sum_{j \in multiblocks(b),\, j \neq x} x(BB_j) \leq 1$
  $\forall\, b \in \{blocks\} \cup \{\text{multi blocks}\}$

☞ Block *b* is either moved individually, as part of a function, as part of one of its enclosing multi-blocks or not at all.



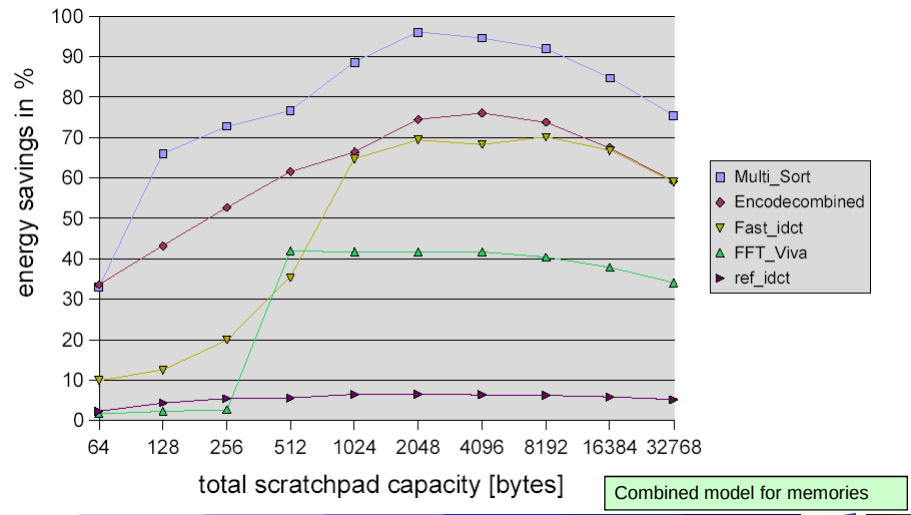
Universität Dortmund

Allocation of basic blocks, sets of adjacent basic blocks and the entire stack



Universität Dortmund

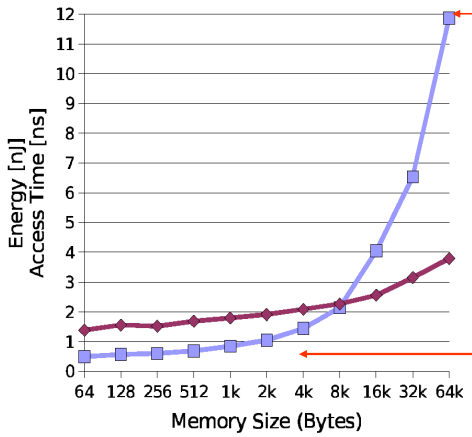
Savings for memory system energy alone



Combined model for memories

Universität Dortmund

Underlying reasons for large energy savings



Potential energy saving is determined by the difference between the energy consumption for large and small memories.

In this case:

factor of ≈ 20

☞ savings of up to 95%

Can even be larger for larger memories

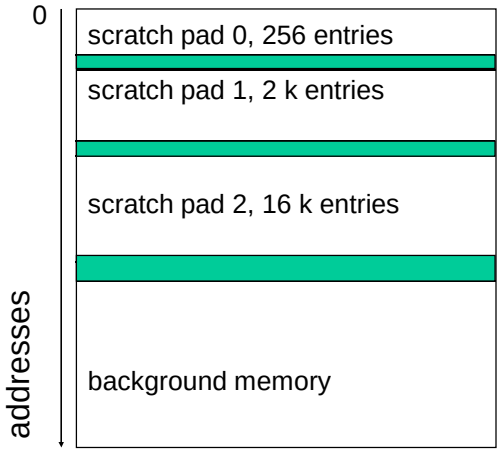
Universität Dortmund

Partitioning

Small is beautiful:

One small SPM is beautiful.

May be, several smaller SPMs are even more beautiful?



Considered partitions

# of partitions	number of partitions of size:						
	4k	2k	1k	512	256	128	64
7	0	1	1	1	1	1	2
6	0	1	1	1	1	2	0
5	0	1	1	1	2	0	0
4	0	1	1	2	0	0	0
3	0	1	2	0	0	0	0
2	0	2	0	0	0	0	0
1	1	0	0	0	0	0	0

Example of all considered memory partitions for a total capacity of 4096 bytes

Optimization for multiple memories

Minimize $C = \sum_j \left(e_j \cdot \sum_i x_{j,i} \cdot n_i \right)$

With e_j: energy ***required*** per access to memory j,
and $x_{j,i}$= 1 if object i is mapped to memory j, =0 otherwise,
and n_i: number of accesses to memory object i,
subject to the constraints:

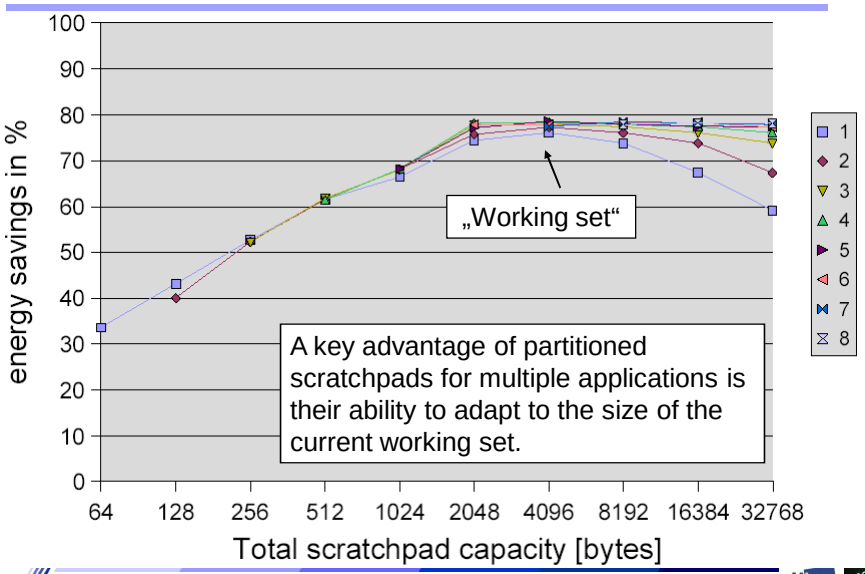
$$\forall j : \sum_i x_{j,i} \cdot S_i \leq SSP_j$$

$$\forall i : \sum_j x_{j,i} = 1$$

With S_i: size of memory object i,
SSP_j: size of memory j.

Main memory included as a special case of j

Results for parts of GSM coder/decoder



Summary for lecture 1

- Introduction: Problems with memories:
  - Too slow (☞ memory wall problem)
  - Too energy hungry
  - Not timing predictable
- Scratch pad memories: several advantages
  - Energy-efficient
  - Timing-predictable
  - Small
  - Available (but hardly any tools)
- Non-overlaying (“static”) allocation
  - IP models for allocation
    - Functions and variables
    - Fine grained model for basic blocks
    - Partitioning

UNIVERSITÄT DORTMUND

Compilation for scratch pad memories (2)

Outline

- Introduction, Motivation
- Scratch-pad memories
  - Characteristics
  - Non-overlaying allocation, partitions, stack, heap, WCET
  - Overlaying (dynamic) allocation
  - Multiple processes & processors
- Caches
  - Predictability, loop transformations
  - Prefetching, layout transformations
- Other levels of the memory hierarchy
  - Array folding
  - ADL-driven compilation for hierarchies
  - Rewriting algorithms for memory hierarchies
- Conclusion, Additional references

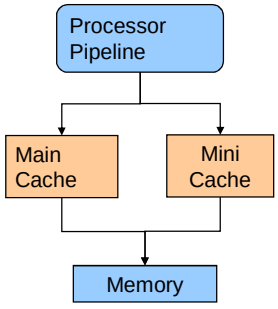
Universität Dortmund

Energy optimization in horizontally partitioned caches

Energy savings due to two effects
- Reduction in miss rate
- AccessEnergy (mini cache) < AccessEnergy (main cache)

Reduction in miss rate
- Aligned with performance
- Exploited by performance improvement techniques

Less Energy per Access in mini cache
- Inverse to performance
- *Energy can decrease even if there are more misses*
- *Opposite to performance optimization techniques!!*

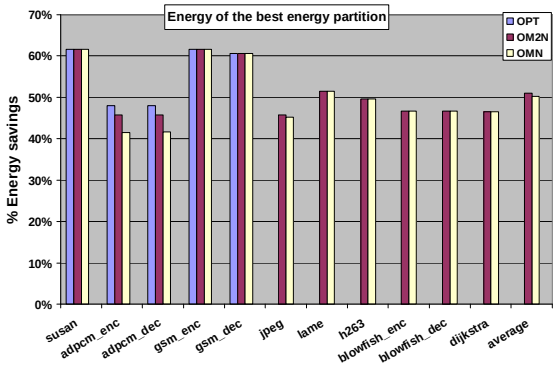


[A. Shrivastava, I. Issenin, N. Dutt: Compilation techniques for energy reduction in horizontally partitioned cache architectures, *Intern. Conf.on Compilers, Architectures and Synthesis For Embedded Systems* (CASES), 2005, pp. 90-96]

Universität Dortmund

Simple energy-oriented heuristic are good

$$\left(\frac{E_{base} - E_{\min}}{E_{base}}\right)$$



- Memory subsystem energy savings achieved by OMN (greedy)
- Base Configuration – All pages are mapped to main cache

OMN achieves 50% memory energy savings

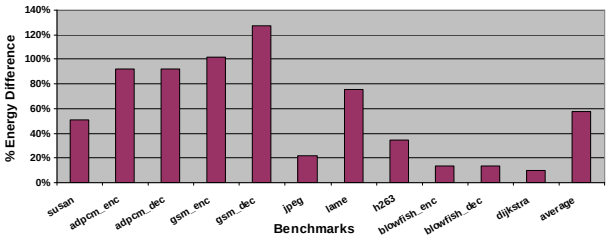
Universität Dortmund

Horizontally Partitioned Caches: Results

Plot

$$\left(\frac{E_{best\_runtime} - E_{best\_energy}}{E_{best\_energy}}\right)$$

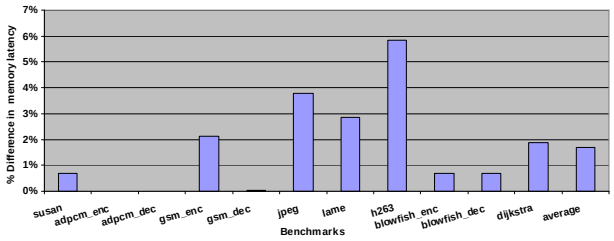
Best Performing Partition consumes 58% more Energy



Plot

$$\left(\frac{R_{best\_energy} - R_{best\_runtime}}{R_{best\_energy}}\right)$$

Best Energy Partition looses 2% Performance



Universität Dortmund

Stack allocation

1. Run stack size analyzer
   Use **stack as 1 large array** of returned size
   ☞ Mapped to SPM if small and frequently accessed
   [approach by Steinke et al. @ Dortmund]

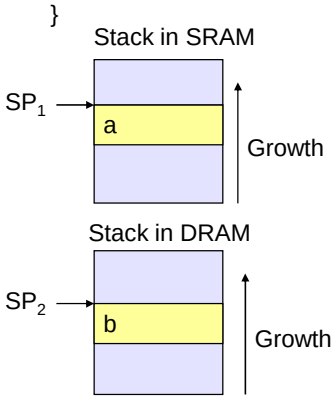
2. Potentially **map frequently & less frequently used elements to different memories**
   [approach by Barua et al.]

   Considers global & local variables (no instructions)

[O. Avissar, R. Barua, D. Stewart: An Optimal Memory Allocation Scheme for Scratch-Pad-Based Embedded Systems, *Transactions on Embedded Computing Systems*, 2002]

Split stack

```
foo()
{
  int a;
  float b;
  …
}
```



☞ 2 stack pointers (overhead)

1. Single stack frame per procedure (fast procedures)
2. Split frames (time consuming procedures)

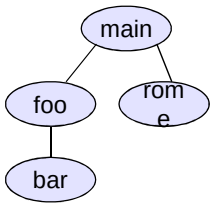
IP model for split stack

Cost function (linear):

Run-time = Access time for global variables +
Access time for stack variables

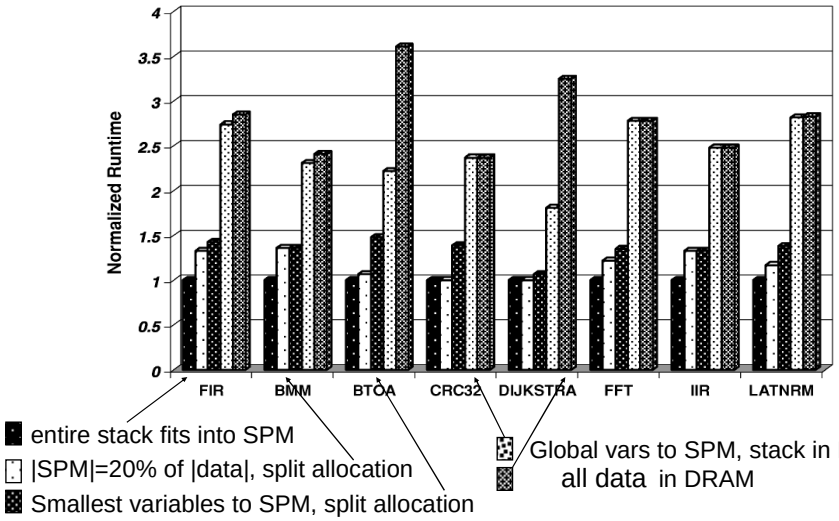
Constraints (linear):

- Each variable has to be mapped to 1 memory
- For all paths in the call graph:
  The sum of all objects mapped to SPM is ≤ |SPM|



Model actually considers an arbitrary number of memories, each with its own size and access time(s) ☞"partitioning".

Results



Heap allocation

1. Run heap size analyzer
   Use **heap as 1 large array** of returned size

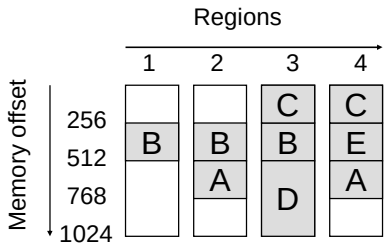
2. Potentially **map heap fragments to SPM**
   Problems:
   - Object sizes frequently not known at compile time
   - Avoid illegal references
   - Approaches using a level of indirection suffer from additional overhead

Universität Dortmund

Heap allocation by Barua

- Copying between memories is done at *region boundaries* (before & after loops, at procedure entry and exit).
- At each copy point, a *bin* is allocated in SPM for heap data.
- **Objects always have the same address within the *bin*.**

- Modified malloc first checks if space in *bin* is available, otherwise allocates in main RAM
- Modified free deallocates space in *bin*.



[A. Dominguez, S. Udayakumaran, R. Barua: Heap Data Allocation to Scratch-Pad Memory in Embedded Systems, *Journal of Embedded Computing*, 2005]

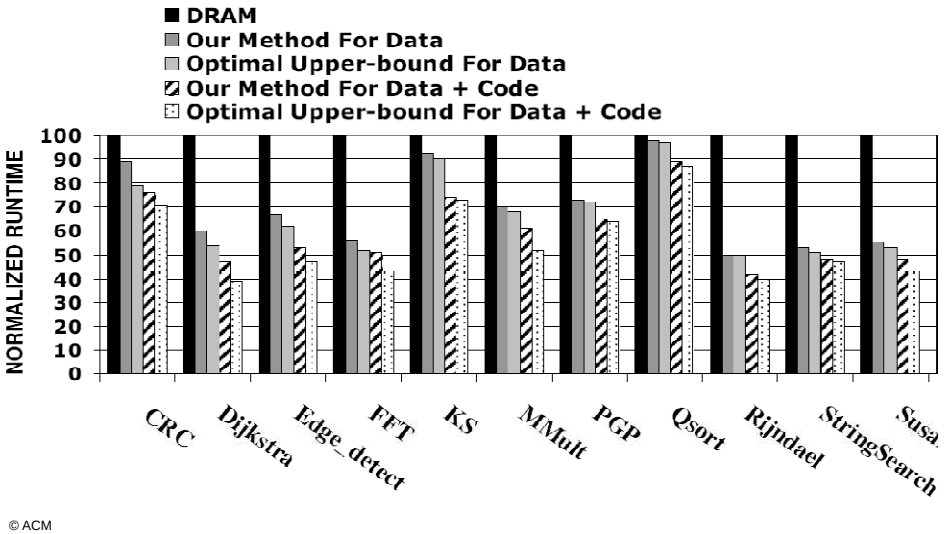
Universität Dortmund

Considering SPM size at link time

- Avoiding executables generated for a specific SPM size
- **Profiler** stores variables sorted by their "frequency per byte" (FPB) in executable
- **Compiler** identifies access to variables with unknown memory allocation by using global symbols.
- After loading the program, a custom **installer**
  - Reads SPM size of current architecture
  - Decides (using FPBs) which variables to put into SPM
  - Patches addresses of variables (incl. stack variables)
  - Tries to use SPM space not used on a calling path
  - Also moves code into SPM

[N. Nguyen, A. Dominguez, R. Barua: Memory Allocation for Embedded Systems with a Compile-Time-Unknown Scratch-Pad Size, *Intern. conf. on Compilers, architectures and synthesis for embedded systems* (CASES), 2005, p. 115-125]

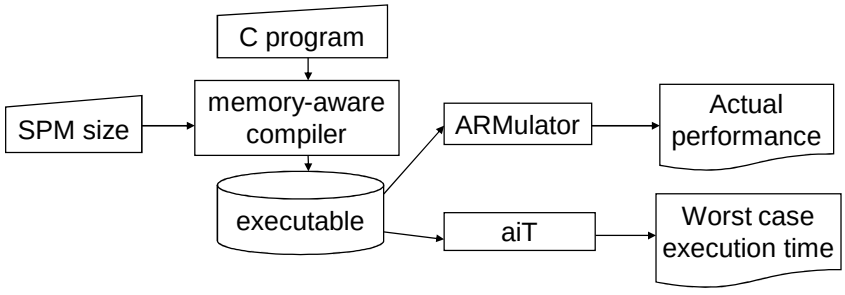
Comparison with code optimized for specific SPM-Size



Scratch-pad/tightly coupled memory based predictability

Pre run-time scheduling *is often the only practical means of providing predictability in a complex system.* [Xu, Parnas]

- ☞ 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



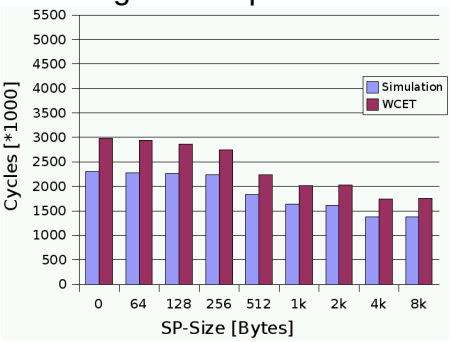
Architectures considered

ARM7TDMI with 3 different memory architectures:

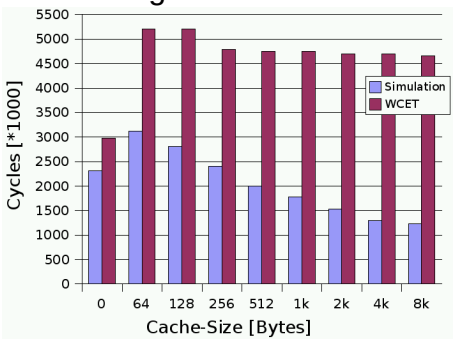
1. **Main memory**
   LDR-cycles: (CPU,IF,DF)=(3,2,2)
   STR-cycles: (2,2,2)
   * = (1,2,0)
2. **Main memory + unified cache**
   LDR-cycles: (CPU,IF,DF)=(3,12,6)
   STR-cycles: (2,12,3)
   * = (1,12,0)
3. **Main memory + scratch pad**
   LDR-cycles: (CPU,IF,DF)=(3,0,2)
   STR-cycles: (2,0,0)
   * =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

Results for G.721

Using Scratchpad:



Using Unified Cache:



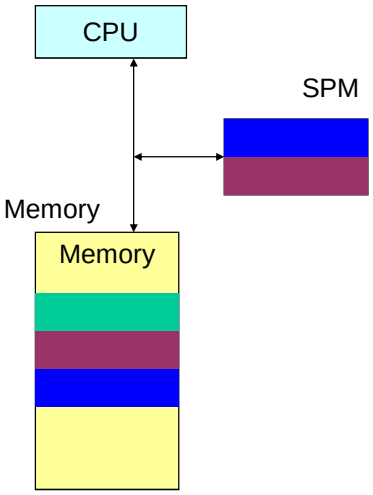
Yes, they are clearly more timing predictable!

- 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

Outline

- Introduction, Motivation
- Scratch-pad memories
  - Characteristics
  - Non-overlaying allocation, partitions, stack, heap, WCET
  - Overlaying (dynamic) allocation
  - Multiple processes & processors
- Caches
  - Predictability, loop transformations
  - Prefetching, layout transformations
- Other levels of the memory hierarchy
  - Array folding
  - ADL-driven compilation for hierarchies
  - Rewriting algorithms for memory hierarchies
- Conclusion, Additional references

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

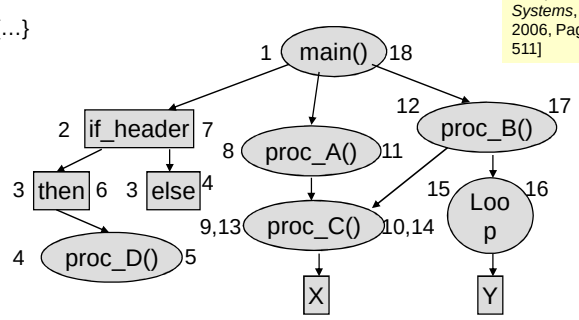
Overlaying allocation by Barua et al. (1)

Potential copying @ control points (major locality changes):
- Beginning and end of procedures
- Before and after loops
- Before and after if statements
- Beginning and end of then and else parts
- Before and after switch statements and their cases

Number control points!

[S. Udayakumaran, A. Dominguez, R. Barua: Dynamic Allocation for Scratch-Pad Memory using Compile-Time Decisions, *ACM Transactions in Embedded Computing Systems*, Vol. V, 2006, Pages 472 - 511]

```
main(){
  if(…) {proc-D()} else {…}
  proc-A()
  proc-B()
}
proc-A() {
  proc-C()
}
proc-B {
  proc-C()
  while(…) {Y=…}
}
proc-C() {X=…}
proc-D() {…}
```

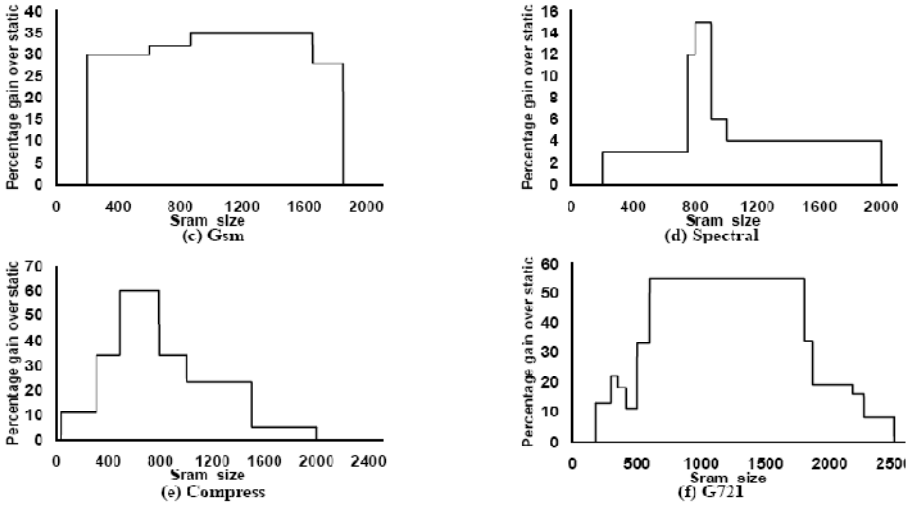


Overlaying allocation by Barua et al. (2)

- At 1st control point: Initialize SPM with variables accessed in first region in decreasing order of freq-per-byte
- Traverse control points according to control point # At each control point:
  - Compute variables that potentially should be swapped in, based on # of accesses per byte
  - Compute variables that should be swapped out
  - Compute variables that should remain
  - Compute addresses of variables to be moved in
  - Generate swap code

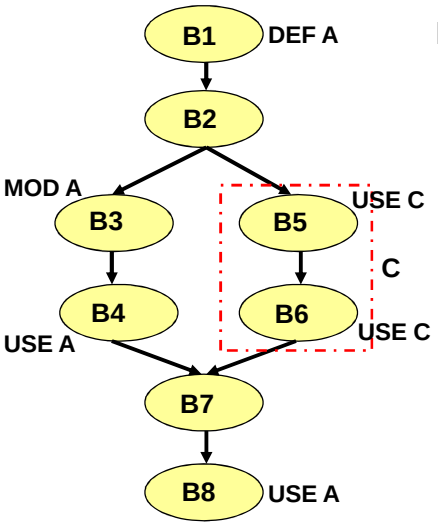
Universität Dortmund

Runtime reduction with respect to non-overlaying ("static") allocation



Universität Dortmund

Overlaying allocation by Verma et al. (1)

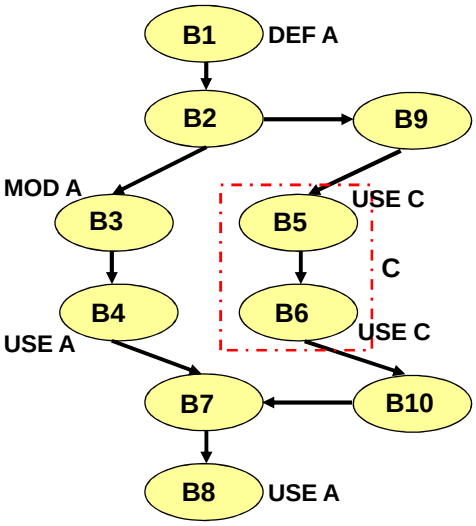


Based on control flow graph.

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

Universität Dortmund

Overlaying allocation by Verma et al. (2)



SPILL_STORE(A);
SPILL_LOAD(C);

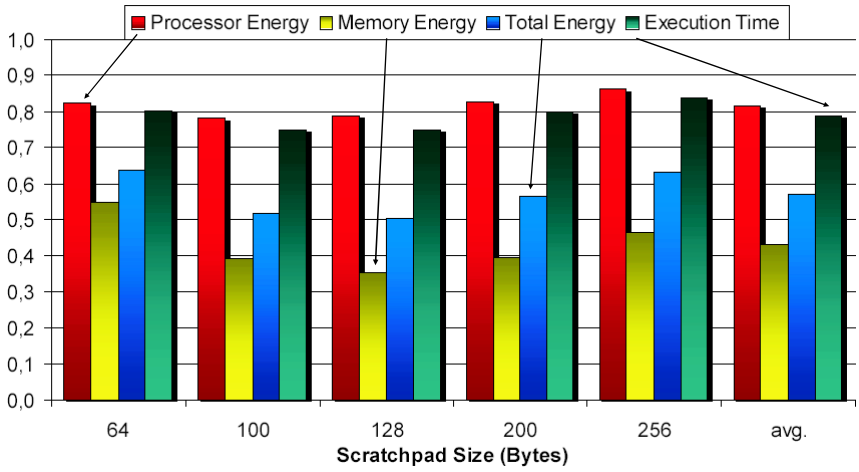
Global set of IP equations reflects cost/benefit relations of potential copy points

SPILL_LOAD(A);

Code handled like data

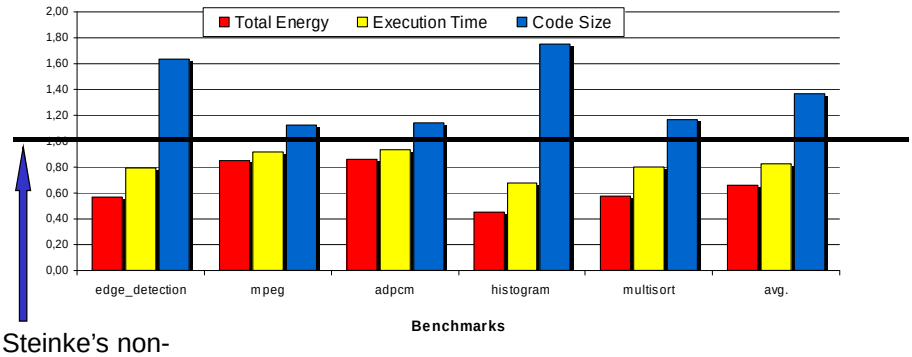
Universität Dortmund

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



Universität Dortmund

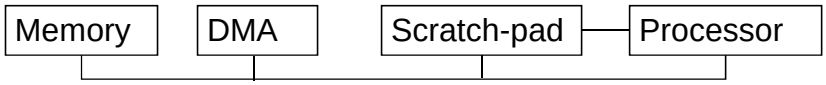
Overlaying allocation within scratch pad
- Energy, execution time, code size -



Steinke's non-overlaying Allocation

Universität Dortmund

Hardware-support for block-copying



The DMA unit was modeled in VHDL, simulated, synthesized. Unit only makes up 4% of the processor chip.

The unit can be put to sleep when it is unused.

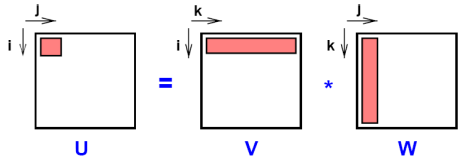
Code size reductions of up to 23% for a 256 byte SPM were determined using the DMA unit instead of the overlaying allocation that uses processor instructions for copying.

[Lars Wehmeyer, Peter Marwedel: Fast, Efficient and Predictable Memory Accesses, *Springer*, 2006]

Universität Dortmund

References to large arrays (1)
- Regular accesses -

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    for (k=0; k<n; k++)
      U[i][j]=U[i][j] + V[i][k] * W[k][j]
```



Tiling ☞

```
for (it=0; it<n; it=it+Sb)
  {read_tile V[it:it+Sb-1, 1:n]
  for (jt=0; jt<n; jt=jt+Sb)
    {read_tile U[it:it+Sb-1, jt:jt+Sb-1]
     read_tile W[1:n,jt:jt+Sb-1]
     U[it:it+Sb-1,jt:jt+Sb-1]=U[it:it+Sb-1,jt:jt+Sb-1]
                              + V[it:it+Sb-1,1:n]
                              * W [1:n, jt:jt+Sb-1]
        write_tile U[it:it+Sb-1,jt:jt+Sb-1]
  }}
```

[M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, A. Parikh: Dynamic Management of Scratch-Pad Memory Space, *DAC*, 2001, pp. 690-695]

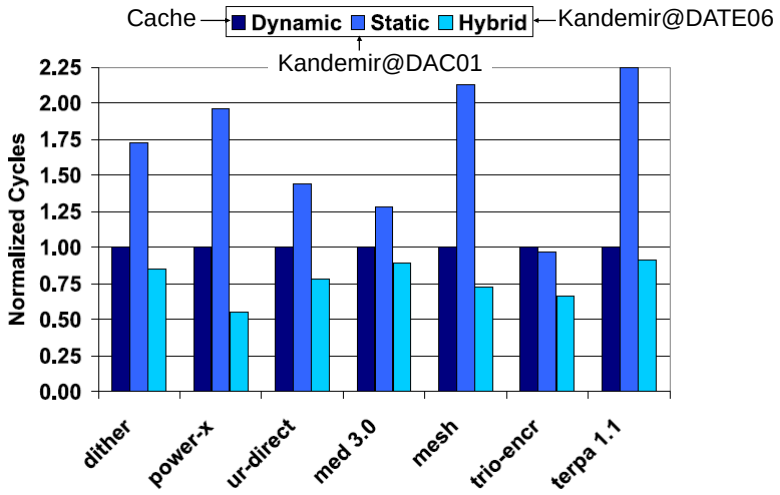
Universität Dortmund

References to large arrays
- Irregular accesses -

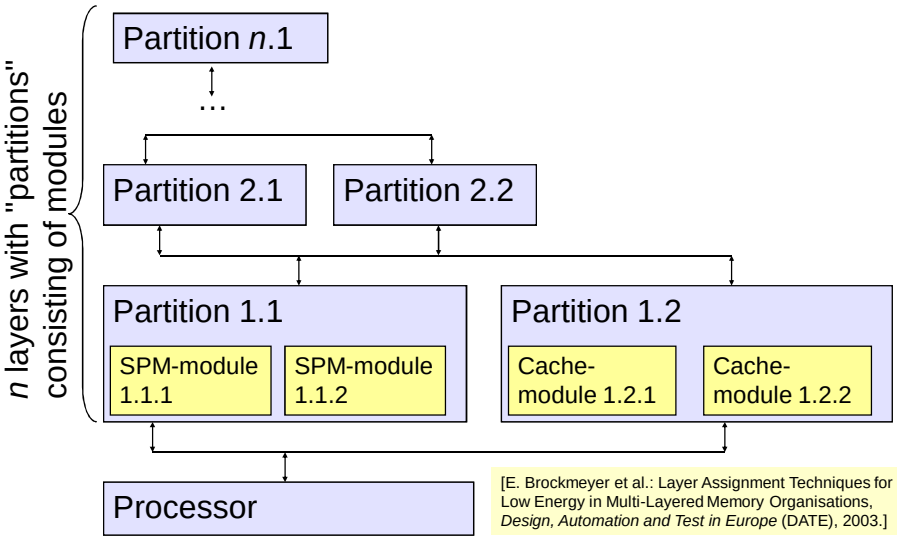
```
for each loop nest L in program P {
  apply loop tiling to L based on the access patterns of
    regular array references;
  for each assignment to index array X
    update the block minimum and maximum values of X;
  compute the set of array elements that are irregularly
    referenced in the current inter-tile iteration;
  compare the memory access costs for using
   and not using SPM;
  if (using SPM is beneficial)
    execute the intra-tile loop iterations by using the SPM
  else
    execute the intra-tile loop iterations by not
    using the SPM
}
```

[G. Chen, O. Ozturk, M. Kandemir, M. Karakoy: Dynamic Scratch-Pad Memory Management for Irregular Array Access Patterns, *DATE*, 2006]

Results for irregular approach



Hierarchical memories: Memory hierarchy layer assignment (MHLA) (IMEC)



[E. Brockmeyer et al.: Layer Assignment Techniques for Low Energy in Multi-Layered Memory Organisations, *Design, Automation and Test in Europe* (DATE), 2003.]

Universität Dortmund

Memory hierarchy layer assignment (MHLA) - Copy candidates -

```
int A[250]
for (i=0; i<10; i++)
 for (j=0; j<10; j++)
  for (k=0; k<10; k++)
   for (l=0; l<10; l++)
    f(A[j*10+l])
size=0; reads(A)=10000
```

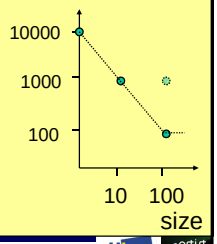
```
int A[250]
for (i=0; i<10; i++)
 for (j=0; j<10; j++)
 {A"[0..9]=A[j*10..j*10+9];
  for (k=0; k<10; k++)
   for (l=0; l<10; l++)
    f(A"[l])}
size=10; reads(A)=1000
```

Copy candidate

A', A" in small memory

```
int A[250]
for (i=0; i<10; i++)
 {A'[0..99]=A[0..99];
 for (j=0; j<10; j++)
  for (k=0; k<10; k++)
   for (l=0; l<10; l++)
    f(A'[j*10+l])}
size=100;reads(A)=1000
```

```
int A[250]
A'[0..99]=A[0..99];
for (i=0; i<10; i++)
 for (j=0; j<10; j++)
  for (k=0; k<10; k++)
   for (l=0; l<10; l++)
    f(A'[j*10+l])
size=100; reads(A)=100
```

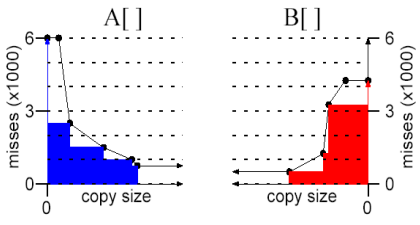


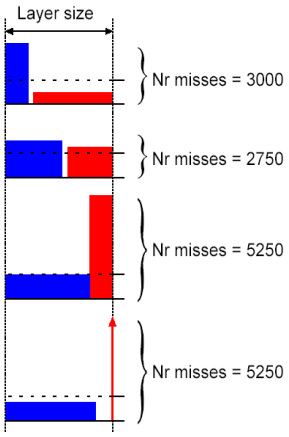
Universität Dortmund

Memory hierarchy layer assignment (MHLA) - Goal -

Goal: For each variable: find permanent layer, partition and module & select copy candidates such that energy is minimized.

Conflicts between variables





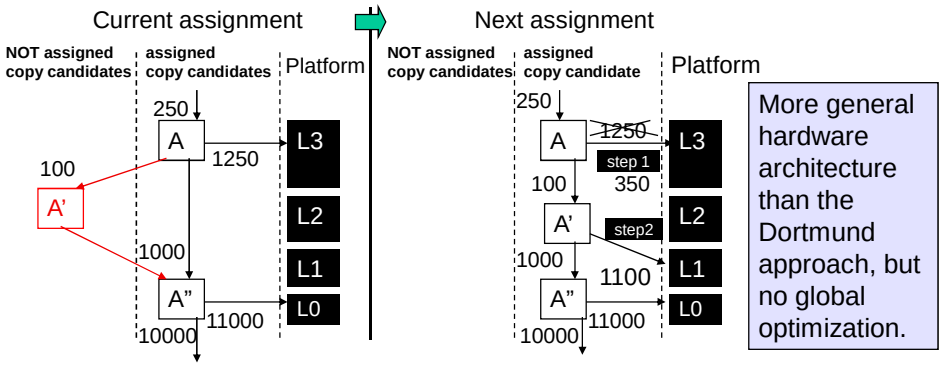
[E. Brockmeyer et al.: Layer Assignment Techniques for Low Energy in Multi-Layered Memory Organisations, *Design, Automation and Test in Europe* (DATE), 2003.]

Universität Dortmund

Memory hierarchy layer assignment (MHLA) - Approach -

Approach:
- start with initial variable allocation
- incrementally improve initial solution

such that total energy is minimized.



More general hardware architecture than the Dortmund approach, but no global optimization.

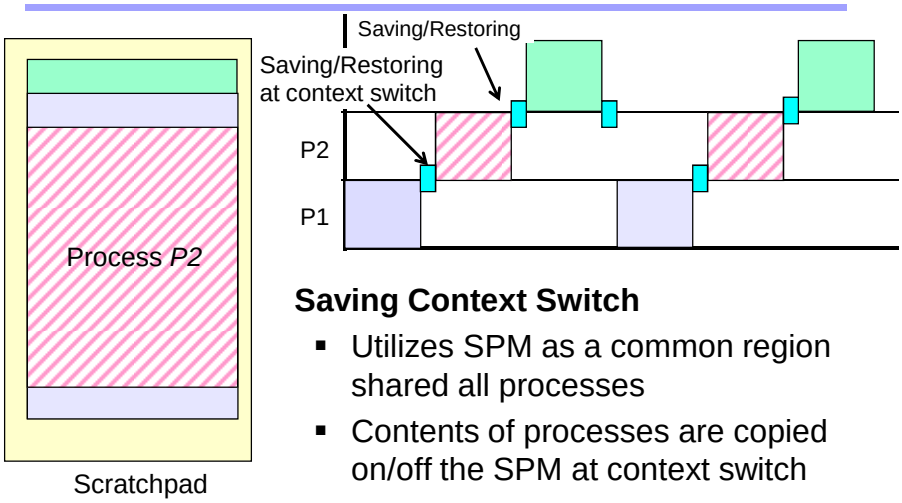
Universität Dortmund

Outline

- Introduction, Motivation
- Scratch-pad memories
  - Characteristics
  - Non-overlaying allocation, partitions, stack, heap, WCET
  - Overlaying (dynamic) allocation
  - Multiple processes & processors
- Caches
  - Predictability, loop transformations
  - Prefetching, layout transformations
- Other levels of the memory hierarchy
  - Array folding
  - ADL-driven compilation for hierarchies
  - Rewriting algorithms for memory hierarchies
- Conclusion, Additional references

Universität Dortmund

Static set of multiple applications: Saving/Restoring Context Switch

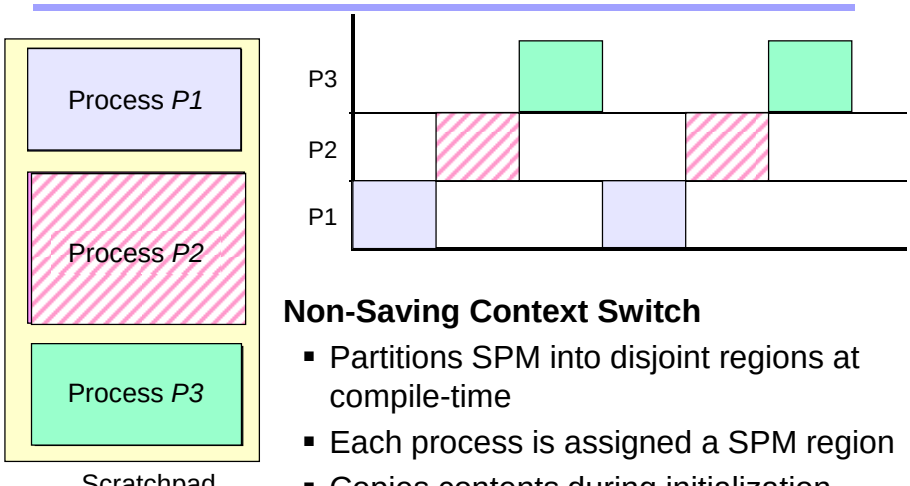


Saving Context Switch

- Utilizes SPM as a common region shared all processes
- Contents of processes are copied on/off the SPM at context switch
- Good for small scratchpads

Universität Dortmund

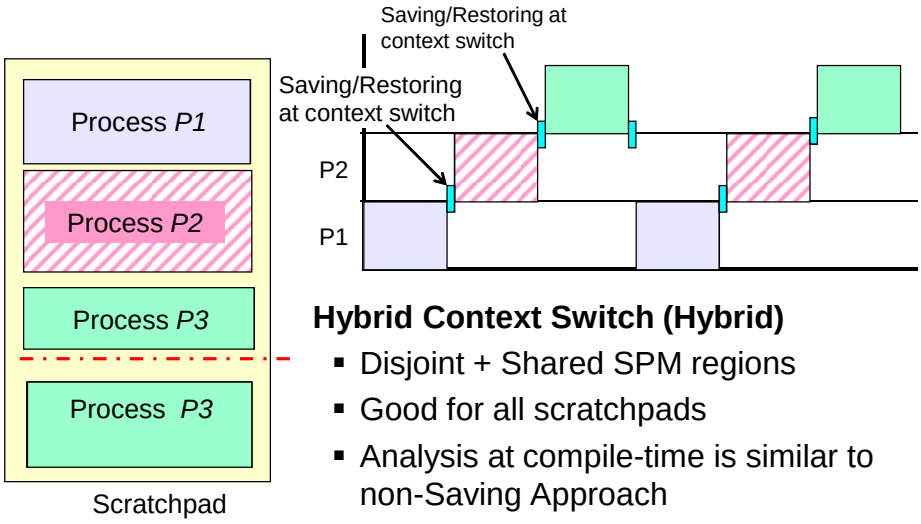
Non-Saving Context Switch



Non-Saving Context Switch

- Partitions SPM into disjoint regions at compile-time
- Each process is assigned a SPM region
- Copies contents during initialization
- Good for large scratchpads

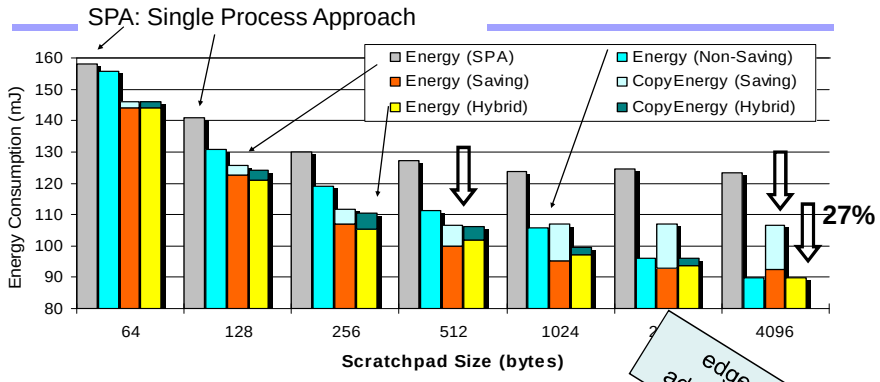
Hybrid Context Switch



Hybrid Context Switch (Hybrid)

- Disjoint + Shared SPM regions
- Good for all scratchpads
- Analysis at compile-time is similar to non-Saving Approach

Multi-process Scratchpad Allocation: Results



- For small SPMs (64B-512B) Saving is better
- For large SPMs (1kB- 4kB) Non-Saving is better
- Hybrid is the best for all SPM sizes.
- Energy reduction @ 4kB SPM is 27% for Hybrid approach

Summary lecture 2

- Non-Overlaying allocation
  - Stack allocation
  - Heap allocation
  - Timing predictability
- Overlaying SPM allocation
  - Single process
    - Barua's call graph approach
    - Verma's CDFG-based approach
    - Kandemir's tiling approach
    - IMEC's multiple levels
  - Multiple processes
    - Dortmund approach: saving, non-saving, hybrid

Compilation for
- Scratch pad memories (3)
- Caches

Outline

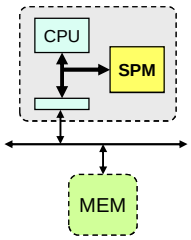
- Introduction, Motivation
- Scratch-pad memories
  - Characteristics
  - Non-overlaying allocation, partitions, stack, heap, WCET
  - Overlaying (dynamic) allocation
  - Multiple processes & processors
- Caches
  - Predictability, loop transformations
  - Prefetching, layout transformations
- Other levels of the memory hierarchy
  - Array folding
  - ADL-driven compilation for hierarchies
  - Rewriting algorithms for memory hierarchies
- Conclusion, Additional references

Dynamic set of multiple applications

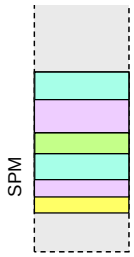
Compile-time partitioning of SPM no longer feasible

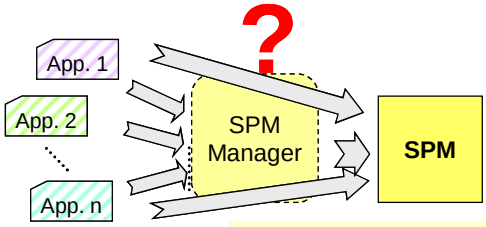
☞Introduction of SPM-manager

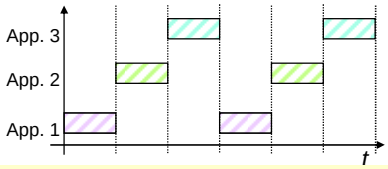
- Runtime decisions, but compile-time supported



Address space:



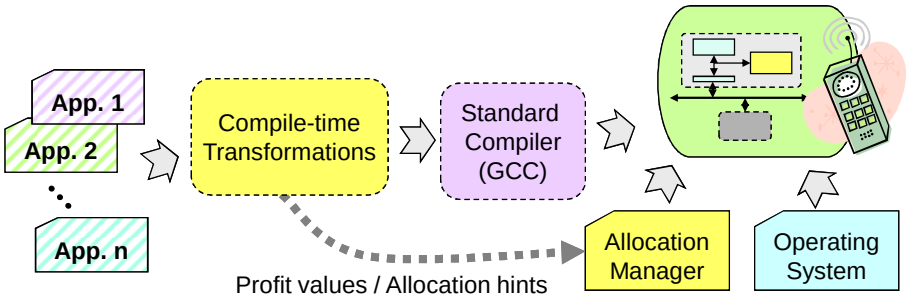




[R. Pyka, Ch. Faßbach, M. Verma, H. Falk, P. Marwedel: Operating system integrated energy aware scratchpad allocation strategies for multi-process applications, *SCOPES*, 2007]

Approach overview

- 2 steps: compile-time analysis & runtime decisions
- No need to know all applications at compile-time
- Capable of managing runtime allocated memory objects
- Integrates into an embedded operating system

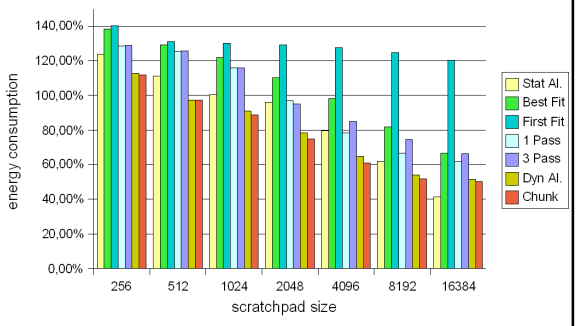


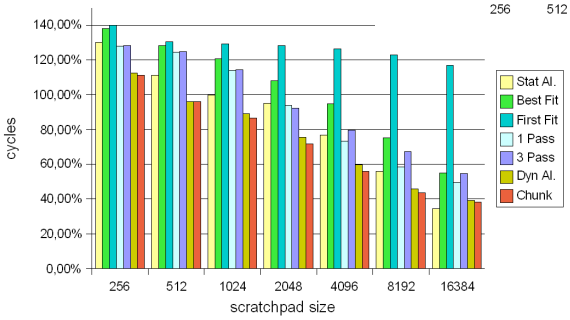
Using MPArm simulator from U. Bologna

Results

MEDIA+ Energy

- **Baseline: Main memory only**
- **Best: Static for 16k → 58%**
- **Overall best: Chunk → 49%**





MEDIA+ Cycles

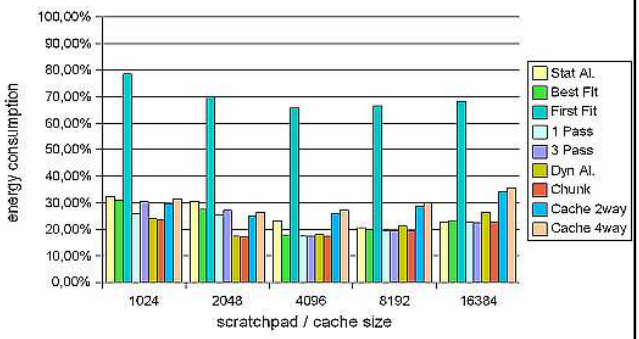
- **Baseline: Main memory only**
- **Best: Static for 16k → 65%**
- **Overall best: Chunk → 61%**

Comparison of SPMM to Caches for SORT

- Baseline: Main memory only
- SPMM peak energy reduction by 83% at 4k Bytes scratchpad
- Cache peak: 75% at 2k 2-way cache
- SPMM capable of outperforming caches
- OS and libraries are not considered yet

Chunk allocation results:

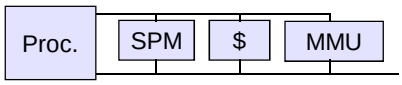
SPM Size	Δ 4-way
1024	74,81%
2048	65,35%
4096	64,39%
8192	65,64%
16384	63,73%



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.

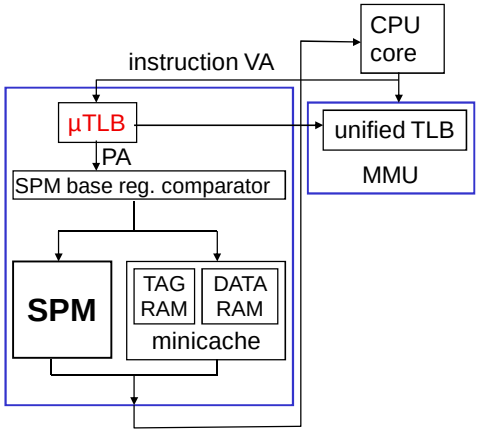


[B. Egger, J. Lee, H. Shin: Scratchpad memory management for portable systems with a memory management unit, *CASES*, 2006, p. 321-330 (best paper)]

Universität Dortmund

SPM+MMU (2)

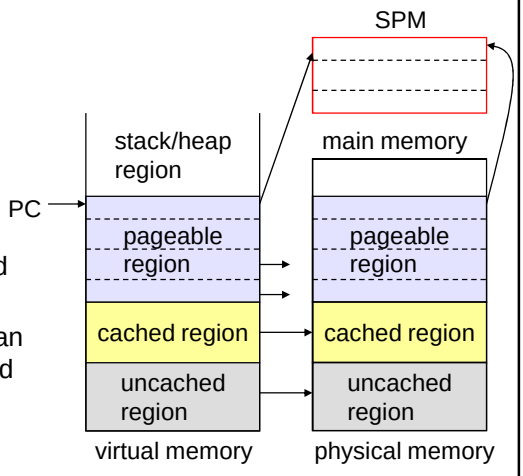
- µTLB generates physical address in 1 cycle
- if address corresponds to SPM, it is used
- otherwise, mini-cache is accessed
- Mini-cache provides reasonable performance for non-optimized code
- µTLB miss triggers main TLB/MMU
- SPM is used only for instructions
- instructions are stored in pages
- pages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



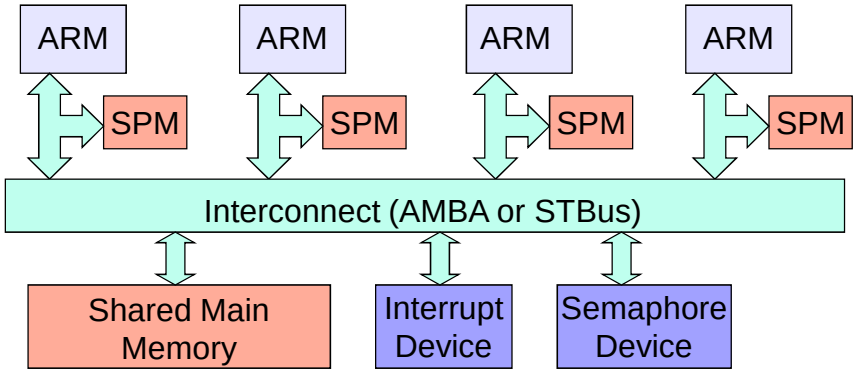
Universität Dortmund

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

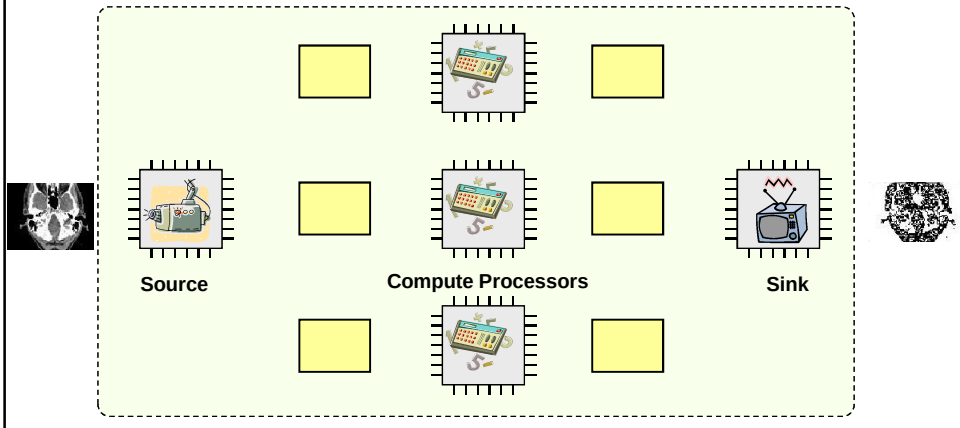


Multi-processor ARM (MPARM) Framework



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

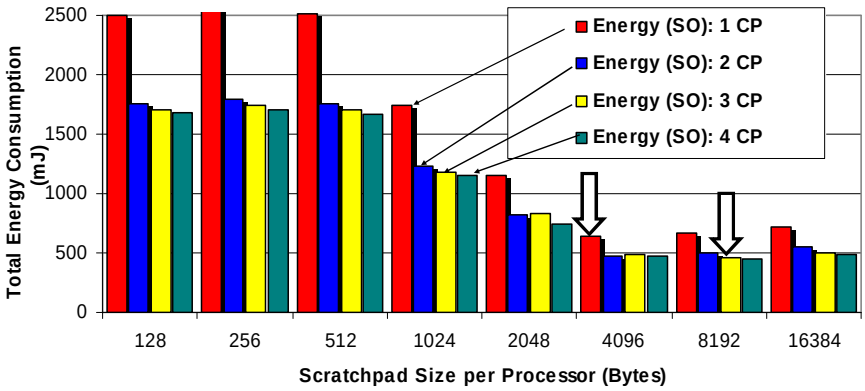
Application Example:
Multi-Processor Edge Detection



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

Universität Dortmund

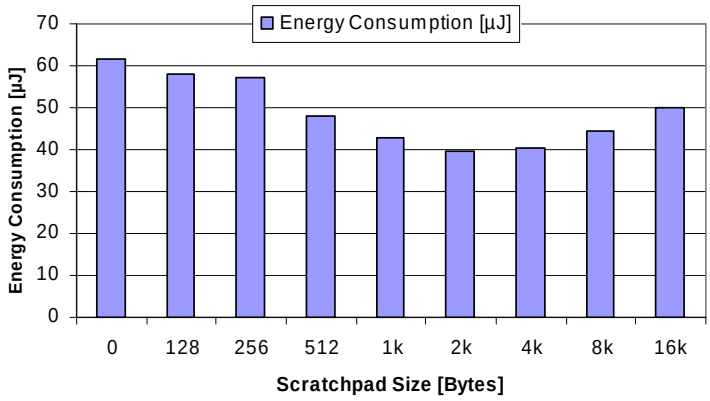
Results:
Scratchpad Overlay for Edge Detection



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

Universität Dortmund

Results
DES-Encryption

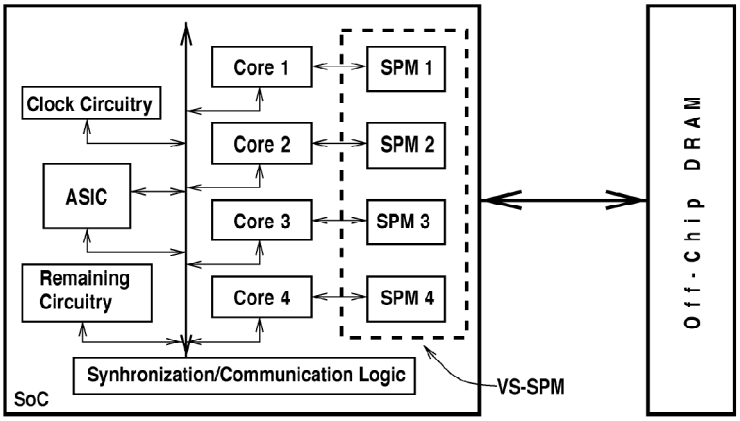


DES-Encryption: 4 processors: 2 Controllers+2 Compute Engines

Energy values from ST Microelectronics

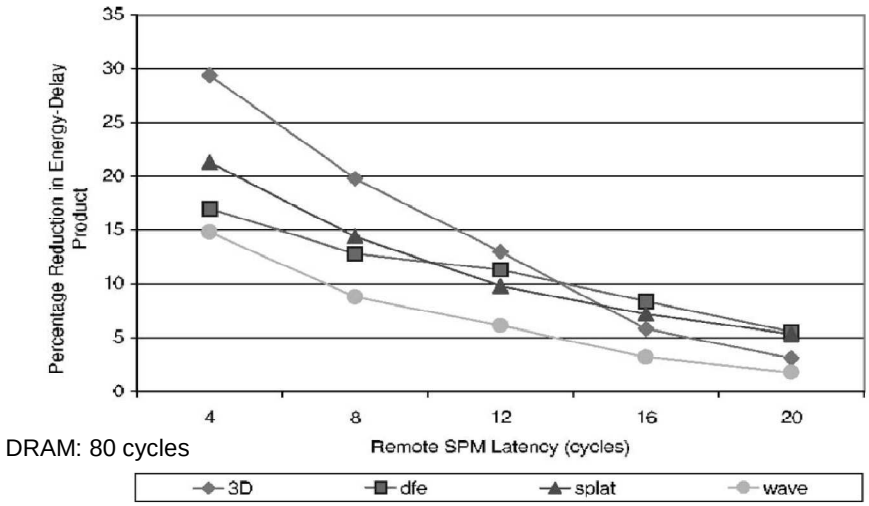
Result of ongoing cooperation between U. Bologna and U. Dortmund supported by ARTIST2 network of excellence.

MPSoC with shared SPMs



[M. Kandemir, I. Kadayif, A. Choudhary, J. Ramanujam, I. Kolcu: Compiler-Directed Scratch Pad Memory Optimization for Embedded Multiprocessors, *IEEE Trans. on VLSI*, Vol. 12, 2004, pp. 281-286]

Energy benefits despite large latencies for remote SPMs



DRAM: 80 cycles

Universität Dortmund

Extensions

- Using DRAM
- Applications to Flash memory (copy code or execute in place): PhD thesis of Lars Wehmeyer
  according to own experiments: very much parameter dependent
- Trying to imitate advantages of SPM with caches: partitioned caches, etc.

Universität Dortmund

Outline

- Introduction, Motivation
- Scratch-pad memories
  - Characteristics
  - Non-overlaying allocation, partitions, stack, heap, WCET
  - Overlaying (dynamic) allocation
  - Multiple processes & processors
- Caches
  - Predictability, loop transformations
  - Prefetching, layout transformations
- Other levels of the memory hierarchy
  - Array folding
  - ADL-driven compilation for hierarchies
  - Rewriting algorithms for memory hierarchies
- Conclusion, Additional references

Improving predictability for caches

- Loop caches
- Mapping code to less used part(s) of the index space
- Cache locking/freezing
- Mapping pieces of software to specific ways
  Methods:
  - 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

☞ “Caches behave almost like a scratch pad”

Locking of I-Caches

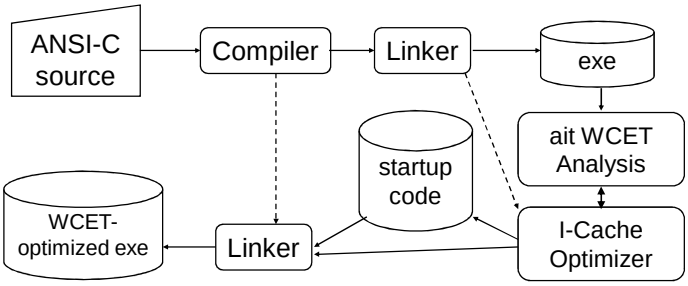
Many caches allow “locking” or “freezing” (no replacements).

Can be used to improve timing predictability:

- Load promising Functions into Cache

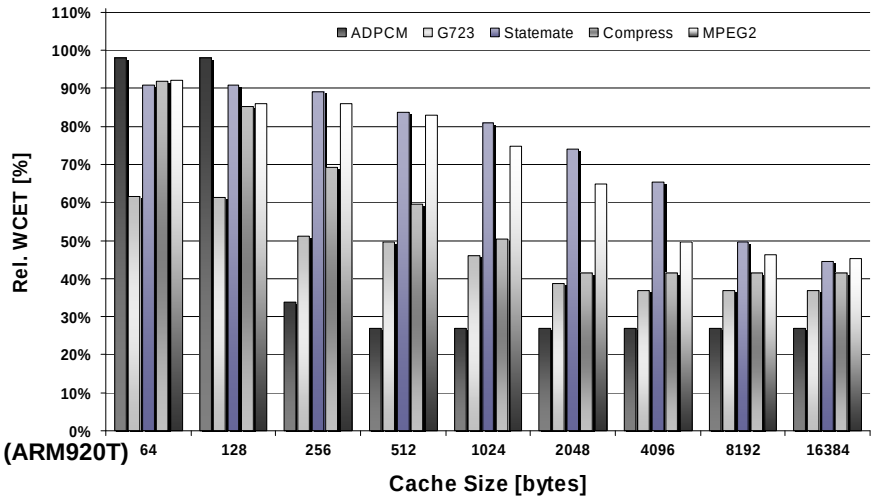
Optimizations:

- Worst case paths can change during optimization
- Requires coupling of timing analysis tool and compiler

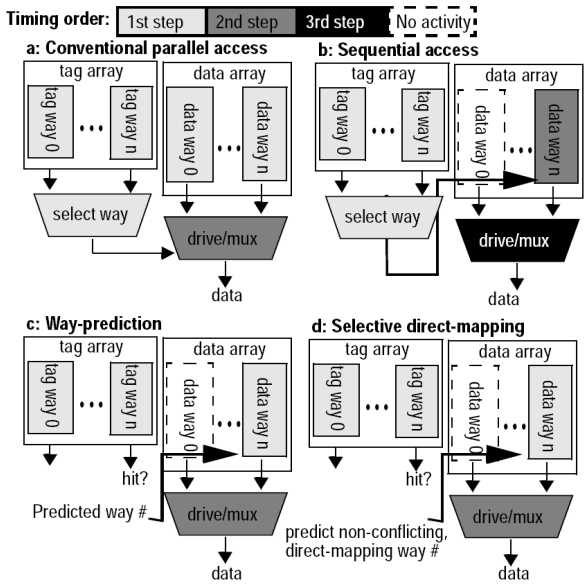


[Heiko Falk, Sascha Plazar, Henrik Theiling: Compile-Time Decided Instruction Cache Locking Using Worst-Case Execution Paths, *CODES/ISSS*, 2007]

Relative WCETs after I-Cache Locking



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]

Universität Dortmund

Outline

- Introduction, Motivation
- Scratch-pad memories
  - Characteristics
  - Non-overlaying allocation, partitions, stack, heap, WCET
  - Overlaying (dynamic) allocation
  - Multiple processes & processors
- Caches
  - Predictability, loop transformations
  - Prefetching, layout transformations
- Other levels of the memory hierarchy
  - Array folding
  - ADL-driven compilation for hierarchies
  - Rewriting algorithms for memory hierarchies
- Conclusion, Additional references

Universität Dortmund

Locality of Reference

Spatial Locality:

- The concept that likelihood of referencing an item is higher if an item close to it was just referenced.

Temporal Locality:

- The concept that a item that is referenced at one point in time will be referenced again in the near future.

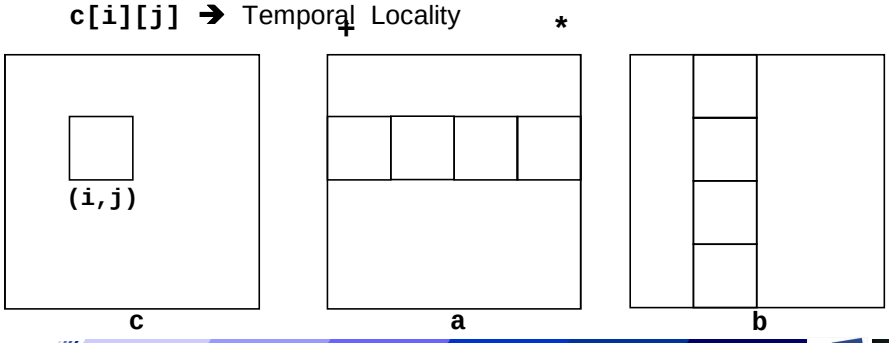
```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    for (k=0; k<N; k++) {
      c[i][j] += a[i][k] * b[k][j]; }}}
```

Locality: Matrix Multiplication

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    for (k=0; k<N; k++) {
      c[i][j] += a[i][k] * b[k][j]; }}}
```

I_1 = (i,j,k) and I_2 = (i,j,k+1)

c[i][j] ➔ Temporal Locality

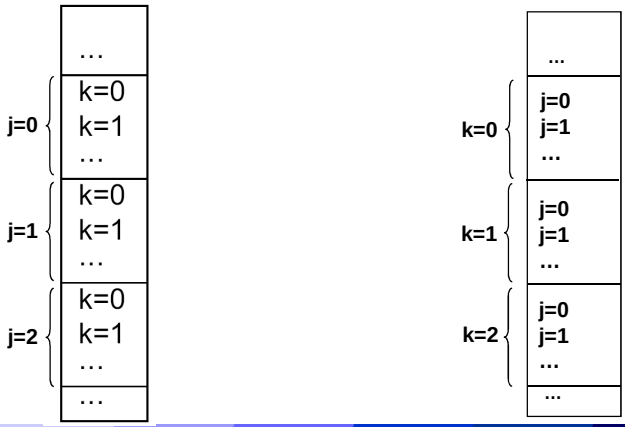


Row major order vs. column major order allocation

Array p[j][k]

Row major order (C)

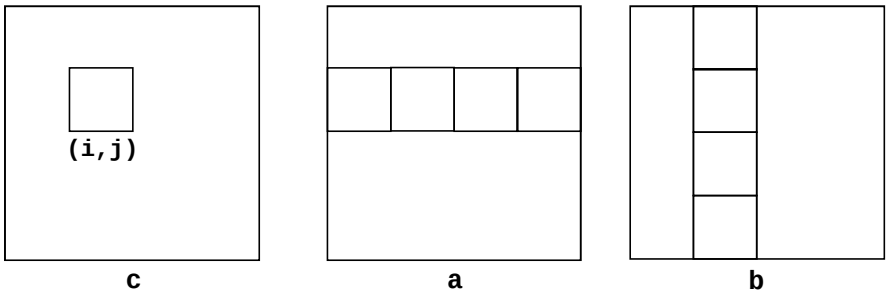
Column major order (FORTRAN)



Locality: Matrix Multiplication (2)

I_1 = (i,j,k) and I_2 = (i,j,k+1)

- c[i][j] ➔ Temporal Locality (layout independent)
- a[i][k] ➔ Spatial Locality (Row major order)
- b[k][j] ➔ Spatial Locality (Column major order)



Data Layout Transformations - Example

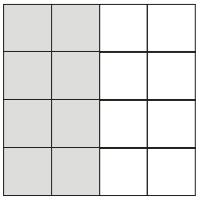
Change layout of arrays according to the access pattern

```
int c[4][4];
for (i=0; i<4; i++) {
  for (j=0; j<2; j++) {
    c[i][j]= …; }}}
```

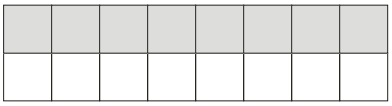
⇨

```
int c[2][8];
for (i=0; i<4; i++) {
  for (j=0; j<2; j++) {
    c[2*i+j]= …; }}}
```

Poor cache behavior



Good cache behavior



[V. Loechner et al.: Precise Data Locality Optimization of Nested Loops, *The Journal of Supercomputing*, Vol 21, 2002]

Increases the spatial locality of array accesses

Cache Aware Compiler Optimizations

Performance of a cache depends upon locality of references.

Data Cache:

- Loop Restructuring Transformations ➔ Temporal Locality
- Data Layout Transformations ➔ Spatial Locality

Instruction Cache:

- Code Layout Transformations ➔ Spatial Locality

Loop Restructuring Transformations - Loop Interchange -

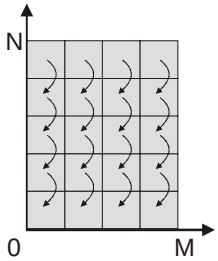
Assumption: row major order

```
for (i=0; i<M; i++) {
  for (j=0; j<N; j++) {
    c[j][i] += … }}}
```

⇨

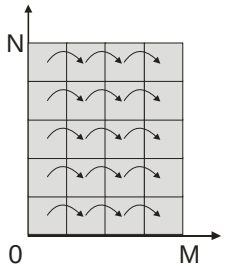
```
for (j=0; i<N; j++) {
  for (i=0; i<M; i++) {
    c[j][i] += … }}}
```

Poor cache behavior



Best performance if innermost loop corresponds to rightmost array index

Good cache behavior



[D. F. Bacon et al.: Compiler Transformations for High-Performance Computing, *ACM Computing Surveys*, 1994]

Universität Dortmund

☞ Loop interchange: Example

☞ Improved locality

```
…#define iter 400000
int a[20][20][20];
void computeijk() {int i,j,k;
      for (i = 0; i < 20; i++) {
            for (j = 0; j < 20; j++) {
                  for (k = 0; k < 20; k++) {
                        a[i][j][k] += a[i][j][k];}}}}
void computeikj() {int i,j,k;
      for (i = 0; i < 20; i++) {
            for (j = 0; j < 20; j++) {
                  for (k = 0; k < 20; k++) {
                        a[i][k][j] += a[i][k][j] ;}}}}…
start=time(&start);for(z=0;z<iter;z++)computeijk();
end=time(&end);
printf("ijk=%16.9f\n",1.0*difftime(end,start));
```

(SUIF interchanges array indexes instead of loops; low resolution time calls)

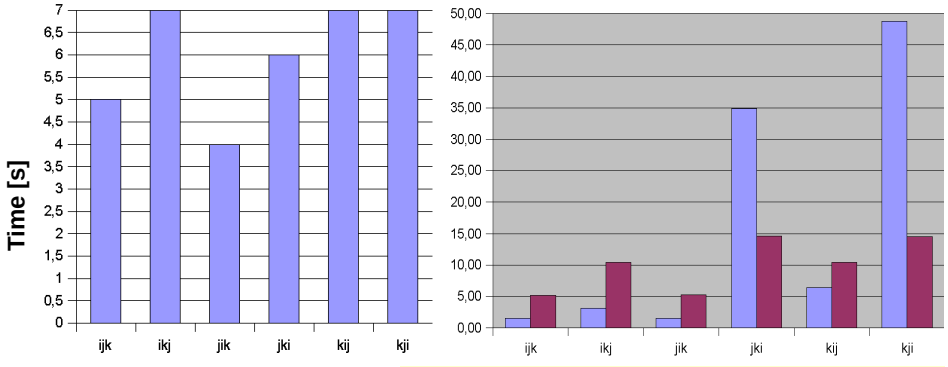
Universität Dortmund

Results:
strong influence of the memory architecture

Loop structure: i j k

Dramatic impact of locality

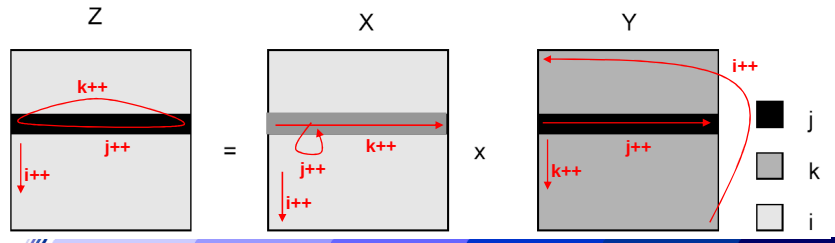
Processor	Ti C6xx	Sun SPARC	Intel Pentium
reduction to [%]	~ 57%	35%	3.2 %



[Till Buchwald, Diploma thesis, Univ. Dortmund, Informatik 12, 12/2004]

Loop tiling/loop blocking
Original version

```
for (i=1; i<=N; i++)
        for(k=1; k<=N; k++){
                r=X[i,k]; /* to be allocated to a register*/
                for (j=1; j<=N; j++)
                        Z[i,j] += r* Y[k,j]
} % Never reusing information in the cache for Y and Z if N
is large or cache is small (O(N³) references for Z and Y).
```



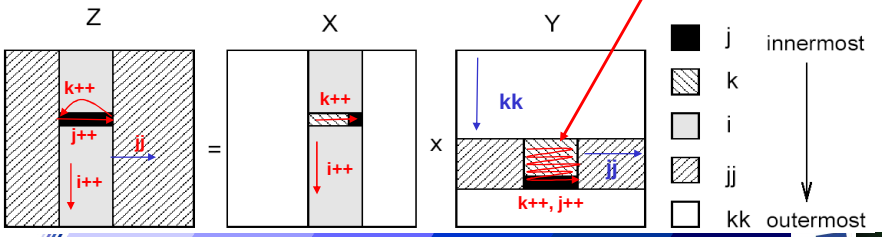
Loop tiling/loop blocking
tiled version

```
for (kk=1; kk<= N; kk+=B)
 for (jj=1; jj<= N; jj+=B)
  for (i=1; i<= N; i++)
   for (k=kk; k<= min(kk+B-1,N); k++){
    r=X[i][k]; /* to be allocated to a register*/
    for (j=jj; j<= min(jj+B-1, N); j++)
     Z[i][j] += r* Y[k][j]
```

Reuse factor of B for Z and N for Y,

O(N³/B) accesses to main memory for Z

Same elements for next iteration of i

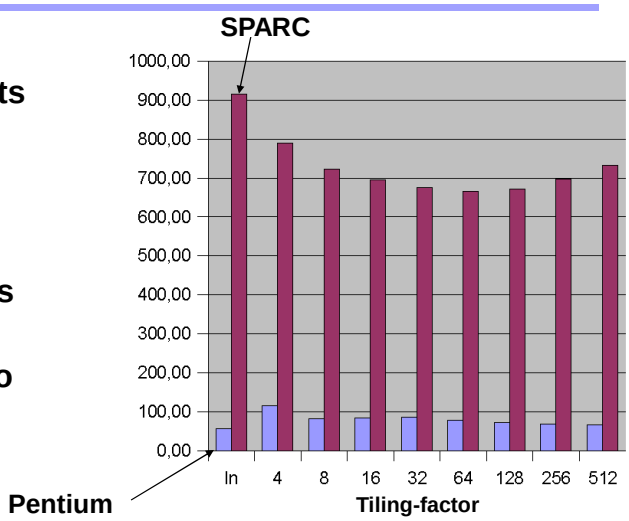


Example

In practice, results by Buchwald are disappointing.

One of the few cases where an improvement was achieved:

Source: similar to matrix mult.



[Till Buchwald, Diploma thesis, Univ. Dortmund, Informatik 12, 12/2004]

Loop fusion (merging), loop fission

```
for(j=0; j<=n; j++)
  p[j]= ... ;
for (j=0; j<=n; j++) ,
  p[j]= p[j] + ...
```

⇔

```
for (j=0; j<=n; j++)
  {p[j]= ... ;
  p[j]= p[j] + ...}
```

Loops small enough to allow zero overhead Loops

Better locality for access to p.
Better chances for parallel execution.

Which of the two versions is best?

Architecture-aware compiler should select best version.

Example: loops with reuse

```
#define size 30
#define iter 40000
int a[size][size];
float b[size][size];
```

```
void ss2() {int i,j;
 for(i=0;i<size;i++) {
  for(j=0;j<size;j++) {
   a[i][j] += 17;}}
 for(i=0;i<size;i++){
  for(j=0;j<size;j++) {
   b[i][j]-=a[i][j];}}}
```

```
void ms2() {int i,j;
 for(i=0;i<size;i++) {
  for(j=0;j<size;j++) {
   a[i][j]+= 17;}
  for(j=0;j<size;j++){
   b[i][j]-=a[i][j];}}}
```

```
void mm2() {int i,j;
 for(i=0;i<size;i++) {
  for(j=0;j<size;j++) {
   a[i][j]+=17;
   b[i][j]-=a[i][j];}}}
```


Loop unrolling

for (j=0; j<=n; j++)
p[j]= ... ;

for (j=0; j<=n; j+=2)
{p[j]= ... ; p[j+1]= ...}

factor = 2

Better locality for access to p.

Less branches per execution of the loop. More opportunities for optimizations.

Tradeoff between code size and improvement.

Extreme case: completely unrolled loop (no branch).

Universität Dortmund

Example: matrixmult

```
#define s 30
#define iter 4000
int
a[s][s],b[s][s],c[s]
[s];
void compute(){int
i,j,k;
 for(i=0;i<s;i++){
  for(j=0;j<s;j++){
   for(k=0;k<s;k++){
    c[i][k]+=
    a[i][j]*b[j][k];
}}}}
```

```
extern void compute2()
 {int i, j, k;
  for (i = 0; i < 30; i++) {
   for (j = 0; j < 30; j++)  {
    for (k = 0; k <= 28; k += 2)
     {{int *suif_tmp;
     suif_tmp = &c[i][k];
     *suif_tmp=
     *suif_tmp+a[i][j]*b[j][k];}
     {int *suif_tmp;
     suif_tmp=&c[i][k+1];
     *suif_tmp=*suif_tmp
           +a[i][j]*b[j][k+1];
 }}}}
 return;}
```


Universität Dortmund

Summary Lecture 3

- Overlaying SPM allocation
  - Multiple processes
    - Dynamic set of processes
    - MMU-based approach (SNU)
  - Multiple processors
    - Verma's approach (☞MPARM)
    - Kandemir's MPSoC approach
- Caches
  - Locked caches
  - Locality of reference
  - Loop transformations, tiling

UNIVERSITÄT DORTMUND

Compilation for Caches

Compilation for other memory hierarchy levels

Universität Dortmund

Outline

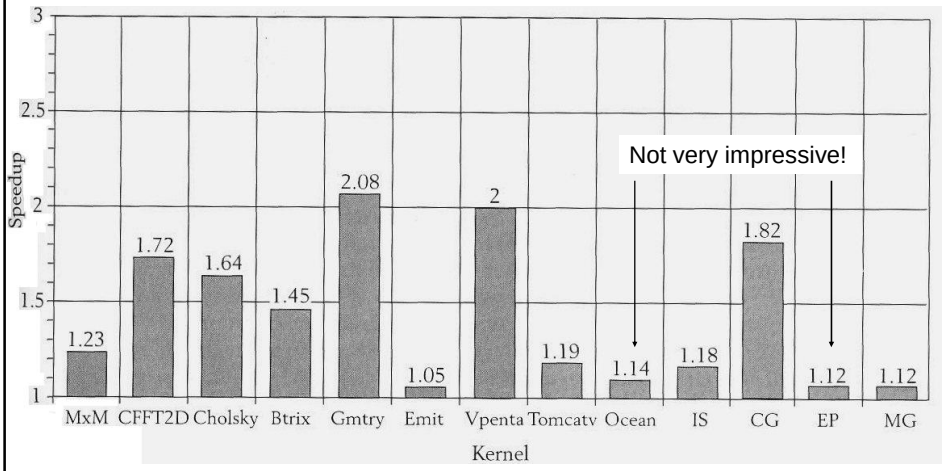
- Introduction, Motivation
- Scratch-pad memories
  - Characteristics
  - Non-overlaying allocation, partitions, stack, heap, WCET
  - Overlaying (dynamic) allocation
  - Multiple processes & processors
- Caches
  - Predictability, loop transformations
  - Prefetching, layout transformations
- Other levels of the memory hierarchy
  - Array folding
  - ADL-driven compilation for hierarchies
  - Rewriting algorithms for memory hierarchies
- Conclusion, Additional references

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]

© Morgan-Kaufman, 2002

Universität Dortmund

Optimization for exploiting processor-memory interface: Problem Definition (1)

XScale is stalled for 30% of time, but each stall duration is small

- Average stall duration = 4 cycles
- Longest stall duration < 100 cycles

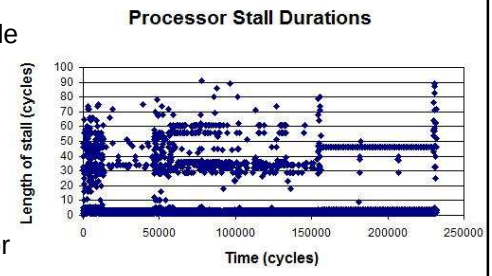
Break-even stall duration for profitable switching

- 360 cycles

Maximum processor stall

- < 100 cycles

NOT possible to switch the processor to IDLE mode



[A. Shrivastava, E. Earlie, N. Dutt, A. Nicolau: Aggregating processor free time for energy reduction, *Intern. Conf. on Hardware/Software Codesign and System Synthesis* (CODES/ISSS), 2005, pp. 154-159]

Based on slide by A. Shrivastava

Universität Dortmund

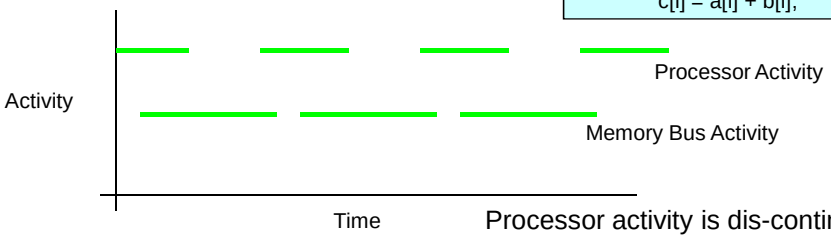
Optimization for exploiting processor-memory interface: Problem Definition (2)

- CT (Computation Time): Time to execute an iteration of the loop, assuming all data is present in the cache
- DT (Data Transfer Time): Time to transfer data required by an iteration of a loop between cache and memory

Consider the execution of a memory-bound loop (DT > CT)

- Processor has to stall

```
for (int i=0; i<1000; i++)
    c[i] = a[i] + b[i];
```



Processor activity is dis-continuous

Memory activity is dis-continuous

Based on slide by A. Shrivastava

Universität Dortmund

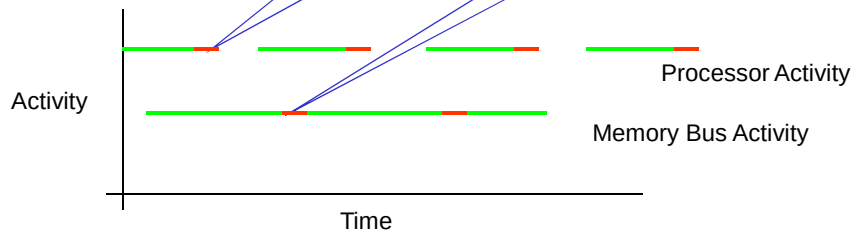
Optimization for exploiting processor-memory interface: Prefetching Solution

```
for (int i=0; i<1000; i++)
        prefetch a[i+4];
        prefetch b[i+4];
        prefetch c[i+4];
        c[i] = a[i] + b[i];
```

Each processor activity period increases

Memory activity is continuous

Total execution time reduces



Processor activity is dis-continuous

Memory activity is continuous

Based on slide by A. Shrivastava

Universität Dortmund

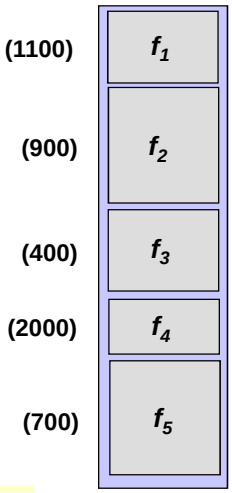
Outline

- Introduction, Motivation
- Scratch-pad memories
  - Characteristics
  - Non-overlaying allocation, partitions, stack, heap, WCET
  - Overlaying (dynamic) allocation
  - Multiple processes & processors
- Caches
  - Predictability, loop transformations
  - Prefetching, layout transformations ⇐
- Other levels of the memory hierarchy
  - Array folding
  - ADL-driven compilation for hierarchies
  - Rewriting algorithms for memory hierarchies
- Conclusion, Additional references

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]

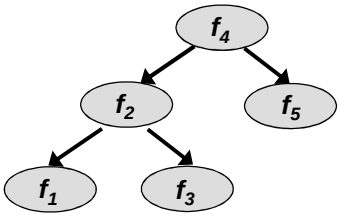
Code Layout Transformations (2)

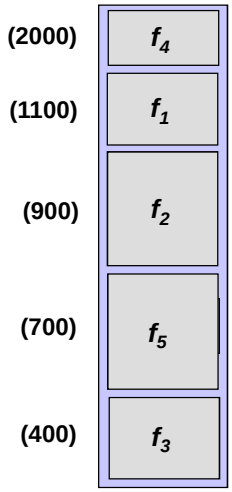
Execution counts based approach:

- 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





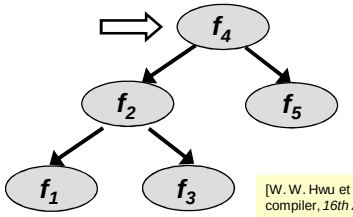
Code Layout Transformations (3)

Call-Graph Based Algorithm:

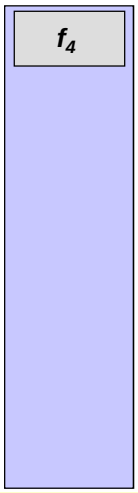
- Create weighted call-graph.
- Place functions according to weighted depth-first traversal.

  $f_4 > f_2 > f_1 > f_3 > f_5$

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]

Universität Dortmund

Code Layout Transformations (3)

Call-Graph Based Algorithm:

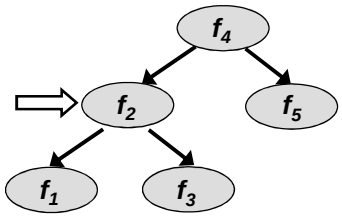
- Create weighted call-graph.
- Place functions according to weighted depth-first traversal.

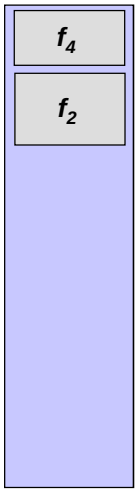
  $f_4 > f_2 > f_1 > f_3 > f_5$

Increases spatial locality.

(2000)

(900)





Universität Dortmund

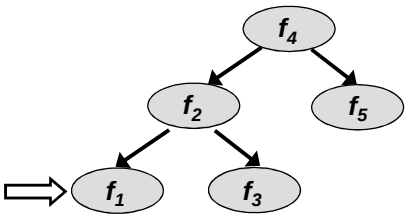
Code Layout Transformations (4)

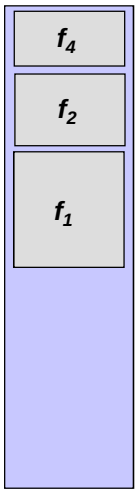
Call-Graph Based Algorithm:

- 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.





Universität Dortmund

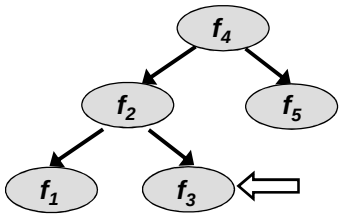
Code Layout Transformations (5)

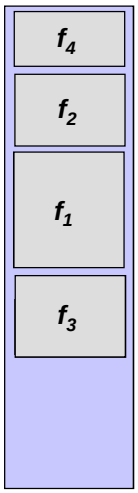
Call-Graph Based Algorithm:

- 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.





Universität Dortmund

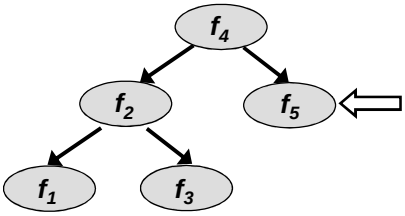
Code Layout Transformations (6)

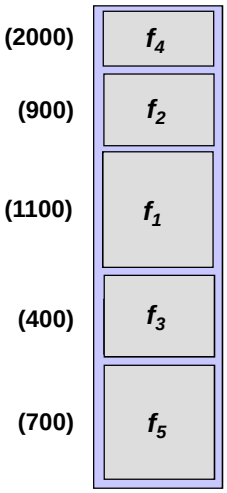
Call-Graph Based Algorithm:

- 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.



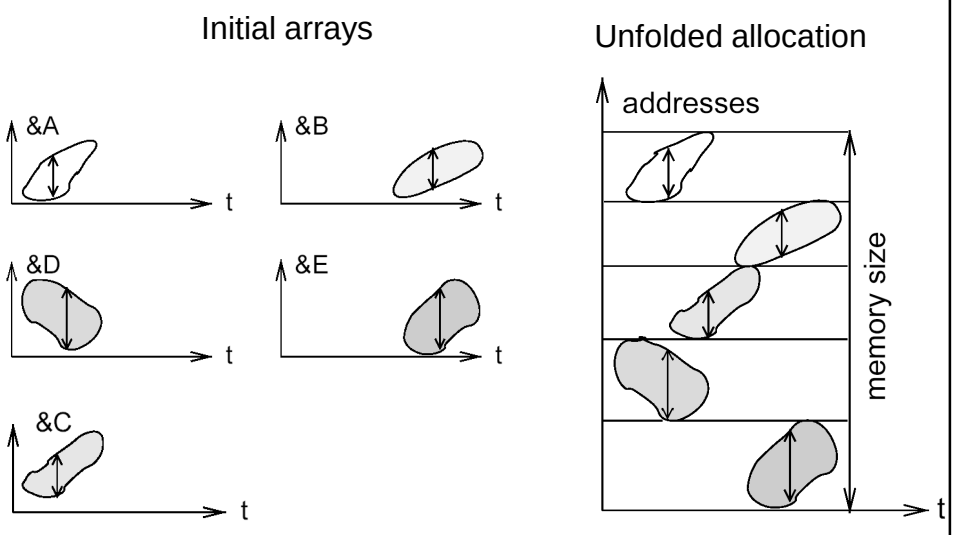


Universität Dortmund

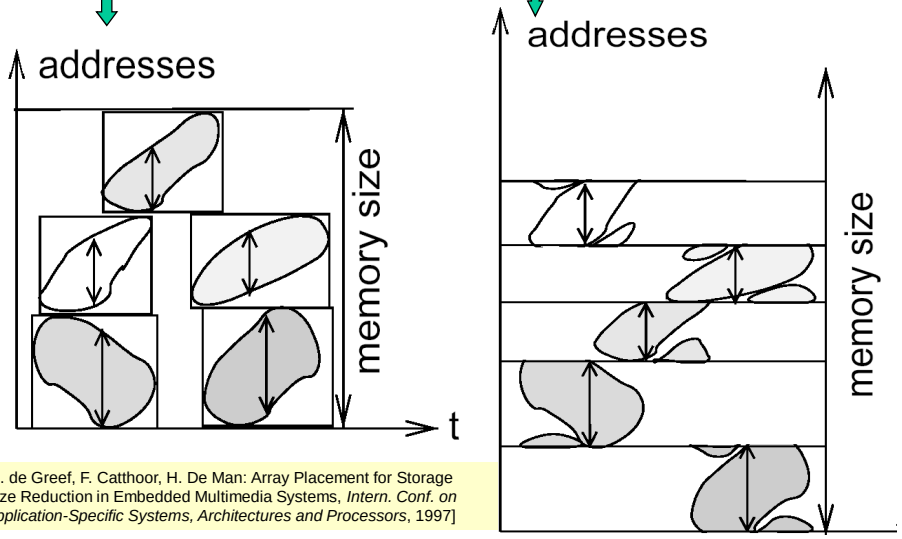
Outline

- Introduction, Motivation
- Scratch-pad memories
  - Characteristics
  - Non-overlaying allocation, partitions, stack, heap, WCET
  - Overlaying (dynamic) allocation
  - Multiple processes & processors
- Caches
  - Predictability, loop transformations
  - Prefetching, layout transformations
- Other levels of the memory hierarchy
  - Array folding
  - ADL-driven compilation for hierarchies
  - Rewriting algorithms for memory hierarchies
- Conclusion, Additional references

Memory allocation for arrays



Inter-array folding and Intra-array folding



[E. de Greef, F. Catthoor, H. De Man: Array Placement for Storage Size Reduction in Embedded Multimedia Systems, *Intern. Conf. on Application-Specific Systems, Architectures and Processors*, 1997]

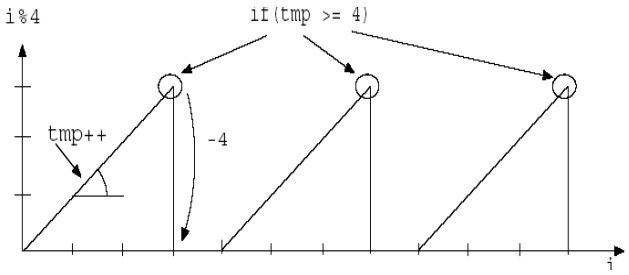
Requires optimization of address computations

- Array folding implemented in IMEC's DTSE optimization.
- Leads to costly div and mod ops.
- ADOPT address optimizations remove these operations
  E.g.: mod → ++ and reset on pointers (indexes)

```
for(i=0; i<20; i++)
      B[i%4];
```

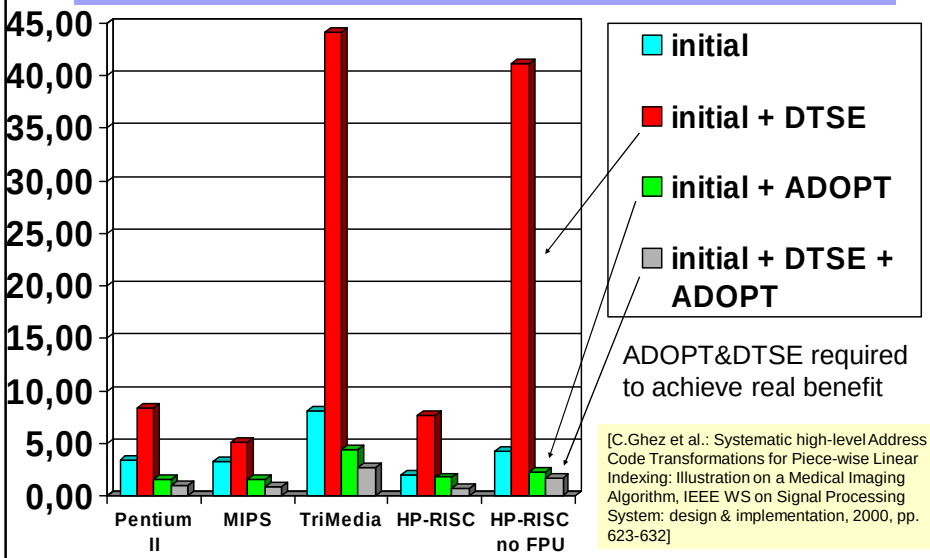
⬇

```
tmp=0;
for(i=0; i<20; i++)
      {if(tmp>=4)
        tmp-=4;
      B[tmp];
      tmp++;}
```



Architecture-aware compiler should find best transformation

Results (Mcycles for cavity benchmark)



ADOPT&DTSE required to achieve real benefit

[C.Ghez et al.: Systematic high-level Address Code Transformations for Piece-wise Linear Indexing: Illustration on a Medical Imaging Algorithm, IEEE WS on Signal Processing System: design & implementation, 2000, pp. 623-632]

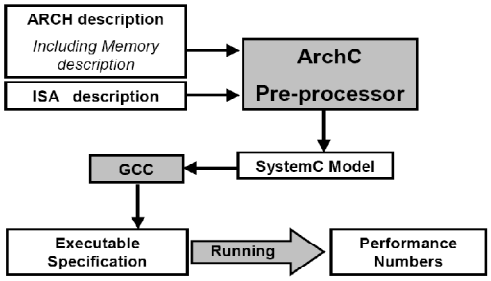
Outline

- Introduction, Motivation
- Scratch-pad memories
  - Characteristics
  - Non-overlaying allocation, partitions, stack, heap, WCET
  - Overlaying (dynamic) allocation
  - Multiple processes & processors
- Caches
  - Predictability, loop transformations
  - Prefetching, layout transformations
- Other levels of the memory hierarchy
  - Array folding
  - ADL-driven compilation for hierarchies
  - Rewriting algorithms for memory hierarchies
- Conclusion, Additional references

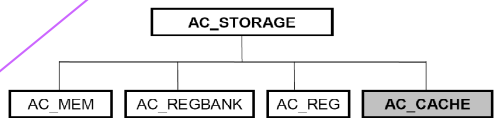
Memory hierarchy description languages: ArchC

Consists of description of ISA and HW architecture

Extension of SystemC (can be generated from ArchC):

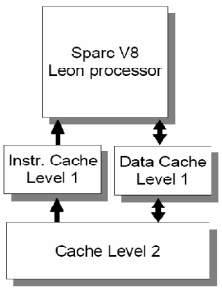


Storage class structure



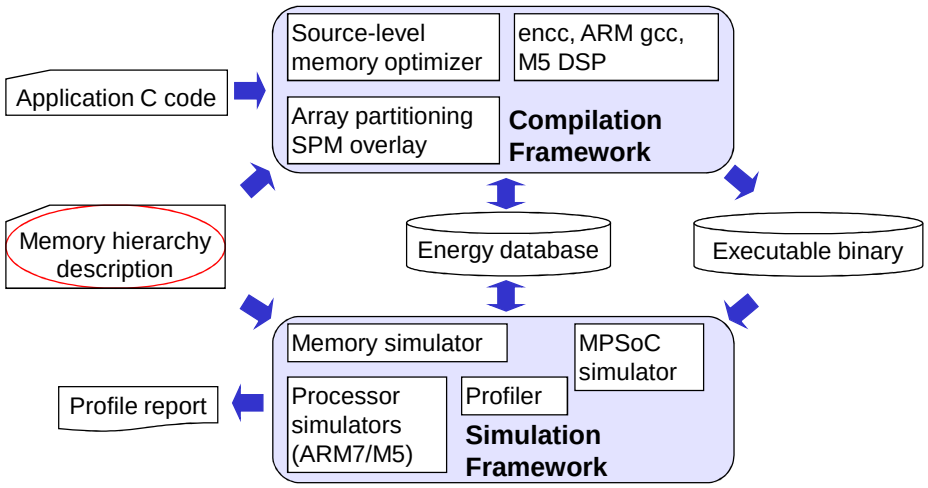
[P. Viana, E. Barros, S. Rigo, R. Azevedo, G. Araújo: Exploring Memory Hierarchy with ArchC, *15th Symposium on Computer Architecture and High Performance Computing*, 2003, pp. 2 – 9]

Example: Description of a simple cache-based architecture




```
AC_ARCH(leon){

  ac_cache   icache("dm", 128, "wt")
  ac_cache   dcache("2w", 64, 4, "wt", "lru")
  ac_cache   ul2cache("dm", 4k, "wt")

  ac_regbank RB:520;
  ac_reg PRS, Y, WIM;

  ac_pipe   pipe = {IF, ID, EX, MEM, WB};

  ARCH_CTOR(leon){
    ac_isa("leon_isa.ac");

    icache.bindTo( ul2cache );  //Memory hierarchy
    dcache.bindTo( ul2cache );  //construction
  };
};
```


Memory Aware Compilation and Simulation Framework (for C) MACC

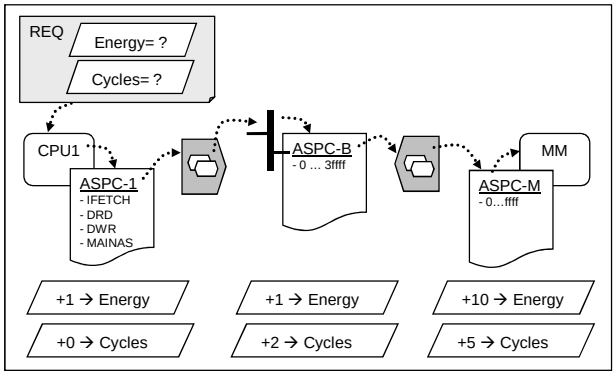


[M. Verma, L. Wehmeyer, R. Pyka, P. Marwedel, L. Benini: Compilation and Simulation Tool Chain for Memory Aware Energy Optimizations, *Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS VI)*, 2006].

Universität Dortmund

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

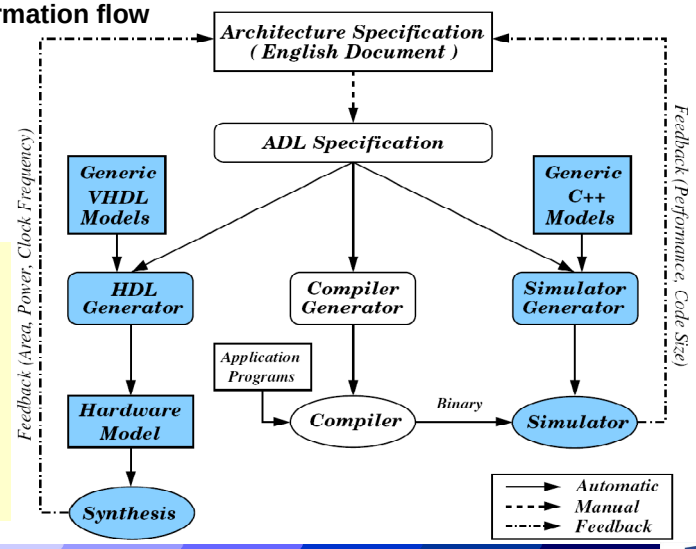


[R. Pyka et al.: Versatile System level Memory Description Approach for embedded MPSoCs, *University of Dortmund, Informatik 12*, 2007]

Universität Dortmund

Controlling tool chain generation through an architecture description language (ADL): EXPRESSION

Overall information flow



[P. Mishra, A. Shrivastava, N. Dutt: Architecture description language (ADL)-driven software toolkit generation for architectural exploration of programmable SOCs, *ACM Trans. Des. Autom. Electron. Syst. (TODAES)*, 2006, pp. 626-658]

Universität Dortmund

Description of Memories in EXPRESSION

Generic approach, based on the analysis of a wide range of systems;

Used for verification.

```
(STORAGE_SECTION
  (DataL1
      (TYPE DCACHE) (WORDSIZE 64)
      (LINESIZE 8) (NUM_LINES 1024)
      (ASSOCIATIVITY 2) (READ_LATENCY 1)  ...
      (REPLACEMENT_POLICY LRU)
      (WRITE_POLICY WRITE_BACK)
  )
  (ScratchPad
      (TYPE SRAM) (ADDRESS_RANGE 0 4095)  ....
  )
  (SB
      (TYPE STREAM_BUFFER) .....
  (InstL1
      (TYPE ICACHE) ……..
  )
  (L2
      (TYPE DCACHE) …….
  )
  (MainMemory
      (TYPE DRAM)
  )
  (Connect
      (TYPE CONNECTIVITY)
      (CONNECTIONS
          (InstL1, L2) (DataL1, SB) (SB, L2)
          (L2, MainMemory)
  )))
```


Universität Dortmund

EXPRESSION: results

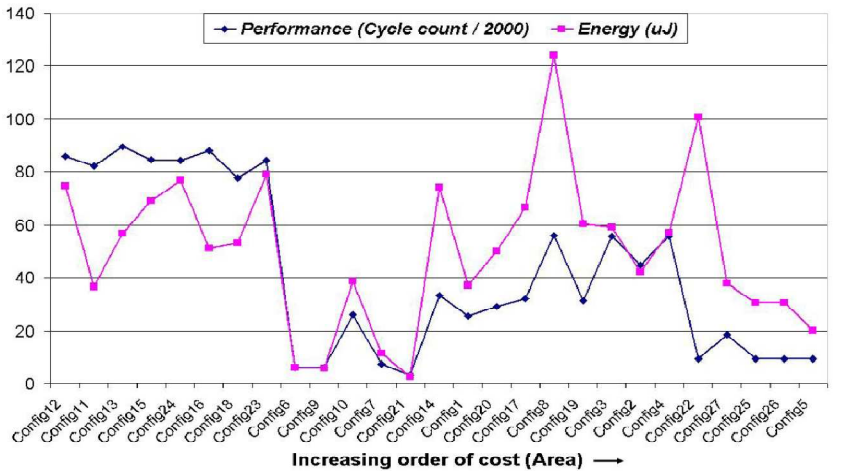


Figure 4.11: Memory exploration results for GSR

Universität Dortmund

Optimization for main memory Exploiting burst mode of DRAM (1)

```
for(i=0;i<9;i++){
  a = a + x[i] + y[i];
  b = b + z[i] + u[i];
}
```

(a) Sample code

= row decode (2 cycles)
= column decode (1 cycle)
= precharge (2 cycles)

(b) Synchronous DRAM access primitives

Dynamic cycle count = 9 x (5 x 4) = 180 cycles

(c) Unoptimized schedule

```
for(i=0;i<9;i+=3){
  a = a + x[i] + x[i+1] + x[i+2] +
          y[i] + y[i+1] + y[i+2];
  b = b + z[i] + z[i+1] + z[i+2]+
          u[i] + u[i+1] + u[i+2];
}
```

(d) Loop unrolled to allow burst mode

Supported trafos: memory mapping, code reordering or loop unrolling

[P. Grun, N. Dutt, A. Nicolau: Memory aware compilation through accurate timing extraction, *DAC*, 2000, pp. 316 – 321]

Universität Dortmund

Optimization for main memory Exploiting burst mode of DRAM (2)

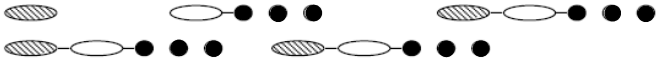
Timing extracted from EXPRESSION model

```
for(i=0; i<9;i+=3){
a=a+x[i]+x[i+1]+x[i+2]+
      y[i]+y[i+1]+y[i+2];
b=b+z[i]+z[i+1]+z[i+2]+
      u[i]+u[i+1]+u[i+2];}
```

(d) Loop unrolled to allow burst mode

Dynamic behavior (dynamic cycle count = 3 x 28 = 84 cycles)

(e) Optimized code without accurate timing



Dynamic cycle count = 3 x 20 = 60 cycles

(f) Optimized code with accurate timing

Open circles of original paper changed into closed circles (column decodes).

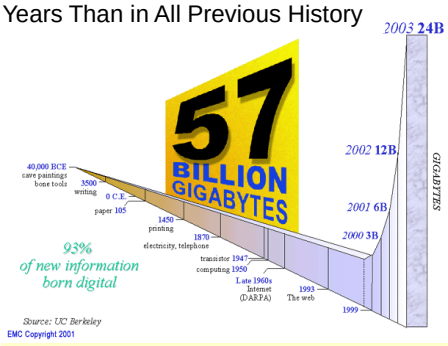
Outline

- Introduction, Motivation
- Scratch-pad memories
  - Characteristics
  - Non-overlaying allocation, partitions, stack, heap, WCET
  - Overlaying (dynamic) allocation
  - Multiple processes & processors
- Caches
  - Predictability, loop transformations
  - Prefetching, layout transformations
- Other levels of the memory hierarchy
  - Array folding
  - ADL-driven compilation for hierarchies
  - Rewriting algorithms for memory hierarchies
- Conclusion, Additional references

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



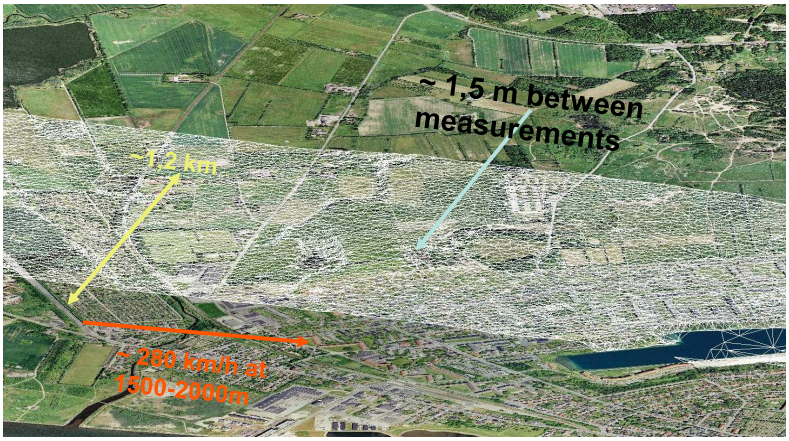
[© Larse Arge, I/O-Algorithms, http://www.daimi.au.dk/~large/ioS07/]

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

Example: LIDAR Terrain Data

- COWI A/S (and others) is currently scanning Denmark

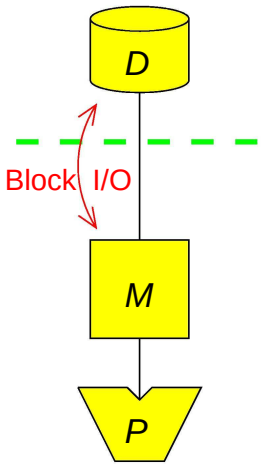


[© Larse Arge, I/O-Algorithms, http://www.daimi.au.dk/~large/ioS07/]

Application Example: Flooding Prediction

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

External Memory Model



N = # of items in the problem instance
B = # of items per disk block
M = # of items that fit in main memory

T = # of items in output

I/O: Move block between memory and disk

We assume (for convenience) that $M > B^2$

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

Scalability Problems: Block Access Matters

- Example: Reading an array from disk
  - Array size N = 10 elements
  - Disk block size B = 2 elements
  - Main memory size M = 4 elements (2 blocks)

1	5	2	6	3	8	9	4	7	10

Algorithm 1: N=10 I/Os

1	2	10	9	5	6	3	4	8	7

Algorithm 2: N/B=5 I/Os

- Difference between N and N/B large since block size is large
  - Example: $N = 256 \times 10^6$, $B = 8000$, *1ms* disk access time
    ⇒ ***N*** I/Os take 256×10^3 *sec* = *4266 min* = *71 hr*
    ⇒ ***N/B*** I/Os take *256/8 sec* = *32 sec*

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

Re-writing algorithms for memory hierarchies

Analysis of algorithm complexity mostly using the *RAM* (random access machine; const. mem. acc. times) model outdated

☞ take memory hierarchies explicitly into account.

Example:

- Usually, divide-&-conquer algorithms are good.
- "Cache"-oblivious algorithms (are good for any size of the faster memory and any block size). Assuming
  - Optimal replacement (Belady's algorithm)
  - 2 Memory levels considered (there can be more)
  - Full associativity
  - Automatic replacement

Unlikely to be ever automatic

[Piyush Kumar: Cache Oblivious Algorithms, in: U. Meyer et al. (eds.): Algorithms for Memory Hierarchies, *Lecture Notes in Computer Science, Volume 2625*, 2003, pp. 193-212]

[Naila Rahman: Algorithms for Hardware Caches and TLB, in: U. Meyer et al. (eds.): Algorithms for Memory Hierarchies, *Lecture Notes in Computer Science, Volume 2625*, 2003, pp. 171-192]

Fundamental Bounds

	Internal	External
Scanning:	N	$\frac{N}{B}$
Sorting:	$N \log N$	$\frac{N}{B}\log_{M/B}\frac{N}{B}$
Permuting	N	$\min\{N, \frac{N}{B}\log_{M/B}\frac{N}{B}\}$
Searching:	$\log_2 N$	$\log_B N$

- Note:
  - Linear I/O: $O(N/B)$
  - Permuting not linear
  - Permuting and sorting bounds are equal in all practical cases
  - B factor VERY important: $\frac{N}{B} < \frac{N}{B}\log_{M/B}\frac{N}{B} << N$

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

Additional References (1)

- D. Page: Effective Use of Partitioned Cache Memories, *PhD Thesis, Department of Computer Science, University of Bristol*, UK 2001.
- O. S. Unsal et al. Cool-Cache: A Compiler-Enabled Energy Efficient Data Caching Framework for Embedded/ Multimedia Processors, *ACM Transactions on Embedded Computing Systems*, Vol. 2, 2003
- J. Kin et al.: The Filter Cache: An Energy Efficient Memory Structure, *30th Annual International Symposium on Microarchitecture (MICRO)*, 1997.
- L. H. Lee et al. Instruction Fetch Energy Reduction Using Loop Caches For Embedded Applications with small Tight Loops, *International Symposium on Low Power Electronics and Design (ISLPED)*, 1999.
- S. C. A. Gordon-Ross, F. Vahid: Exploiting Fixed Programs in Embedded Systems: A Loop Cache Example, *Computer Architecture Letters*, 1, 2002.
- S. Steinke: Untersuchung des Energieeinsparungspotenzials in eingebetteten Systemen durch energieoptimierende Compilertechnik (in German), *PhD Thesis, University of Dortmund, Germany*, 2003.

Additional References (2)

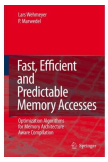
- P. R. Panda, N. Dutt, A. Nicolau: Memory Issues in Embedded Systems-On-Chip, *Kluwer Academic Publishers, Norwell, MA,* 1999.
- S. Steinke et al.: Assigning Program and Data Objects to Scratchpad for Energy Reduction, *Design Automation and Test in Europe Conference (DATE'02)*, 2002.
- L. Wehmeyer, P. Marwedel: Influence of On-chip Scratchpad Memories on WCET, *4th International 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 Conference (DATE'05)*, 2005
- L. Wehmeyer, U. Helmig, P. Marwedel: Compiler-optimized Usage of Partitioned Memories, *3rd Workshop on Memory Performance Issues (WMPI)*, 2004.
- F. Angiolini, L. Benini, A. Caprara: Polynomial-Time Algorithm for On-Chip Scratchpad Memory Partitioning, *International Conference on Compilers, Architecture and Synthesis of Embedded Systems (CASES'03)*, 2003

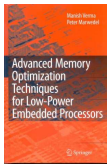
Additional References (3)

- J. Chame Shin, M. W. Hall: A Compiler Algorithm for Exploiting Page-Mode Memory Accesses in Embedded-DRAM Devices, *Workshop on Media and Stream Processors*, 2002
- M. Verma, P. Marwedel: Overlay of Scratchpad Memory for Low Power Embedded Processors, *IEEE Transactions on Very Large Scale Integration Systems (TVLSI),* Vol.14, 2006.
- E. Brockmeyer et al.: Layer Assignment Techniques for Low Energy in Multi-Layered Memory Organisations, *Design, Automation and Test in Europe Conference* (DATE), 2003.
- M. Verma, K. Petzold, L. Wehmeyer, H. Falk, P. Marwedel: Scratchpad Sharing Strategies for Multiprocess Embedded Systems: A First Approach, *IEEE 3rd Workshop on Embedded Real-Time Multimedia (ESTIMedia)*, 2005.
- ❖ L. Wehmeyer, P. Marwedel: Fast, Efficient and Predictable Memory Accesses Optimization Algorithms for Memory Architecture Aware Compilation, *Springer*, 2006
- ❖ M. Verma, Peter Marwedel: Advanced Memory Optimization Techniques for Low-Power Embedded Processors, *Springer*, 2007

New Books

- Lars Wehmeyer, Peter Marwedel: Fast, Efficient and Predictable Memory Accesses, *Springer*, 2006
- Manish Verma, Peter Marwedel: Advanced Memory Optimization Techniques for Low-Power Embedded Processors, *Springer*, May 2007
- Peter Marwedel: Embedded System Design, *Springer*, 2005 (German edition: 2007)





Summary lecture 4

- Caches
  - Prefetching
  - Code layout transformations
- Other memory hierarchy levels
  - Array folding
  - ADL-driven compilation for memory hierarchies
  - Rewriting algorithms for memory hierarchies

Overall summary

- Introduction, Motivation
- Scratch-pad memories
  - Characteristics
  - Non-overlaying allocation, partitions, stack, heap, WCET
  - Overlaying (dynamic) allocation
  - Multiple processes & processors
- Caches
  - Predictability, loop transformations
  - Prefetching, layout transformations
- Other levels of the memory hierarchy
  - Array folding
  - ADL-driven compilation for hierarchies
  - Rewriting algorithms for memory hierarchies
- Conclusion, Additional references

Credit goes to all students involved in the design of SPM tools at Dortmund, in particular to Stefan Steinke, Robert Pyka, Manish Verma and Lars Wehmeyer.

Thank you for your attention!
Questions?