

Compilers for Embedded Systems

Summer Term 2022

Heiko Falk

Institute of Embedded Systems
Electrical Engineering, Computer Science and Mathematics
Hamburg University of Technology

Chapter 7

LIR Optimizations and Transformations

Outline

1. Introduction & Motivation
2. Compilers for Embedded Systems – Requirements & Dependencies
3. Internal Structure of Compilers
4. Pre-Pass Optimizations
5. HIR Optimizations and Transformations
6. Code Generation
- 7. LIR Optimizations and Transformations**
8. Register Allocation
9. WCET-Aware Compilation
10. Outlook

Chapter Contents

7. LIR Optimizations and Transformations

- Generation of Bit-Packet Operations for NPUs
 - Motivation of Bit-True Data and Value Flow Analyses
 - Partial Order \mathcal{L}_4
 - Bit-True Analysis: Forward and Backward Simulation
 - Bit-True Optimizations: Dead Code Elimination; Generation of insert/extract operations
- Scratchpad Memory Optimizations
 - Properties of Main Memories, Caches and Scratchpads
 - Fix SPM Allocation (Functions, global Variables)
 - Fix SPM Allocation (Functions, Basic Blocks, global Variables)
 - SPM Allocations for Multi-Process Applications (partitioned, exclusive, hybrid)

Follow-Up: Data Flow Graphs

Data Flow Graph

- Node represents an operation
- Edges between nodes represent definitions (*DEFs*) and uses (*USEs*) of data

Accuracy of a DFG

- At the LIR level, a DFG node denotes a single machine operation
- Since the operands of machine operations are usually processor registers, the edges thus represent the data flow through *entire* registers.

DFGs & Bit-Packets

Bit-Packets

- Sequence of consecutive bits
- of arbitrary length
- starting at arbitrary positions
- and eventually crossing word-boundaries

DFGs and Bit-Packets

- DFGs model data flow based on atomic registers
- ☞ Information about irregularly-structured portions of registers is not provided
- ☞ ***Classical DFG-based techniques are usually inappropriate to generate bit-packet operations!***

Example

TPM and Bit-Packets

- Composed rule

```
dreg: tpm_BinaryExpAND ( tpm_BinaryExpSHR (
    dreg, const ) , const )
```

can cover the expression `(c >> 4) & 0x7` and generate the efficient operation `EXTR.U d_0, d_c, 4, 3`

- But: TPM reaches its limits for more complex code patterns to be covered
 - 👉 What if numbers `4 / 0x7` are not given as constants, but as values of variables?
 - 👉 What if other combinations of operators beyond `& / >>` are used to extract and insert bit-packets in C?
- 👉 Tree grammar would grow quickly and would still generate code of rather poor quality!

Approach

Application of a Conventional Code Selection

- Tree grammar produces LIR containing machine operations that use entire registers as operands
- Tree grammar produces no bit-packet operations
- Using rules like, e.g.,

```
dreg: tpm_BinaryExpAND ( dreg, const )
```

```
dreg: tpm_BinaryExpSHR ( dreg, const )
```

the expression `(c >> 4) & 0x7` would naively be covered and translated into

```
SH d_0, d_c, -4; AND d_1, d_0, 7;
```

Subsequent LIR Optimization

- Detects operations that extract / insert bit-packets and creates corresponding **extr** / **insert** bit-packet operations.

Classical Data Flow Analysis

Problem

- Classical data flow analyses (*DFA*) allow reasoning about the *flow of information* at the register-level:
 - 👍 Which operation uses / defines a certain piece of data residing in a certain register?
 - 👍 Which operations are data flow-dependent?
- Classical data flow analyses allow no statements about
 - 👎 the *value of information*, i.e., the potential value that a register can have at a certain point of time, or about
 - 👎 the potential value that a *part of a register* can have at a certain point of time.

Bit-True Value Flow Analysis

Value Flow Analysis (VFA)

- Analyzes data flow, similar to DFA
- but performs additional estimations about the contents of the memory cells involved in the computations.

Bit-True Data and Value Flow Analysis (BDVFA)

- Value flow analysis is done for each individual bit of the involved memory cells.
- ☞ Allows statements about the potential value of each bit of a memory cell at a certain point of time.
- ☞ ***In the following: Presentation of a BDVFA with multi-valued logic for registers as supported memory cells.***

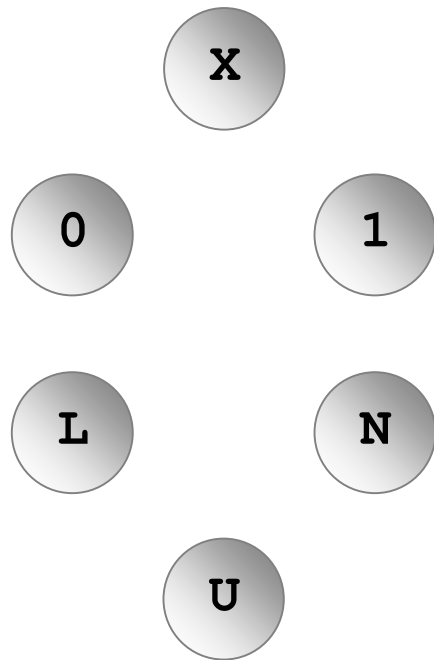
Data and Value Flow Graph (DVFG)

Definition (*Data and Value Flow Graph*)

Let F be an LIR function. The *Data and Value Flow Graph (DVFG)* of F is a directed graph $DVFG = (V, E, d, u)$ with

- Node set V identical with that of classical DFGs (☞ cf. chapter 6)
- Let $op_i(p_{i,1}, \dots, p_{i,n})$ and $op_j(p_{j,1}, \dots, p_{j,m})$ be two operations of F with parameters $p_{i,x}$ and $p_{j,y}$, resp. Let v_i and v_j be the nodes representing op_i and op_j . For each use of a register r by $p_{j,y}$ that is defined by $p_{i,x}$, the DVFG contains an edge $e = (v_i, v_j) \in E$.
- d and u provide bit-true value estimates for the edges $e \in E$ (*Down* and *Up* values). Let r be that register that is modeled by e , and let r have a width of k bits. Then, d and u are functions $d \mid u: E \rightarrow \mathcal{L}_4^k$ with \mathcal{L}_4 being a partial order representing the potential value of a single bit.

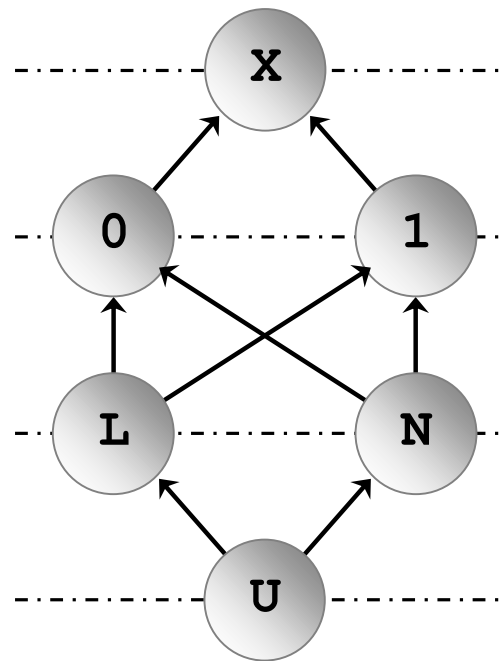
The Partial Order \mathcal{L}_4 (1)



Per bit of a register, an element from \mathcal{L}_4 is used to model which value this bit can have:

- **0** – The considered bit has the value 0
- **1** – A bit has a value of 1
- **U** – A bit's value is completely unknown
- **X** – The value of a bit is irrelevant (*don't care*)
- **L** – The value of a bit is unknown, but its provenience (*Location*) is known
- **N** – Like **L**, but the bit with known provenience has been inverted once (*Negated location*)

The Partial Order \mathcal{L}_4 (1)



\mathcal{L}_4 is a partial order:

- A ' $<$ ' operator defines a less-than relation between elements from \mathcal{L}_4 as shown by the directed edges in the Hasse chart
 - \mathbb{U} provides least information and is thus the smallest element according to the $<$ operator
 - \mathbb{X} provides most information and is thus the largest element according to the $<$ operator
- Hasse chart has four horizontal levels


☞ \mathcal{L}_4

Examples for \mathcal{L}_4 (1)

For some edge e , let r be an 8-bit register that is modeled by e .



Graphical notation:  represents up value,  down value

Exemplary labels for edge e and their interpretation:

 UUUUUUUU The values of all bits of r are completely unknown.

 00101010 The value of r is 42.

 0010X010 Bit 3 of r is irrelevant; r can be equal to 34 or 42.

 XXXXXXXX r is completely irrelevant
 r has no effect on the data flow at all

Examples for \mathcal{L}_4 (2)

In addition, r' is an 8-bit register that represents an input value for the considered LIR function F (e.g., a function parameter).

Exemplary labels for edge e and their interpretation:

↑ `00Lr',200000`

The value of bit 5 of r is unknown. But it is identical with the value of bit 2 of r' .

↑ `00Nr',400000`

The value of bit 5 of r is unknown. But it is identical with the negated value of bit 4 of r' .

↑ `00Lr',2Lr',1Lr',0000`

r contains a bit-packet consisting of bits 2 to 0 of r' , starting at bit position 3 inside r .

Arithmetic in \mathcal{L}_4 – Conjunction

\wedge	1	0	L_i	N_i	U	X
1	1	0	L_i	N_i	U	X
0	0	0	0	0	0	0
L_i	L_i	0	L_i	0	U	U
N_i	N_i	0	0	N_i	U	U
U	U	0	U	U	U	U
X	X	0	U	U	U	X

Remark:

If L/N values with different provenience are combined, the conjunction always produces U.

Arithmetic in \mathcal{L}_4 – Disjunction

\vee	1	0	L_i	N_i	U	X
1	1	1	1	1	1	1
0	1	0	L_i	N_i	U	X
L_i	1	L_i	L_i	1	U	U
N_i	1	N_i	1	N_i	U	U
U	1	U	U	U	U	U
X	1	X	U	U	U	X

Remark:

If L/N values with different provenience are combined, the disjunction always produces U.

Arithmetic in \mathcal{L}_4 – Negation

\neg	1	0	L_i	N_i	U	X
	0	1	N_i	L_i	U	X

Workflow of the BDVFA

Given

- A low-level intermediate representation *LIR* to be optimized

Two-Phased Approach

- For each function *F* of *LIR*:
 - Compute the initial data and value flow graph $D = (V, E, \emptyset, \emptyset)$ of *F* with empty down and up functions *d* und *u*
 - Determine down values $d(e)$ of all edges by applying forward analysis
 - Determine up values $u(e)$ of all edges by applying backward analysis

Forward Analysis

Goal

- To compute only \Downarrow values for D
- \Downarrow value $d(e)$ represents the bit-true result of a node $v \in V$ (i.e., of an outgoing edge of v), if the operator of v is applied to all its operands (i.e., to the \Downarrow values of incoming edges of v).

Approach

- (Repeated) traversal through D following the edges' direction
 - ☞ “*Forward*” Analysis
- For each currently visited node $v \in V$:
 - Apply *Forward Simulation* of v 's operator to all \Downarrow values of all incoming edges
 - Store new \Downarrow values for the outgoing edges of v

Workflow of Forward Analysis (1)

- queue<DVFG_node> $q = \langle \text{set of all source nodes of } D \rangle$;
- $d(e) = \mathbf{U}^*$ for all edges $e \in E$;
- while (! $q.empty()$)
 - DVFG_node $v = \langle \text{first element of } q \rangle$; $q.remove(v)$;
 - $E_{out} = \{ e \in E \mid e = (v, v_x) \}$; $E_{in} = \{ e \in E \mid e = (v_x, v) \}$;
 - if ($\langle v \text{ represents a constant number } c \rangle$)
 - $d'(e) = \{0, 1\}^* = \langle \text{binary representation of } c \rangle$ for all $e \in E_{out}$;
 - else
 - if ($\langle v \text{ represents an unknown input variable } i \text{ of } F \rangle$)
 - $d'(e) = \{\mathbf{L}_i\}^* = \langle \text{bit locations of } i \rangle$ for all $e \in E_{out}$;
 - else
 - $d'(e) = \langle \text{forward simulation of } v \rangle$ for all $e \in E_{out}$;

Workflow of Forward Analysis (2)

- while (!q.empty())
 - ... *<see previous slide>*;
 - for (*<all edges $e = (v, v_x) \in E_{out}$ >*)
 - if (*<current $d(e)$ is bit-wise less than $d'(e)$ according to $<$ operator in \mathcal{L}_4 >*)
 - $d(e) = d'(e)$;
 - if (!q.contains(v_x))
 - $q.insert(v_x)$;

Remarks

- Constants and input variables provide initial values for the \Downarrow values of leaf nodes with elements 0 , 1 and \perp .
- For a k -bit register r modeled by edge e , forward analysis firstly computes an intermediate \Downarrow value $d'(e)$.
- $d(e) \in \mathcal{L}_4^k$ is only set to this intermediate value $d'(e) \in \mathcal{L}_4^k$ if
 - for at least one bit position i ($0 \leq i \leq k$) $d_i(e) < d'_i(e)$ holds
 - AND
 - for no bit position i ($0 \leq i \leq k$) $d'_i(e) < d_i(e)$ holds
- ☞ Since only steadily increasing bit values are assigned to the \Downarrow values in the course of the analysis, each node $v \in V$ is only added a finite number of times to the queue q .
- ☞ Forward analysis terminates necessarily, run-time complexity is $O(|E|)$.

Forward Simulation (1)

Goal

- For each node $v \in V$ that represents a machine operation op in the LIR of F , and for each outgoing edge $e \in E_{out}$:

Forward simulation computes the \Downarrow value of e depending on the \Downarrow values of all incoming edges $e_{in,1}, \dots, e_{in,N} \in E_{in}$:

$$d'(e) = FS_{op}(d(e_{in,1}), \dots, d(e_{in,N}))$$

Challenge

- To provide a bit-true simulation function FS_{op} for each possible machine operation of LIR based on \mathcal{L}_4^k .
- FS_{op} must model the behavior of op for the considered target processor exactly and bit-true!

Forward Simulation (2)

Overall Approach

- Each and every machine operation op can in principle be described at the bit-level using the Boolean standard operators \wedge , \vee and \neg .
- ☞ Describe FS_{op} as formula over the operators \wedge , \vee and \neg of \mathcal{L}_4^k , in analogy to a Boolean description of op .

Bitwise Logical Operators

- Bitwise logical machine operations op (e.g., AND, OR, NOT, XOR, NOR, NAND, ...) can easily be modeled in \mathcal{L}_4^k using \wedge , \vee and \neg .
- ☞ Approach clear.

Forward Simulation (3)

Arithmetical Operations

- Derivation of a representation in \mathcal{L}_4^k for arithmetical machine operations *op* laborious, but doable.

☞ Example addition:

- Half-adder: Adds bits $a, b \in \mathcal{L}_4$, produces bits $s, c \in \mathcal{L}_4$:

$$s = a \otimes b = (a \wedge \neg b) \vee (\neg a \wedge b); c = a \wedge b;$$

- Full-adder: Adds bits $a, b, c_{in} \in \mathcal{L}_4$, produces bits $s, c_{out} \in \mathcal{L}_4$:

$$s = (a \otimes b) \otimes c_{in}; c_{out} = (a \wedge b) \vee (a \wedge c_{in}) \vee (b \wedge c_{in});$$

- k -bit addition in \mathcal{L}_4^k :

Successively apply formulae of full-adder to bit positions $0, \dots, k$ and compute bits of the resulting sum.

Forward Simulation (4)

Register Transfer Operations

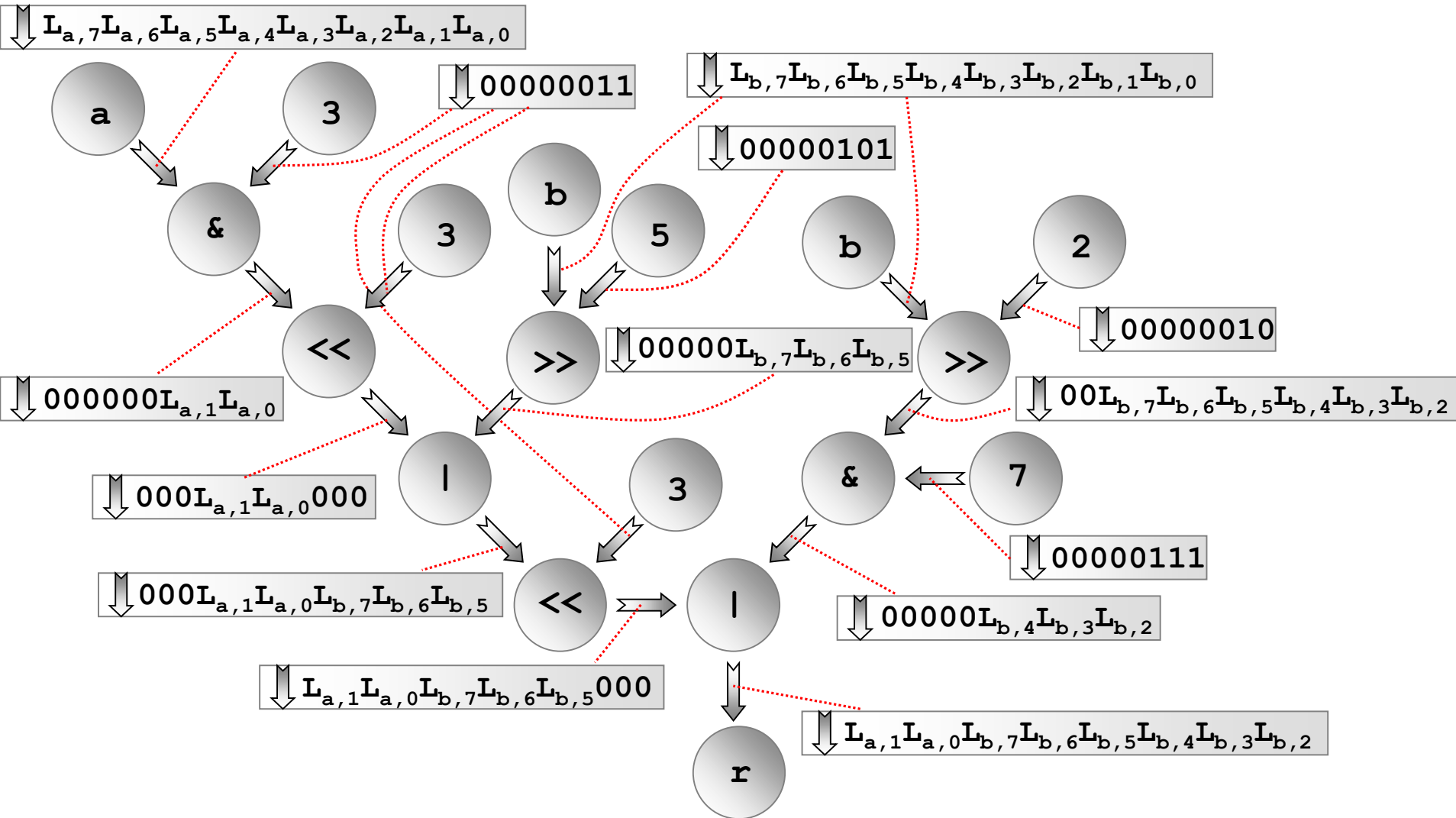
- Copying of register contents (*Register-Register-Moves*) is easily modeled in \mathcal{L}_4^k by copying of \Downarrow values.

Memory Transfer Operations

- Since the BDVFA provides bit-true data and value flow information explicitly only for registers and NOT for external memories,
 - store-operations anyways do not generate any \Downarrow values, since they are typically sinks in the DVFG,
 - load-operations only generate \mathbb{U}^* as \Downarrow values.

☞ ***Other classes of machine operations need to be modeled similarly.***

Example Forward Simulation



Backward Analysis (1)

Motivation and Goals

- Conjunction, disjunction and negation in \mathcal{L}_4 produce \mathbf{x} (don't care) only if one of their operands is already \mathbf{x} .
- ☞ Since the \Downarrow values of source nodes of the DVFG only consist of 0, 1 and \mathbf{L} , the forward analysis never generates \mathbf{x} .
- Goal of backward analysis is thus to generate \mathbf{x} for individual bit positions under consideration of the \Downarrow values computed so far.

Backward Analysis (2)

Approach

- (Repeated) traversal through D in reverse direction of the edges
 - ☞ “Backward” Analysis
- For each currently visited node $v \in V$:
 - Answering of the question which bits of the \uparrow values of incoming edges of v are irrelevant, so that still the exact \uparrow values of v 's outgoing edges are created.
 - Apply *Backward Simulation* of v 's operator to \uparrow values of the incoming and outgoing edges
 - Store new \uparrow values for the incoming edges of v

Workflow of Backward Analysis

- queue<DVFG_node> $q = \langle \text{set of all sink nodes of } D \rangle$;
 $u(e) = d(e)$ for all edges $e \in E$;
- while (! $q.empty()$)
 - DVFG_node $v = \langle \text{first element of } q \rangle$; $q.remove(v)$;
 $E_{out} = \{ e \in E \mid e = (v, v_x) \}$; $E_{in} = \{ e \in E \mid e = (v_x, v) \}$;
 - for ($\langle \text{all edges } e = (v_x, v) \in E_{in} \rangle$)
 - $u'(e) = \langle \text{backward simulation of } v \text{ using } E_{out} \text{ and } E_{in} \setminus \{e\} \rangle$;
 - if ($\langle \text{current } u(e) \text{ is bit-wise less than } u'(e) \text{ according to } \langle \text{operator in } \mathcal{L}_4 \rangle$)
 - $u(e) = u'(e)$;
 - if (! $q.contains(v_x)$)
 $q.insert(v_x)$;

Backward Simulation (1)

Goal

- For each node $v \in V$ that represents a machine operation op in the LIR of F , and for each incoming edge $e \in E_{in}$:

Backward simulation computes the \Uparrow value of e depending on the \Uparrow values of all outgoing edges $e_{out,1}, \dots, e_{out,N} \in E_{out}$ and of all incoming edges except e itself:

$$u'(e) = BS_{op}(u(e_{out,1}), \dots, u(e_{out,N}), u(e_{in} \in E_{in} \setminus \{e\}))$$

In Analogy to Forward Simulation

- A bit-true simulation function BS_{op} must be provided for each possible machine operation of LIR based on \mathcal{L}_4^k .

Backward Simulation (2)

Overall Approach

- Exploitation of neutral elements and of zero elements of operators in order to identify don't cares.

Bitwise Logical Operators

- Let $b_{1,k}$ and $b_{2,k} \in \mathcal{L}_4$ be single bits at position k of the \uparrow value of the two operands of a logical operation and let $b_{3,k}$ be the k -th bit of the \uparrow value of the operation's result.

- For $b_{2,k} = b_{3,k} = 0$:


$$b_{1,k} \text{ AND } b_{2,k} = b_{3,k} \quad \Leftrightarrow \quad b_{1,k} \text{ AND } 0 = 0 \quad \Leftrightarrow \quad \mathbf{x} \text{ AND } 0 = 0$$

- Analogously for OR: $b_{1,k} \text{ OR } 1 = 1 \quad \Leftrightarrow \quad \mathbf{x} \text{ OR } 1 = 1$

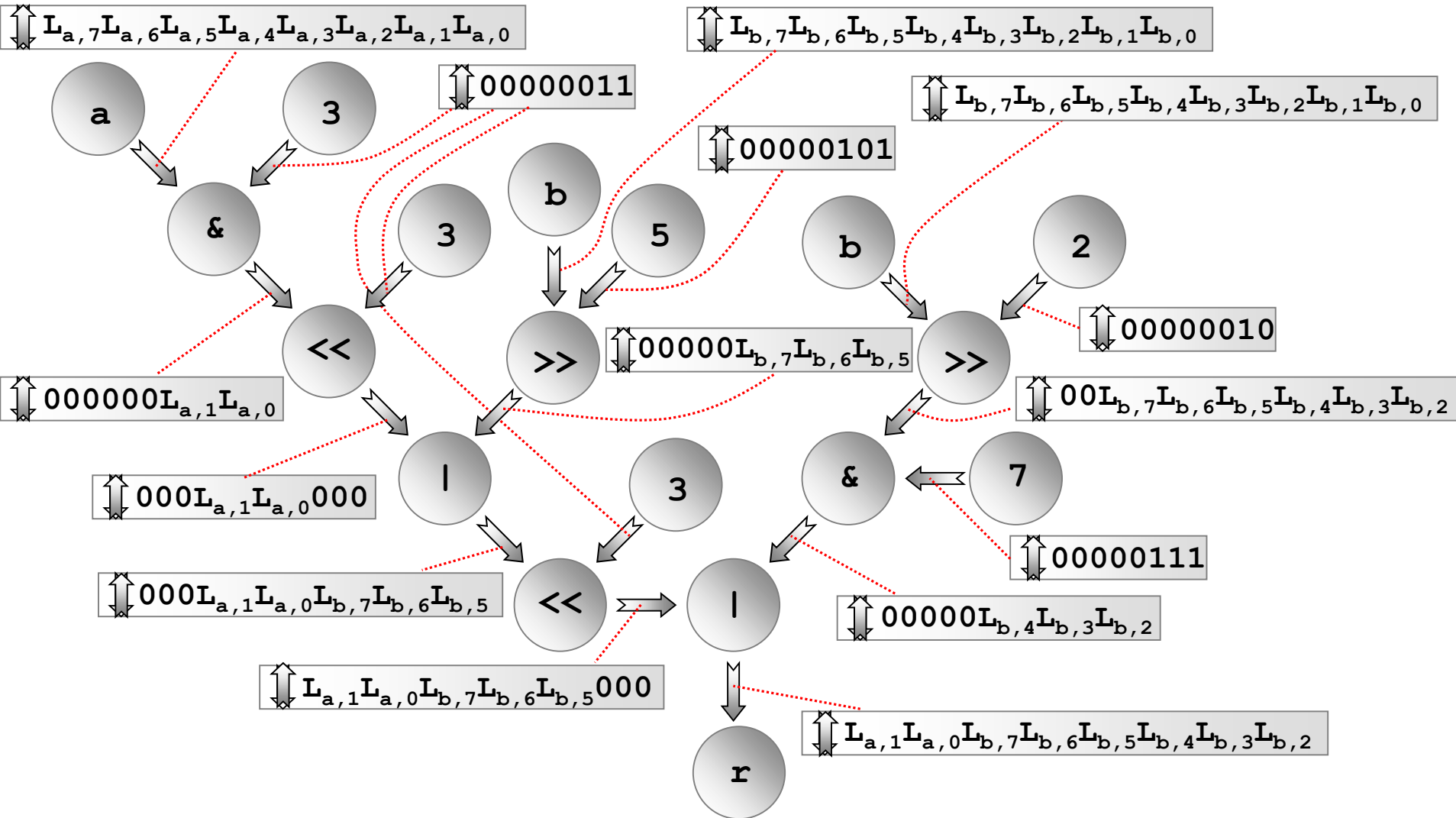
☞ Approach for other logical operations similar.

Backward Simulation (3)

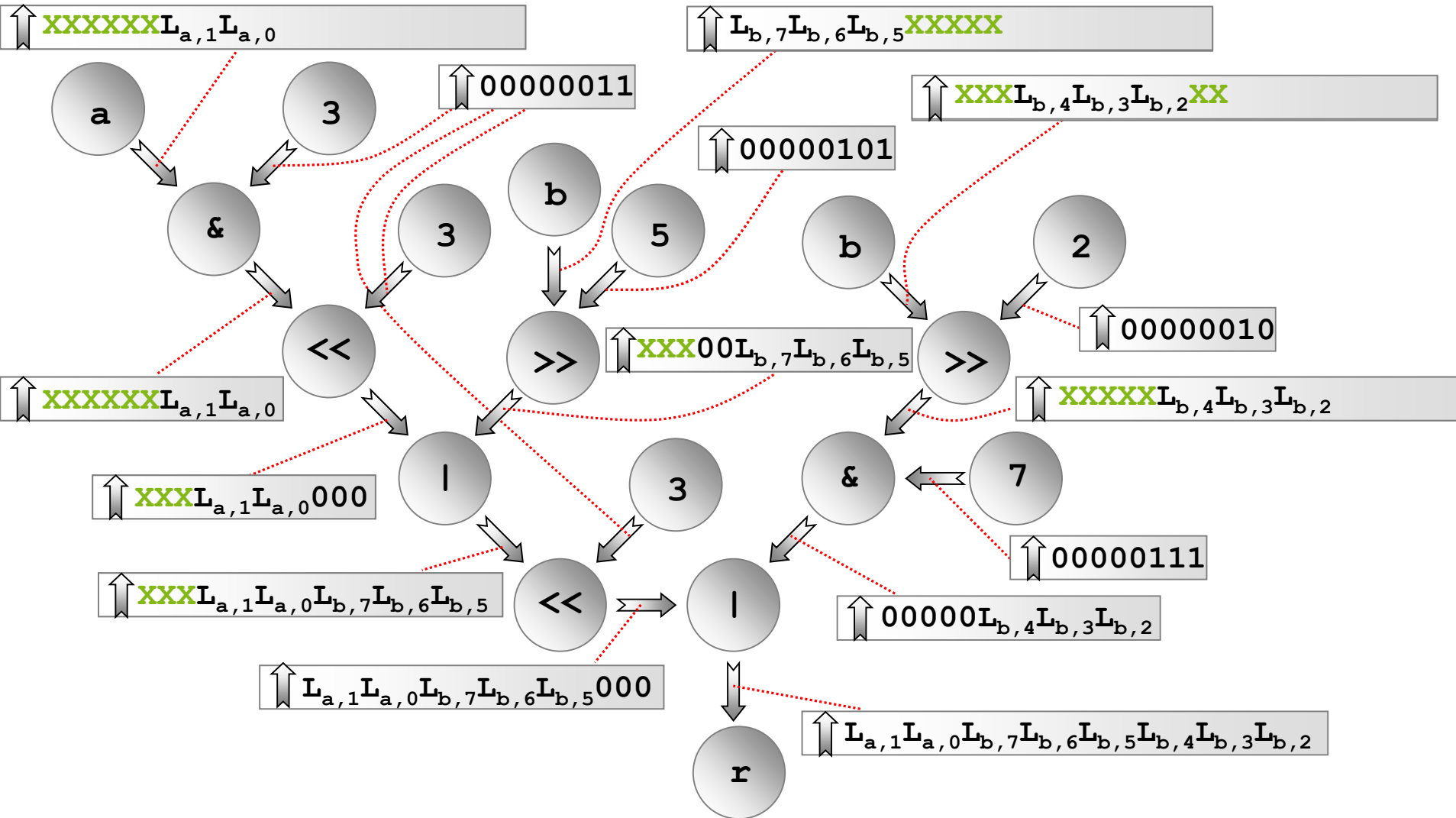
Arithmetical Operations

- Identification of irrelevant bits due to complexity of arithmetical operations often difficult.
- ☞ Plain example – Shift operator:
 - $a \ll 3$: Shifts contents of a left by 3 bits
Least-significant 3 bits are filled with 0,
Most-significant 3 bits are shifted out / truncated
☞ In the  value of a , the 3 most-significant bits are x
 - $a \gg 3$: Analogously for least-significant 3 bits, under consideration of arithmetical or logical shifting
- ☞ ***Other machine operations need to be analyzed carefully and must be modeled similarly.***

Example Backward Simulation (1)



Example Backward Simulation (2)



Application of BDVFA: Dead Code Elimination (DCE)

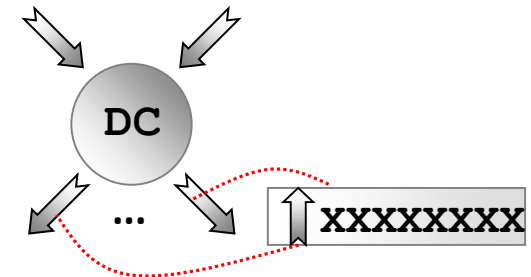
Definition (*Dead Code*)

(LIR-) Operations that compute only values that are not used on any executable path from the operation are called *Dead Code*.

Dead Code and BDVFA

- In the DVFG, individual bits that are completely irrelevant for subsequent computations are set to \mathbf{x} in the \uparrow values of edges.

☞ *An LIR operation whose outgoing edges all carry only \mathbf{x}^* as \uparrow value is dead code.*



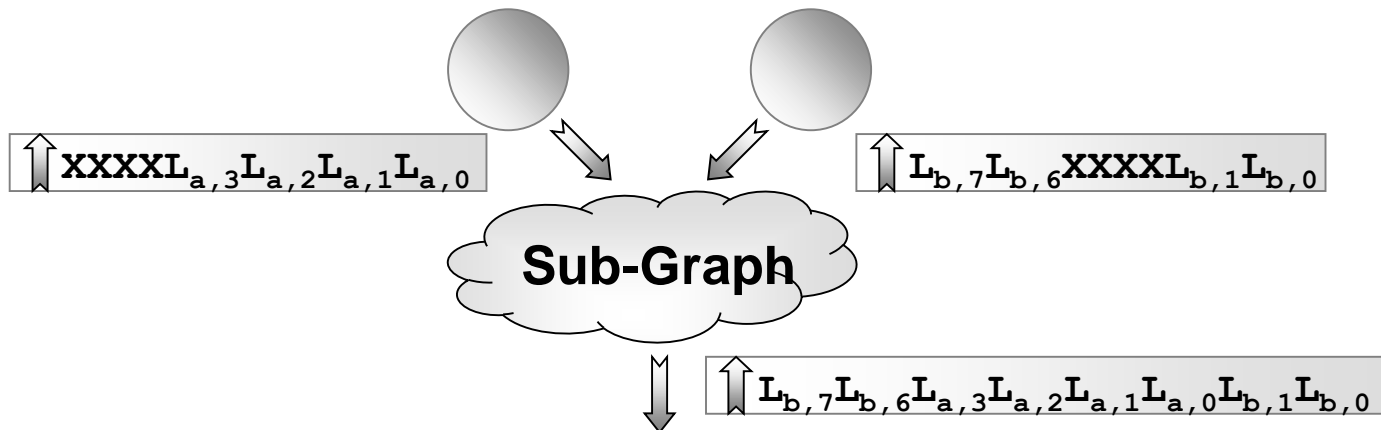
Workflow of Bit-True Dead Code Elimination

- queue<DVFG_node> q ;
- for (<all edges $e = (v, w) \in E$ with $u(e) = \mathbf{x}^*$ >)
 - $q.insert(v)$;
- while (! $q.empty()$)
 - DVFG_node $v = \langle \text{first element of } q \rangle$; $q.remove(v)$;
 - $E_{out} = \{ e \in E \mid e = (v, v_x) \}$; $E_{in} = \{ e \in E \mid e = (v_x, v) \}$;
 - if (($u(e) = \mathbf{x}^*$ for all $e \in E_{out}$) && (< v has no side-effects>))
 - mark v ;
 - for (<all edges $e = (v_x, v) \in E_{in}$ >)
 - $u(e) = \mathbf{x}^*$;
 - if (< v_x is not yet marked>) $q.insert(v_x)$;
- remove all LIR operations associated with marked nodes;

Application of BDVFA: Bit-Packet Insert Operations (1)

Insert Operations and BDVFA

- Insertion of a bit-packet into a register by some arbitrary part of the DVFG directly visible from the \uparrow values:

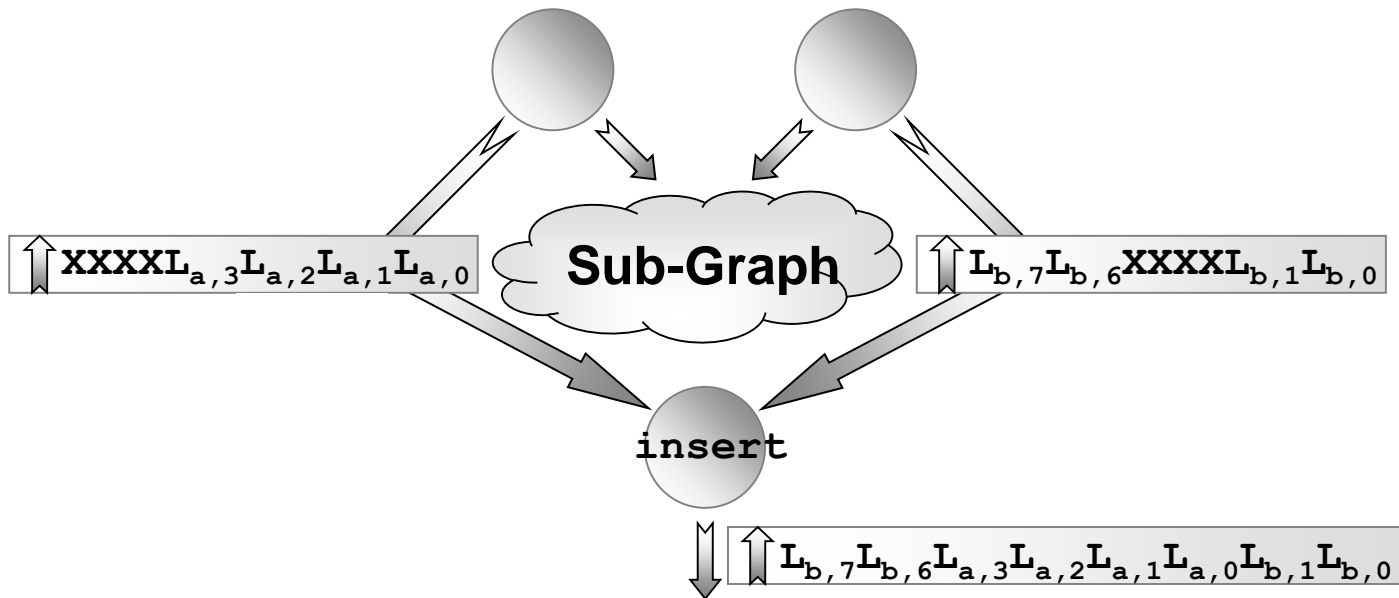


- ☞ *An optimization has to partition the \uparrow value of an edge into disjoint bit-packets according to the edge's L values and needs to generate matching insert operations according to this partitioning.*

Application of BDVFA: Bit-Packet Insert Operations (2)

Insert Operations and BDVFA (ctd.)

- Optimization of the example by generation of an insert operation and by adjustment of edges:

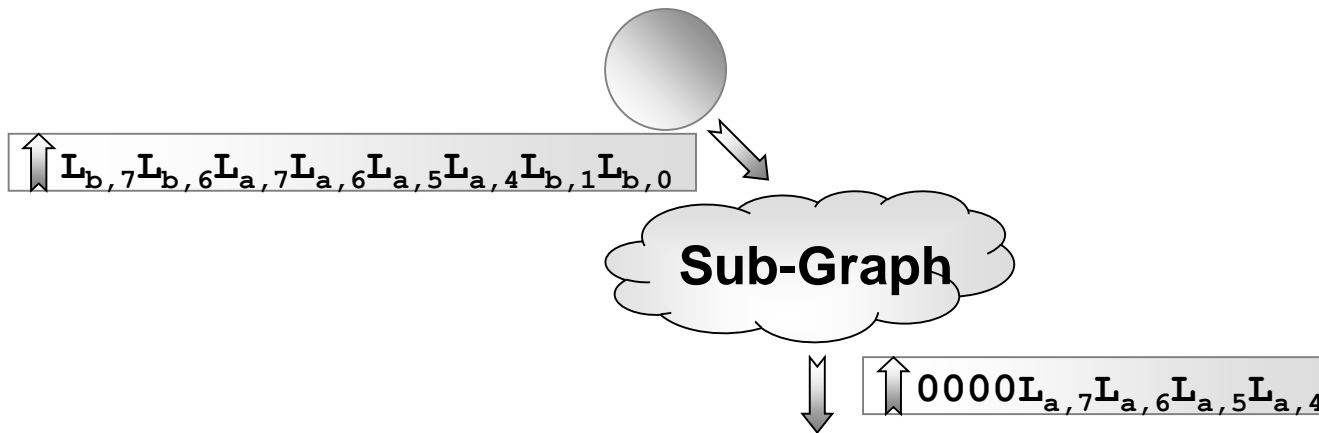


- ☞ *Provided that no further edges leave the sub-graph after adjustment of edges, the entire sub-graph will be removed by a subsequent DCE!*

Application of BDVFA: Bit-Packet Extract Operations (1)

Extract Operations and BDVFA

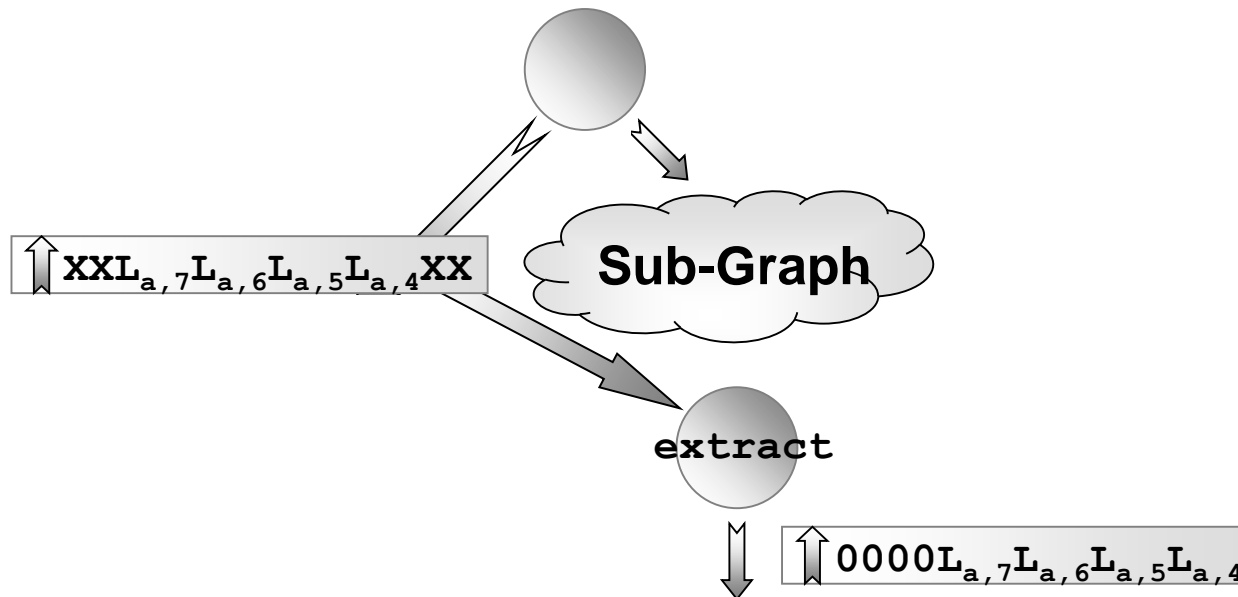
- An extraction of a bit-packet by some arbitrary sub-graph of the DVFG is again visible from the \uparrow values:



Application of BDVFA: Bit-Packet Extract Operations (2)

Extract Operations and BDVFA (ctd.)

- Optimization of the example by generation of an extract operation and by adjustment of edges:



☞ Possibly, the sub-graph can again be removed by a DCE.

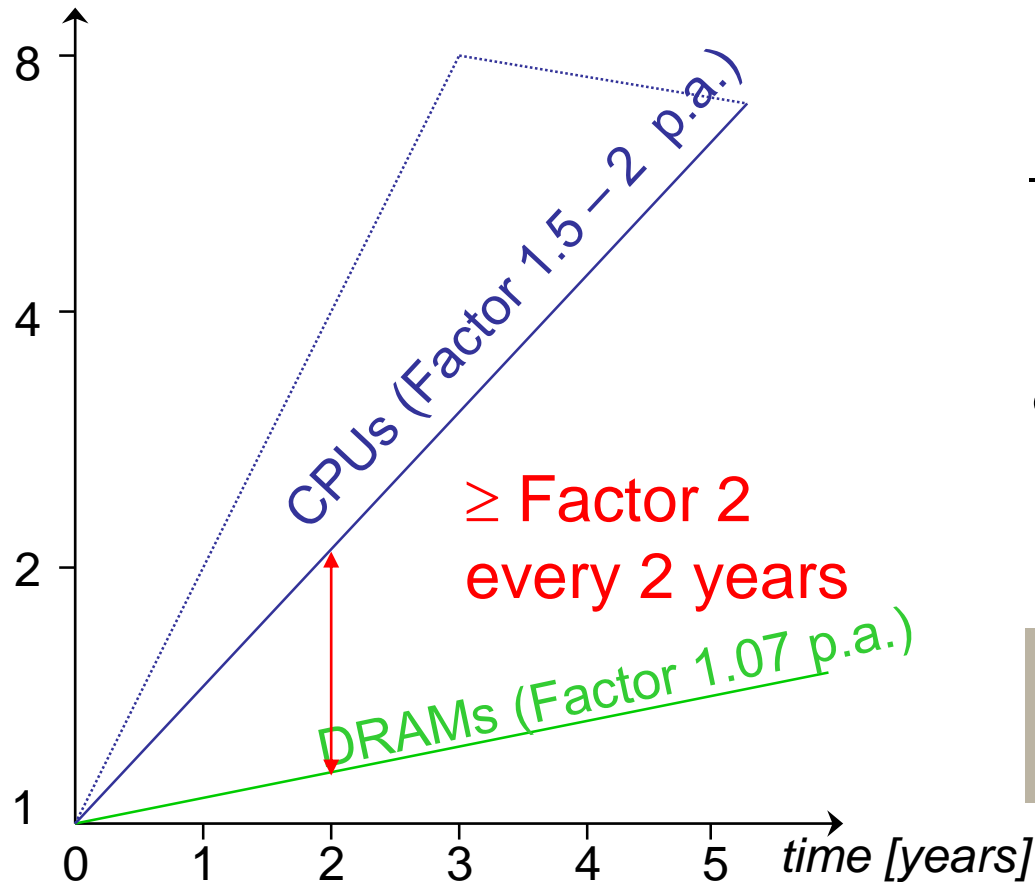
Chapter Contents

7. LIR Optimizations and Transformations

- Generation of Bit-Packet Operations for NPUs
 - Motivation of Bit-True Data and Value Flow Analyses
 - Partial Order \mathcal{L}_4
 - Bit-True Analysis: Forward and Backward Simulation
 - Bit-True Optimizations: Dead Code Elimination; Generation of insert/extract operations
- Scratchpad Memory Optimizations
 - Properties of Main Memories, Caches and Scratchpads
 - Fix SPM Allocation (Functions, global Variables)
 - Fix SPM Allocation (Functions, Basic Blocks, global Variables)
 - SPM Allocations for Multi-Process Applications (partitioned, exclusive, hybrid)

Properties of Today's Memories (1)

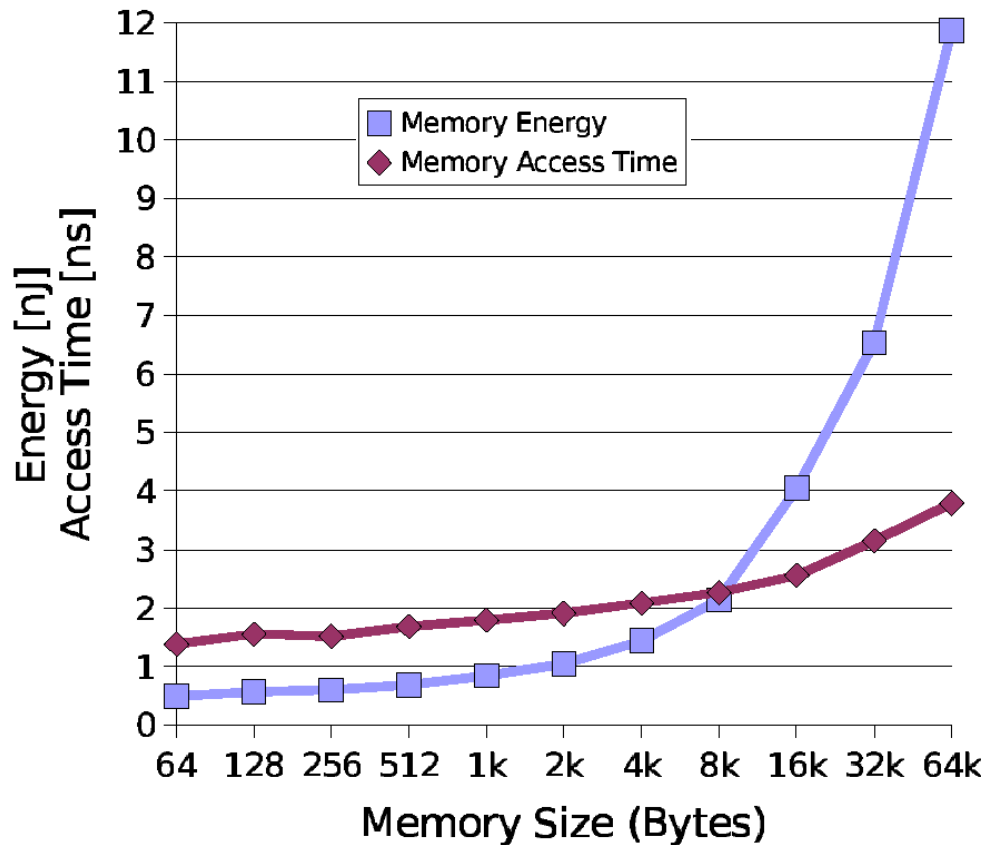
Relative speed



- Speed gap between CPUs and DRAMs doubles every 2 years.
- Fast CPUs are massively slowed down by slow memories
- ☞ "Memory Wall" problem

[P. Machanik. Approaches to Addressing the Memory Wall. Technical Report, University of Brisbane, Nov 2003]

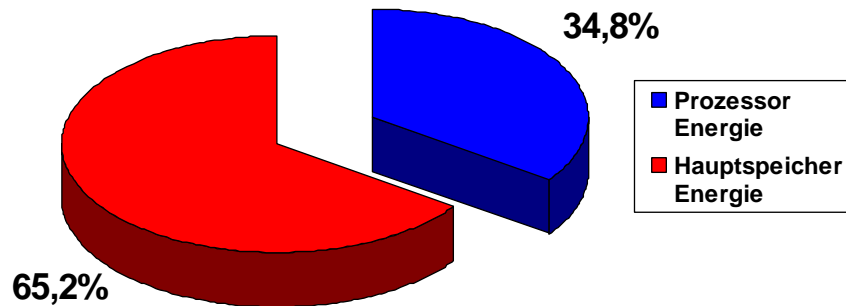
Properties of Today's Memories (2)



- With increasing size of a memory, a single memory access consumes disproportionately high energy.
 - With increasing size, memory accesses also take proportionally longer.
- ☞ *Fabrication technology of memories suggests to use small memories!*

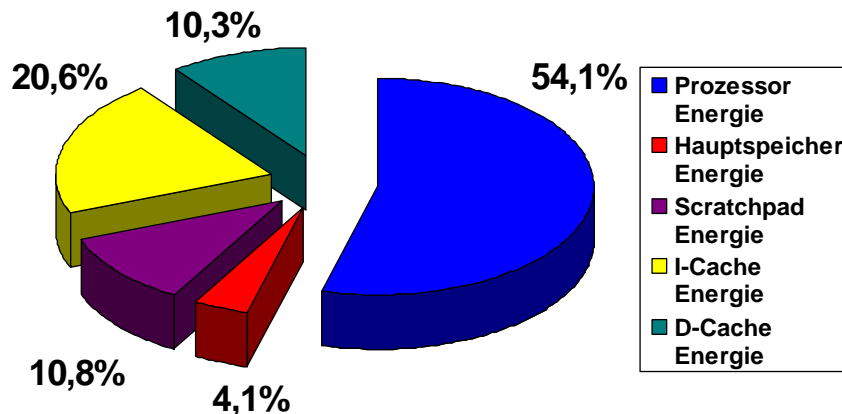
Properties of Today's Memories (3)

- ARM7 Mono-Core without Cache:



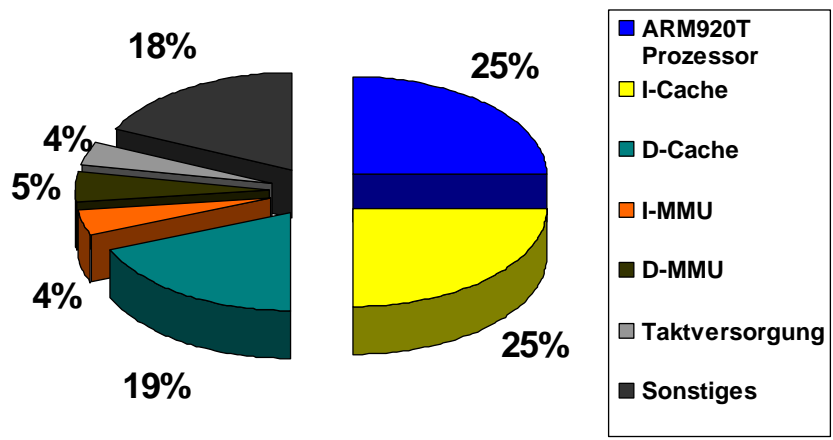
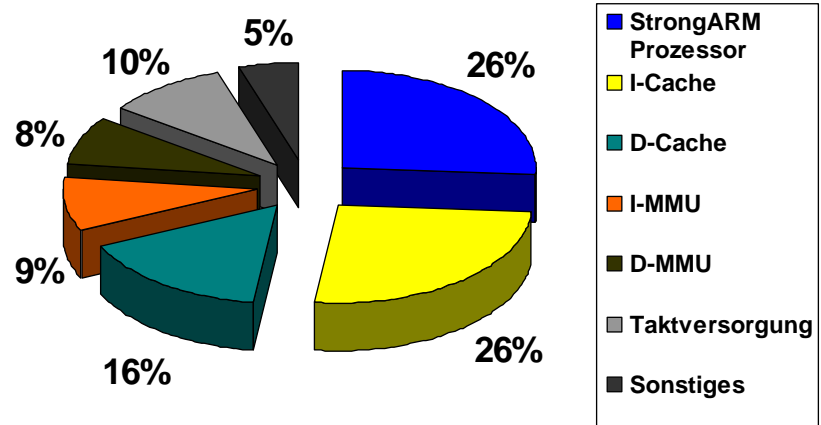
- Memory subsystem often consumes more than 50% of the entire system's total energy budget.
- Pie charts show averages over more than 160 different energy measurements each.

- ARM7 Multi-Core with Caches:



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

Properties of Today's Memories (4)



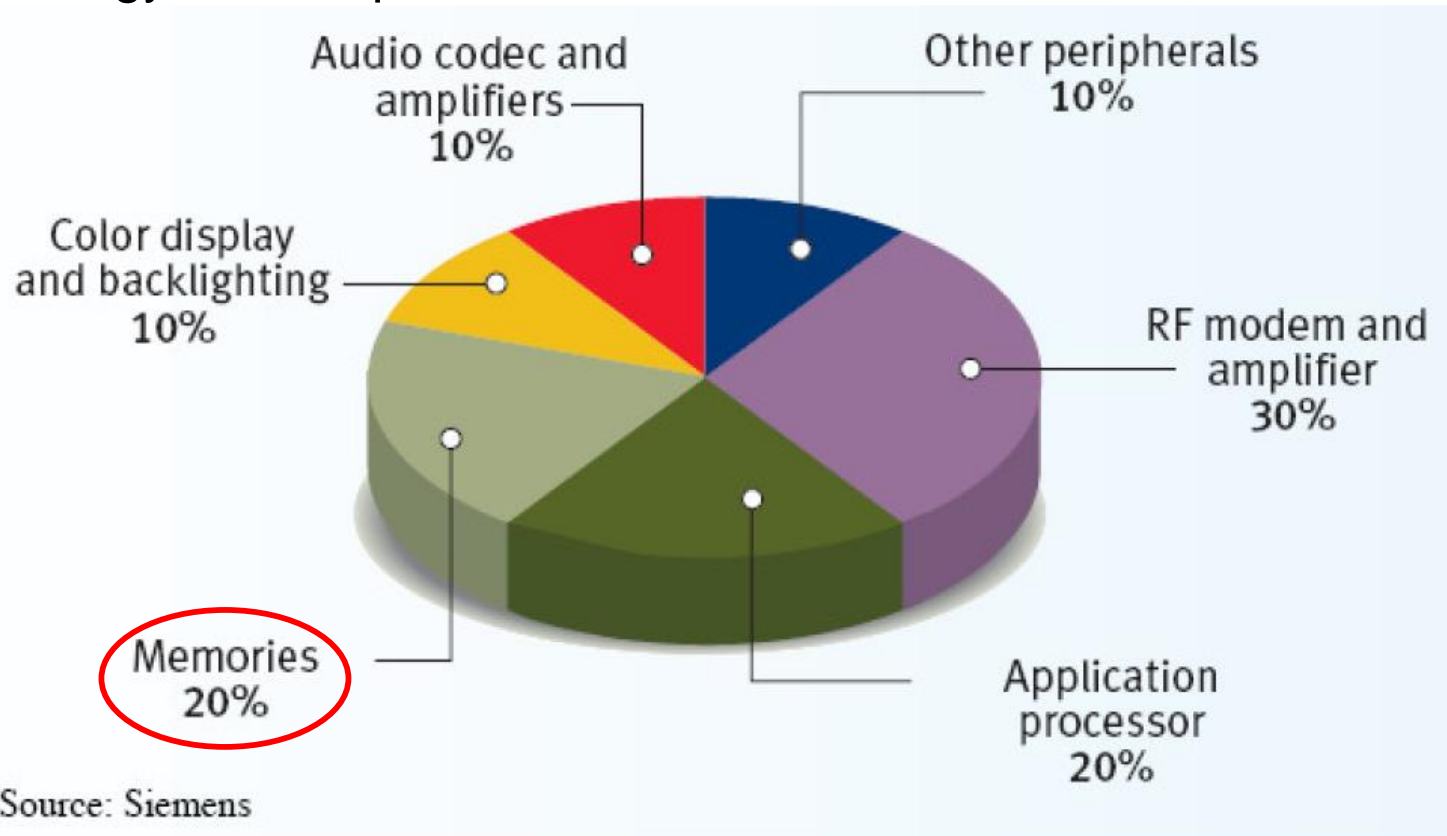
[O. S. Unsal, I. Koren, C. M. Krishna, C. A. Moritz. University of Massachusetts, Amherst, 2001]

- Orders of magnitudes for energy consumption of memories also confirmed by other independent groups from industry and academia.

[S. Segars (ARM Ltd.). Low power design techniques for microprocessors. ISSCC 2001]

Properties of Today's Memories (5)

- Energy consumption of mobile devices

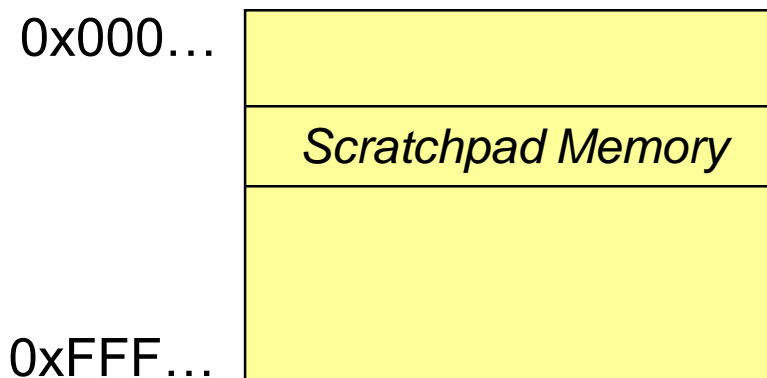


[O. Vargas (Infineon). Minimum power consumption in mobile-phone memory subsystems. Pennwell Portable Design, Sep. 2005]

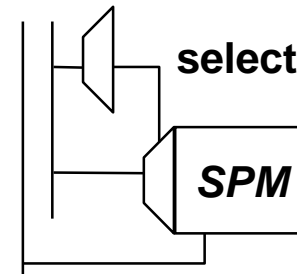
Scratchpad Memories

Structure

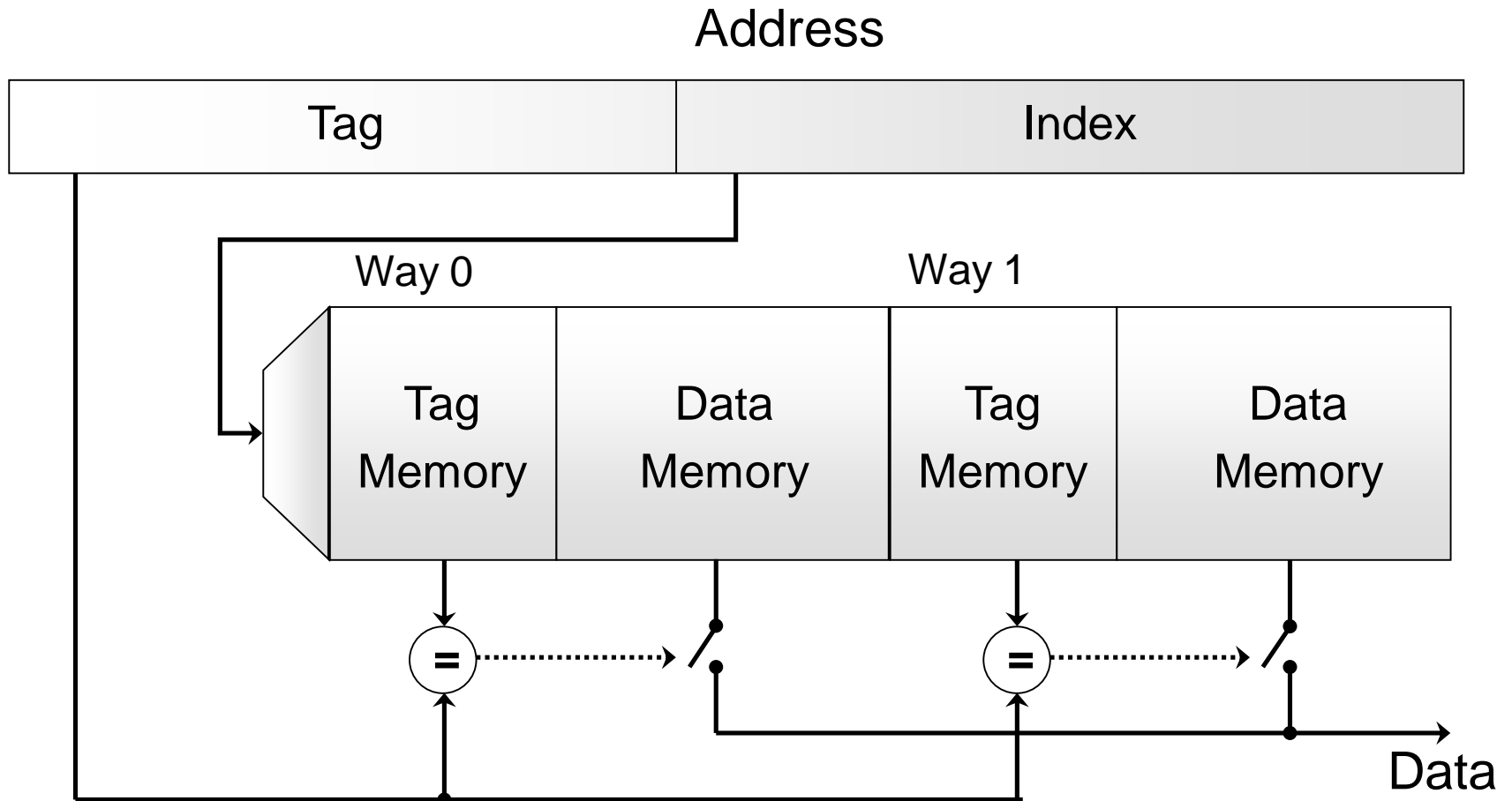
- Scratchpads (SPMs) are small, physically separate memories
- They are mostly placed on the same chip as the processor (so-called *on-chip* memories)
- ☞ *Due to little size and on-chip placement: Extremely fast and energy-efficient memories*
- Are seamlessly mapped into the processor's address space:



- Access by a simple address decoder that recognizes addresses on the bus from the SPM's address region:



Recall: Structure of Set-Associative Caches



Properties of Scratchpad Memories (1)

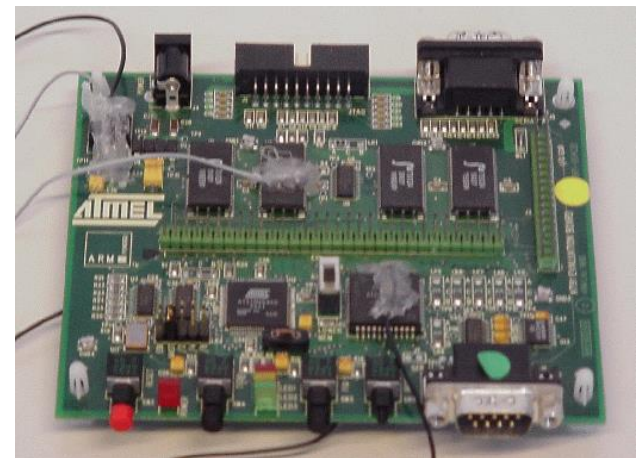
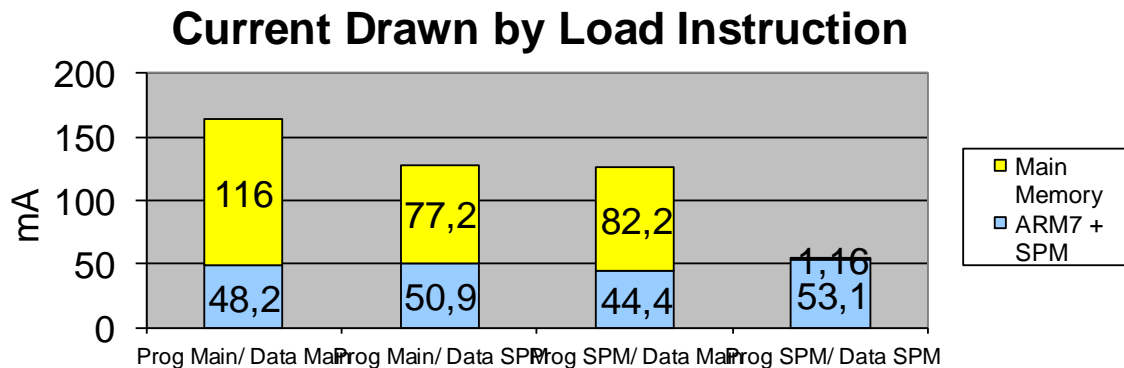
Predictability

- Each SPM access takes only constant time, usually 1 clock cycle.
- In contrast to this: A cache access takes a variable time, depending on whether it results in a *cache hit* or *cache miss*.
- ☞ Run-time behavior of scratchpads is 100% predictable while behavior of caches is difficult to impossible to predict.
- ☞ Caches are severely limited regarding real-time capabilities while SPMs are frequently used in the domain of hard real-time systems.

Properties of Scratchpad Memories (2)

Current Drain Compared to Main Memories

- Measurements using real hardware (Atmel ARM7 evaluation board) show that, e.g., a load instruction draws a factor 3 less current if both the instruction and the data to be loaded reside in SPM instead of the off-chip main memory:

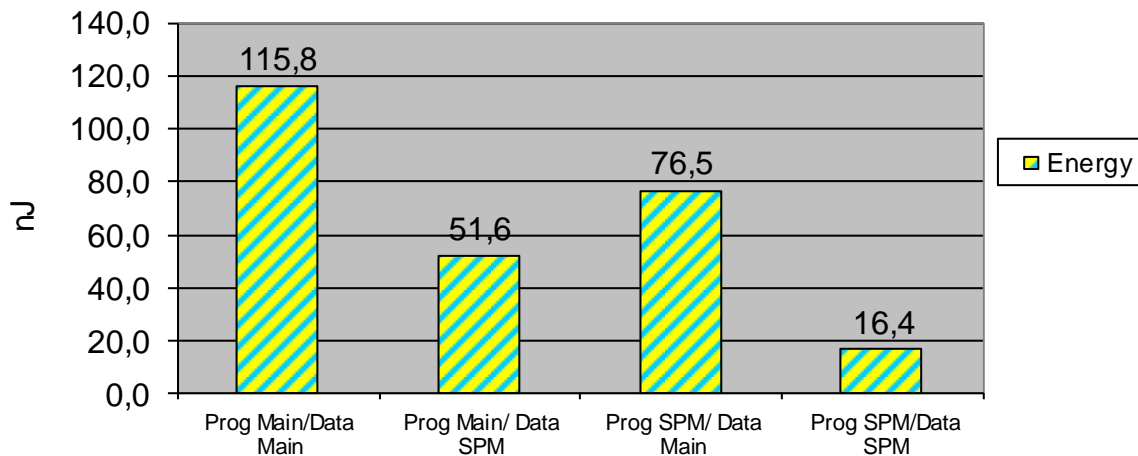


Properties of Scratchpad Memories (3)

Energy Consumption Compared to Main Memories

- Similar measurements using the same hardware reveal that energy consumption of the load instruction can be reduced by factor of 7:

Energy Consumed by Load Instruction



Recall:

$$E = \int P \, dt = \int (V * I) \, dt$$

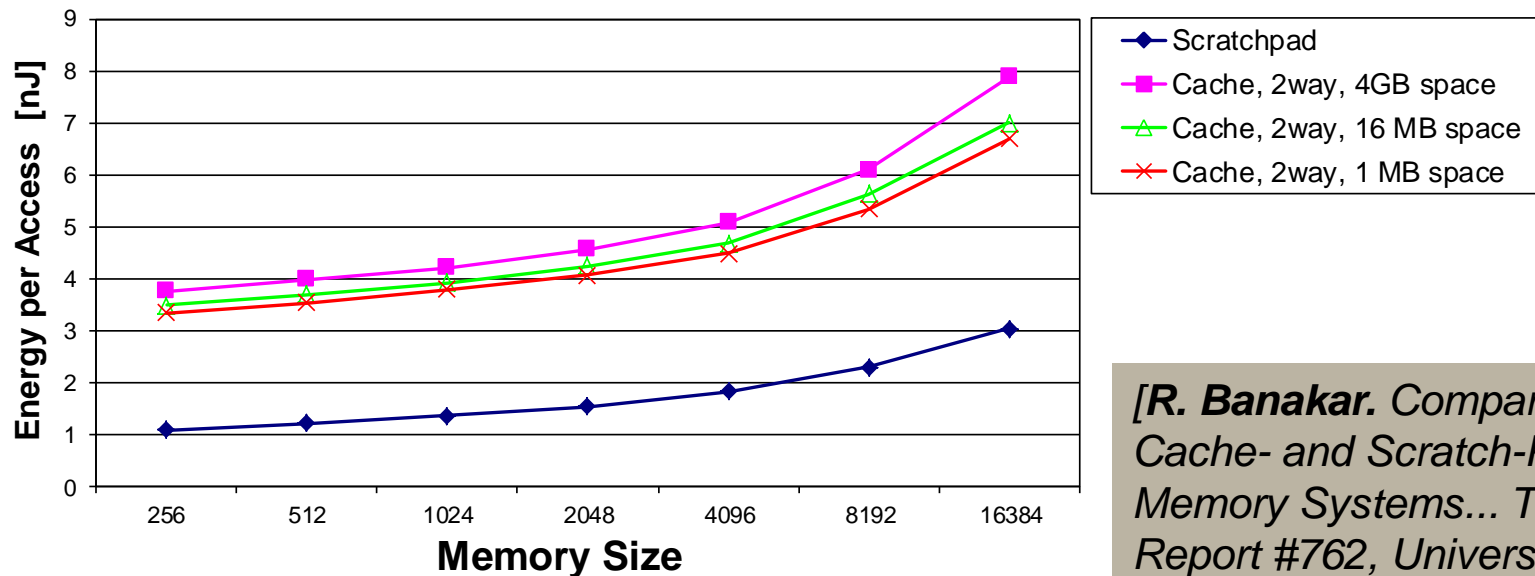
Assumption: Supply voltage is constant and drawn current does not vary too much over time

$$E \approx V * I * t$$

Properties of Scratchpad Memories (4)

Energy Consumption Compared to Caches

- Size and number of tag memories, comparators and multiplexors depends on the size of the cached memory region.
- Energy consumed by these hardware components considerable:



[R. Banakar. Comparison of Cache- and Scratch-Pad based Memory Systems... Technical Report #762, Universität Dortmund, Sep. 2001]

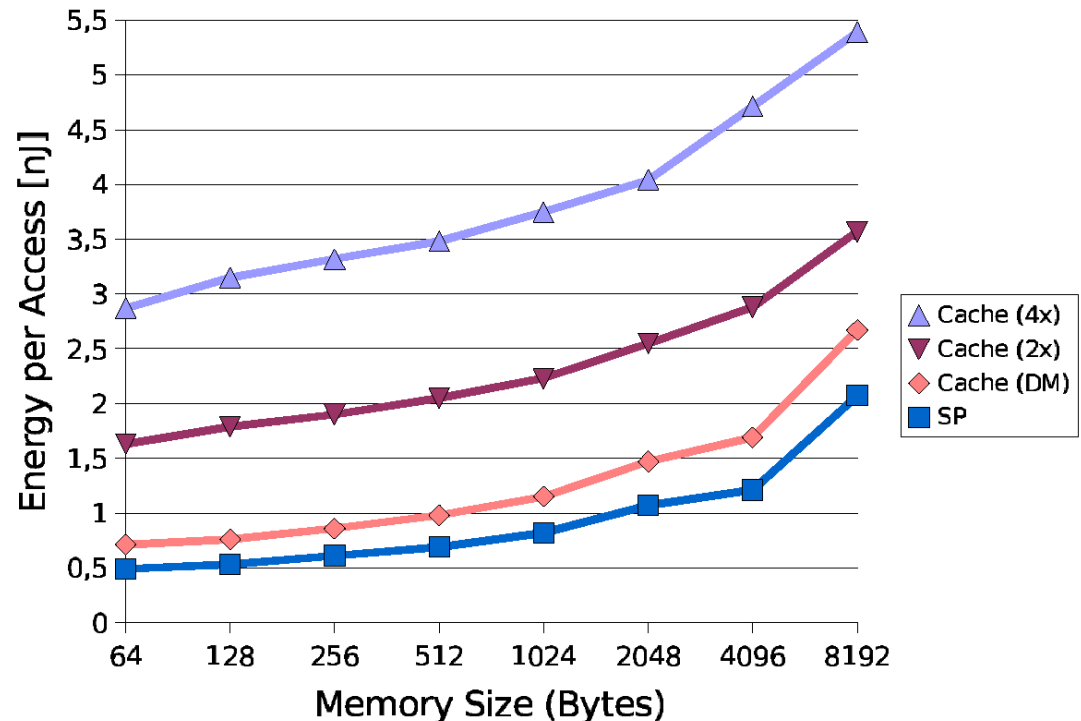
Properties of Scratchpad Memories (5)

Energy Consumption Compared to Caches

- Energy consumption of caches additionally depends heavily on the degree of associativity:

Caution: Underlying technology for this diagram different to that from the previous slide.

Thus, there are deviations in the absolute values shown here and on the previous slide!



Integer-Linear Programming

Technique to Model Linear Optimization Problems

- Optimization of an objective function z under consideration of constraints n_1, \dots, n_m
- Objective function and constraints are linear expressions of the integer decision variables x_1, \dots, x_n

$$z : \sum_{i=1}^n c_i * x_i \quad \rightarrow \text{to be minimized or maximized}$$

$$n_j : \sum_{i=1}^n a_{j,i} * x_i \leq b_j$$

Constants $a_{j,i}, b_j, c_i \in \mathbb{R}$
Variables $x_i \in \mathbb{Z}$

- Optimal resolution of so-called ILPs (*Integer Linear Programs*) using standard ILP solvers (e.g., `lp_solve`, `cplex`)
Complexity: exponential in the worst case, but usually “OK”.

Fix SPM Allocation: Functions & Global Variables (1)

Goal

- Allocation of the code of complete LIR functions and of global variables onto the SPM
(local variables usually lie on the stack and will not be considered here for this reason)
- Compiler determines at *compile-time* which functions and global variables occupy the SPM.

This SPM allocation remains fix during run-time of the optimized program, i.e., the assignment of functions and variables to the SPM does not change at all during the entire run-time.

Fix SPM Allocation: Functions & Global Variables (2)

Definitions

Notation: Upper-case letters \cong constants,
lower-case letters \cong variables

- $MO = \{MO_1, \dots, MO_N\}$ Set of all memory objects to be considered for SPM allocation, i.e., functions F and global variables V , resp.
- $F \cup V$
- S Size of the available SPM in bytes
- S_i Size of memory object MO_i in bytes
- $\Delta E_{i,\text{single}}$ Energy that is saved if MO_i is moved from main memory to SPM, per single execution of $MO_i \in F$ or per single access to $MO_i \in V$, resp.

Fix SPM Allocation: Functions & Global Variables (3)

Definitions (ctd.)

- A_i Total amount of accesses or of executions to MO_i
- $\Delta E_{i,\text{total}}$ Totally saved energy if MO_i is moved from main memory to SPM for the overall execution of the program to be optimized
(= $A_i * \Delta E_{i,\text{single}}$)
- x_i Binary decision variable for MO_i
 $x_i = 1 \Leftrightarrow MO_i$ is moved onto SPM

Fix SPM Allocation: Functions & Global Variables (4)

Determination of Parameter Values

- S : Provided by user, constant
- S_i : Easy to determine using an LIR: Either the sum of the sizes of all machine instructions of a function, or the sum of the sizes of all sub-variables (e.g. for composed types like arrays or structs)
- $\Delta E_{i,\text{single}}$:
 - For $MO_i \in V$: Energy model (☞ *cf. chapter 3*) provides difference between access to main memory and SPM
 - For $MO_i \in F$: Energy model provides difference ΔE_{IFetch} between instruction fetch from main memory or from SPM. Simulation of the program to be optimized yields amount $A_{i,\text{instr}}$ of executed instructions for MO_i
$$\Delta E_{i,\text{single}} = A_{i,\text{instr}} * \Delta E_{\text{IFetch}}$$

Fix SPM Allocation: Functions & Global Variables (5)

Determination of Parameter Values (ctd.)

- A_i : Same simulation that provides $\Delta E_{i,\text{single}}$ can be used to obtain access and execution frequencies for MO_i
- ☞ Prior to the scratchpad optimization of a program, a simulation run is performed in order to determine parameters required for the optimization
- ☞ Such a simulation creates a run-time profile of the program to be optimized. Thus:
This simulation phase before an optimization is called *Profiling*.

Fix SPM Allocation: Functions & Global Variables (6)

ILP Formulation

- Objective function: Maximize energy savings for the whole program

$$z : \sum_{i=1}^N \Delta E_{i,\text{total}} * x_i \rightsquigarrow \max .$$

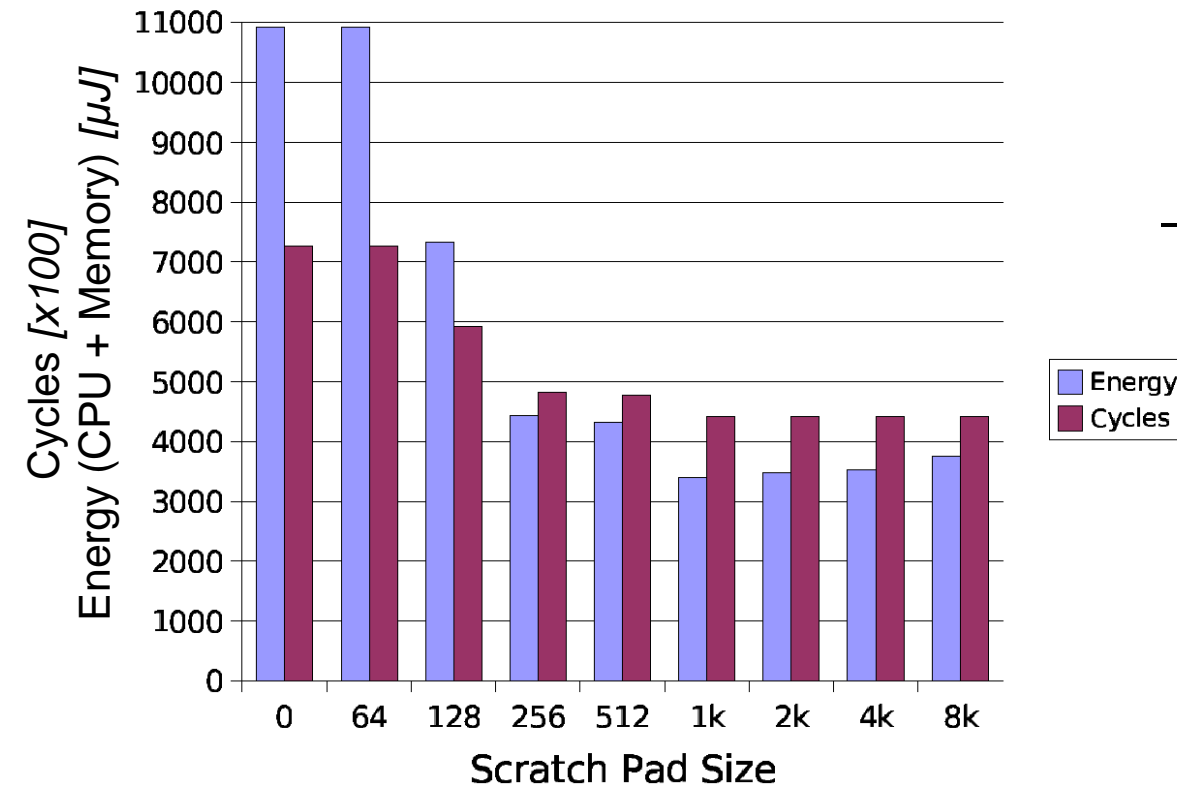
- Constraint: The SPM's capacity must not be exceeded

$$n_1 : \sum_{i=1}^N S_i * x_i \leq S$$

[S. Steinke. Untersuchung des Energieeinsparungspotenzials in eingebetteten Systemen durch energieoptimierende Compilertechnik. Dortmund 2002]

Fix SPM Allocation: Functions & Global Variables (7)

Results (*MultiSort Benchmark*)



- 64b SPM too small to hold global variables/functions.
- Steady energy and run-time reductions until 1kB SPM due to allocation of more MOs to SPM.
- From 2kB onwards minor degradations w.r.t. energy consumption since no more MOs can be allocated (all MOs already reside on SPM) but the energy consumption of larger SPMs increases for technological reasons.

Fix SPM Allocation: Functions, Basic Blocks & Global Variables (1)

Motivation

- Allocation of entire functions possibly disadvantageous:
 - 👉 Entire functions contain lots of code and occupy large space in the SPM
 - 👉 Parts of a function's code (e.g., code outside loops) are executed rarely. They thus produce only very small energy savings but, however, occupy costly SPM capacity.
- 👉 *(Scarce resource) SPM capacity exploited only in a sub-optimal way.*

Goal

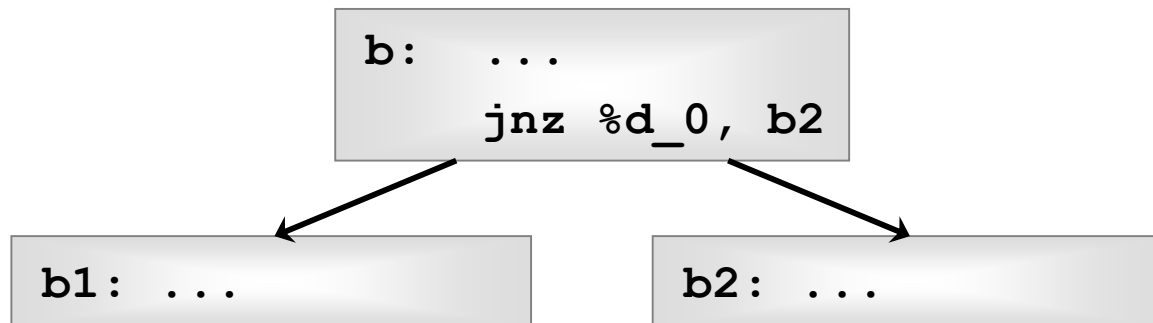
- Allocation of the code of complete LIR functions, of individual *basic blocks* and of global variables onto the SPM.

Fix SPM Allocation: Functions, Basic Blocks & Global Variables (2)

Problem when Moving Single Basic Blocks

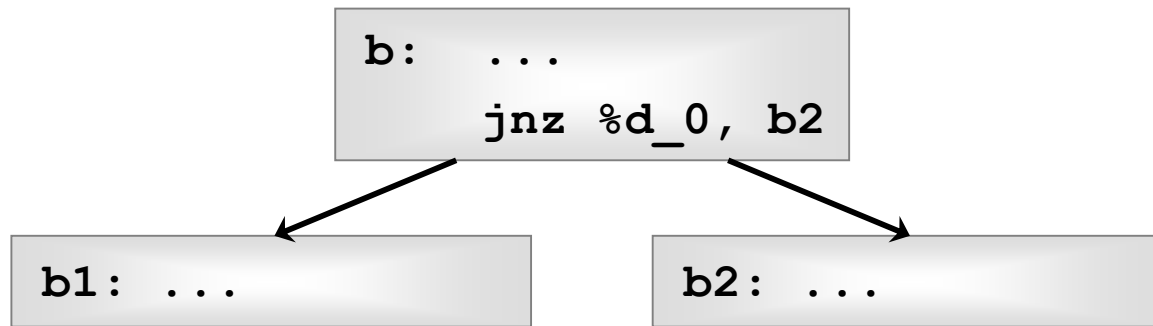
Remember: A basic block b may contain a branch instruction only as very last instruction

- If the branch at the end of b is conditional, b has two successors $b1$ and $b2$ in the CFG that are executed if the conditional branch is either taken or not taken:



Fix SPM Allocation: Functions, Basic Blocks & Global Variables (3)

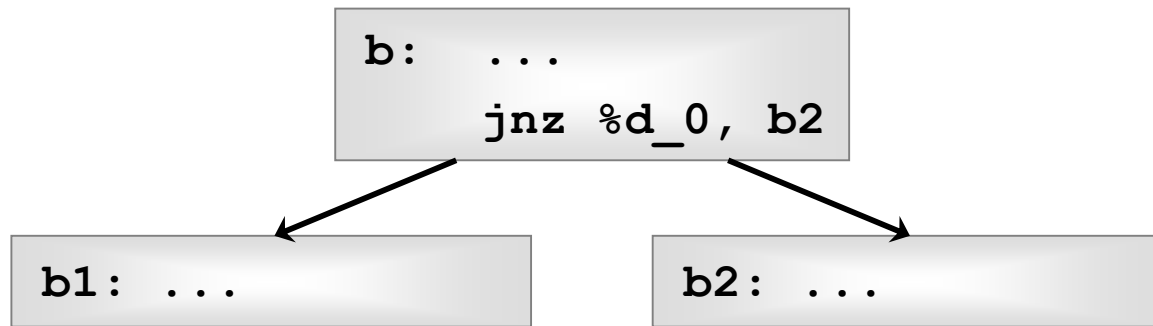
Problem when Moving Single Basic Blocks



- `b1` is reached from `b` implicitly if the conditional branch is not taken, since
 - 👍 the program counter is incremented after the non-taken branch and thus points to the next following instruction, and
 - 👍 the code of `b1` directly follows that of `b` in memory.

Fix SPM Allocation: Functions, Basic Blocks & Global Variables (4)

Problem when Moving Single Basic Blocks



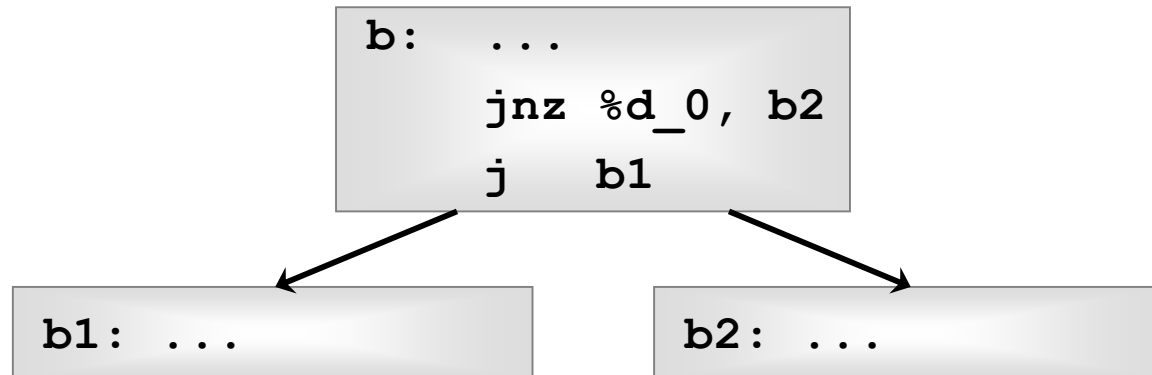
What if b resides in SPM but $b1$ not (or vice versa)?

- If the branch is not taken, the next instruction following b in the SPM will be executed.
- ☞ *Since $b1$ no longer follows b in the memory, incorrect code will be executed!*

Fix SPM Allocation: Functions, Basic Blocks & Global Variables (5)

A Naive Solution

- Extend all basic blocks b featuring such an implicit CFG edge by an unconditional branch to $b1$:



Disadvantage

- Unconditional branch extremely inefficient (code size, run-time and energy consumption) if both b and $b1$ still reside in the same memory.

Fix SPM Allocation: Functions, Basic Blocks & Global Variables (6)

A Better Approach

- Extension of a basic block b featuring such an implicit CFG edge by an unconditional branch only if b and b_1 are actually allocated to different memories

Advantage

- Unconditional branches are generated additionally only there where it is really necessary

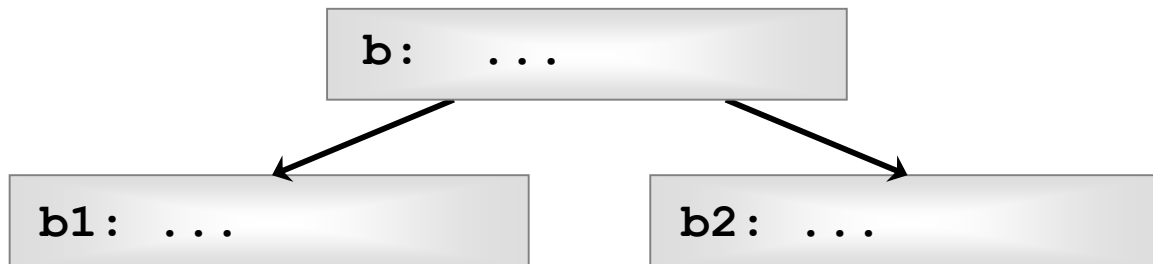
Problem

- Code size S_b of b now depends on the decision variables x_b and x_{b_1} that model the memory allocation of b in the ILP.
☞ *How to model a non-constant/variable parameter S_b in the ILP?*

Fix SPM Allocation: Functions, Basic Blocks & Global Variables (7)

Multi-Basic Blocks

- Sets of basic blocks that are connected in the CFG.
- Let G be the CFG of a function f , G' be a connected sub-graph of G . The set of all basic blocks of G' is a multi-basic block.



- $\{b, b1\}$, $\{b, b2\}$ and $\{b, b1, b2\}$ are multi-basic blocks.
- $\{b1, b2\}$ is no multi-basic block: Its associated G' is not connected.

Fix SPM Allocation: Functions, Basic Blocks & Global Variables (8)

(Multi-) Basic Blocks in the ILP Formulation

- ILP for SPM allocation optimizes memory objects from the sets of all functions F , of all single basic blocks B , of all multi-basic blocks MB and of all global variables V .
- MB is constructed by generating all possible connected sub-graphs G' of the CFG.

Definitions

- $MO = \{MO_1, \dots, MO_N\}$ Set of all memory objects to be considered
 $= F \cup B \cup MB \cup V$ for SPM allocation
- Meaning of all other terms ($S, S_i, \Delta E_{i,\text{total}}, \dots$) as before

Fix SPM Allocation: Functions, Basic Blocks & Global Variables (9)

Determination of Parameter Values

- S_i : For $MO_i \in F$ or $MO_i \in V$: as before;
 - For $MO_i \in B$: Size of all instructions of the basic block, plus the size of an unconditional branch if MO_i has an implicit successor in the CFG;
 - For $MO_i \in MB$: Size of all instructions of all basic blocks included in MO_i , plus the size of k unconditional branches if MO_i has k implicit successors outside MO_i in the CFG.
- $\Delta E_{i,\text{single}}$: as before, but now just analogously to S_i under consideration of the novel unconditional branches
- A_i : as before per profiling, now just also for all $MO_i \in B \cup MB$

Fix SPM Allocation: Functions, Basic Blocks & Global Variables (10)

ILP Formulation

- Objective function: Maximize energy savings for the whole program

$$z : \sum_{i=1}^N \Delta E_{i,\text{total}} * x_i \rightsquigarrow \max .$$

- Constraint 1: The SPM's capacity must not be exceeded

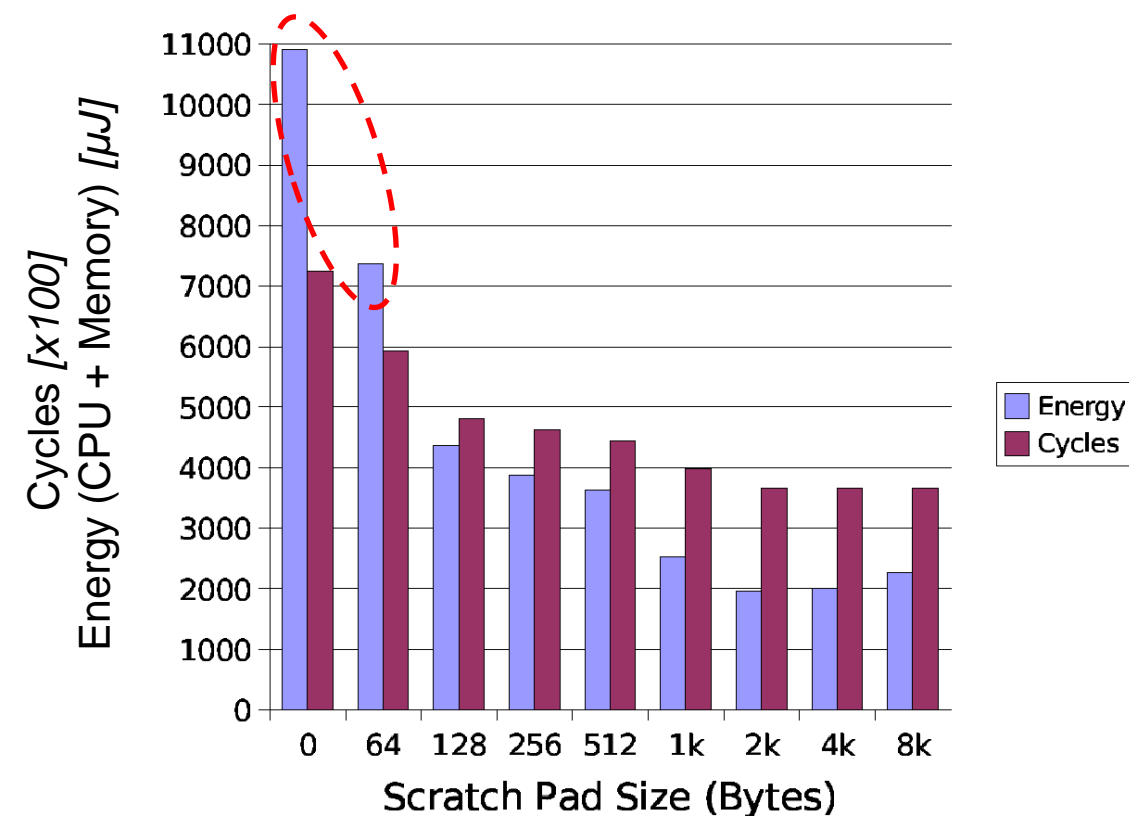
$$n_1 : \sum_{i=1}^N S_i * x_i \leq S$$

- Additional constraint per $MO_b \in B$: MO_b may only be allocated to the SPM by at most one decision variable within the ILP (for b itself, for b 's function or for all multi-basic blocks that contain b)

$$\forall MO_b \in B : x_b + x_{f(b)} + \sum_{MO_j \in MB: MO_b \in MO_j} x_j \leq 1$$

Fix SPM Allocation: Functions, Basic Blocks & Global Variables (11)

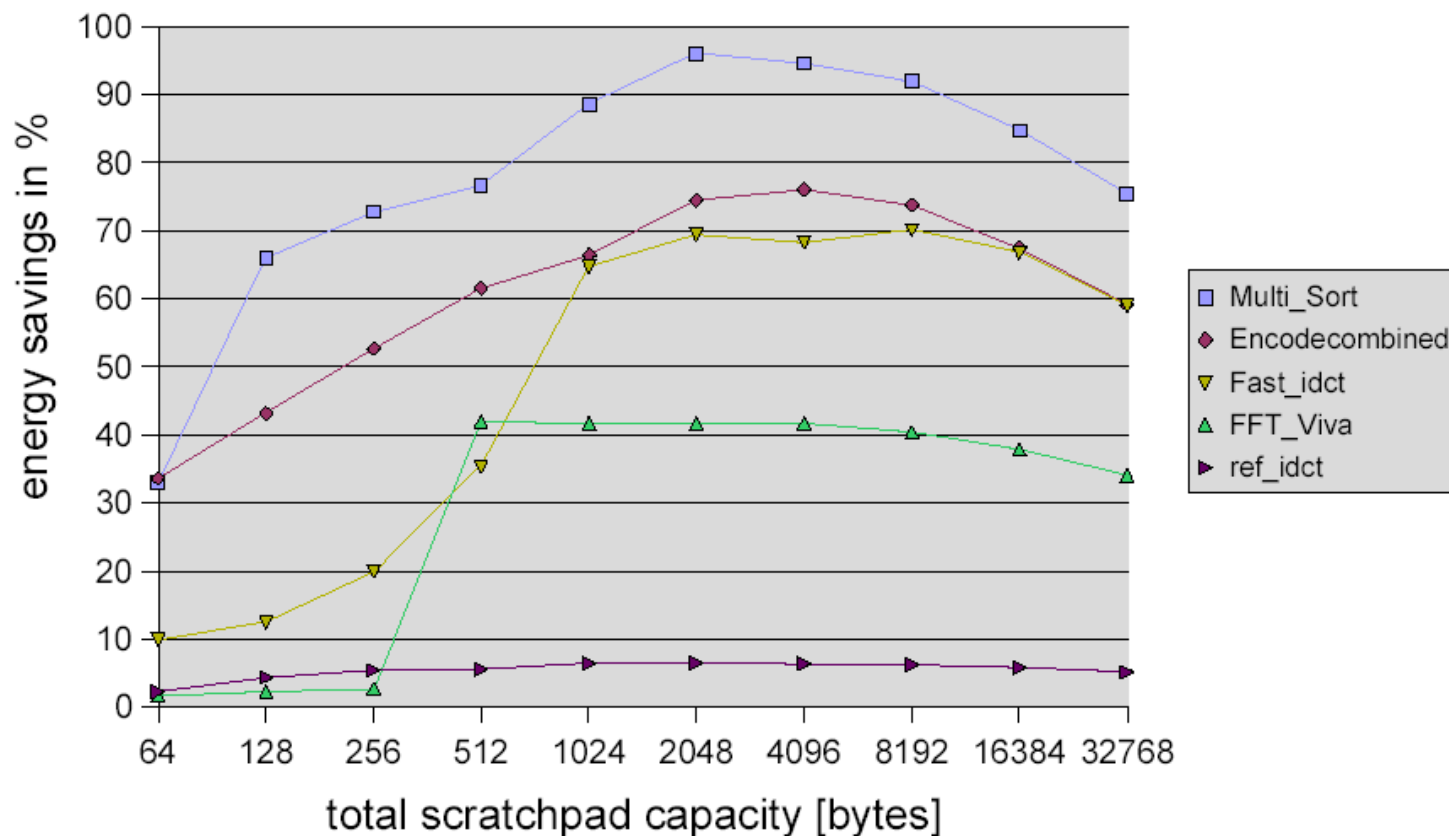
Results (*MultiSort Benchmark*)



- 64b SPM are now successfully exploited for the allocation of pieces of code.
- *Caution: The diagram here also features the allocation of the run-time stack onto the SPM. For this reason, this diagram cannot directly be compared with that from slide 63!*

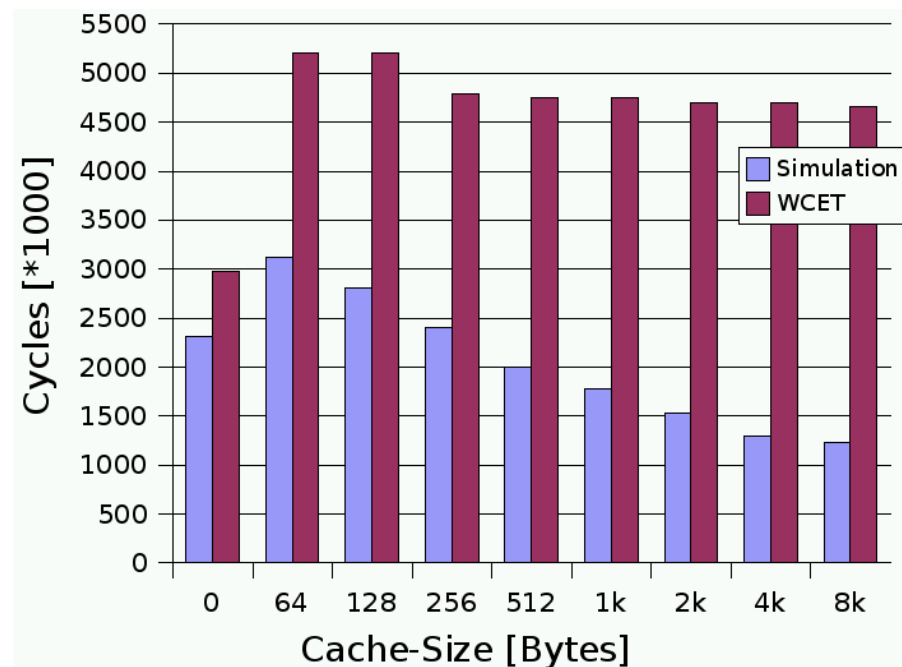
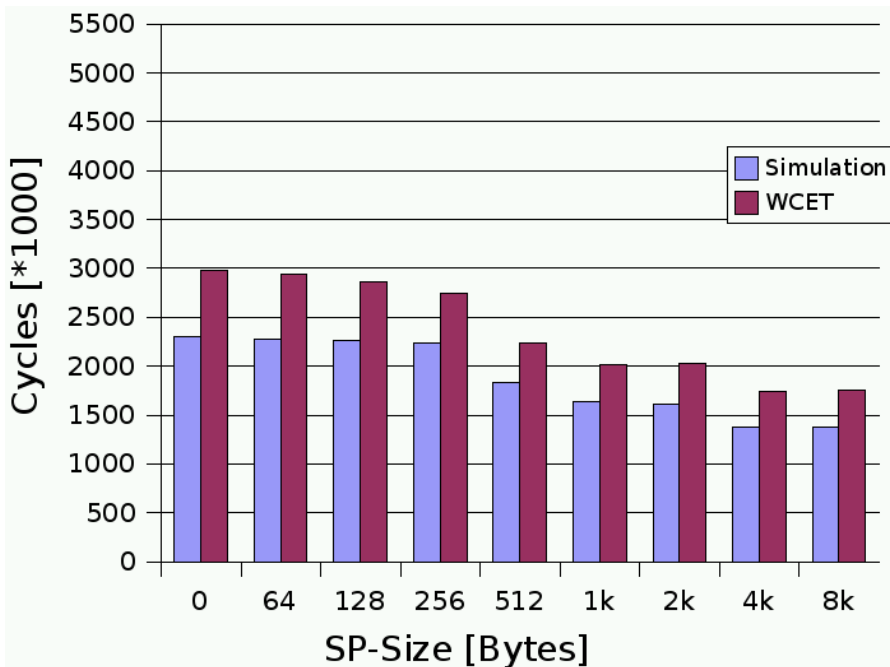
Fix SPM Allocation: Functions, Basic Blocks & Global Variables (12)

Detailed Results only for Memory Subsystem



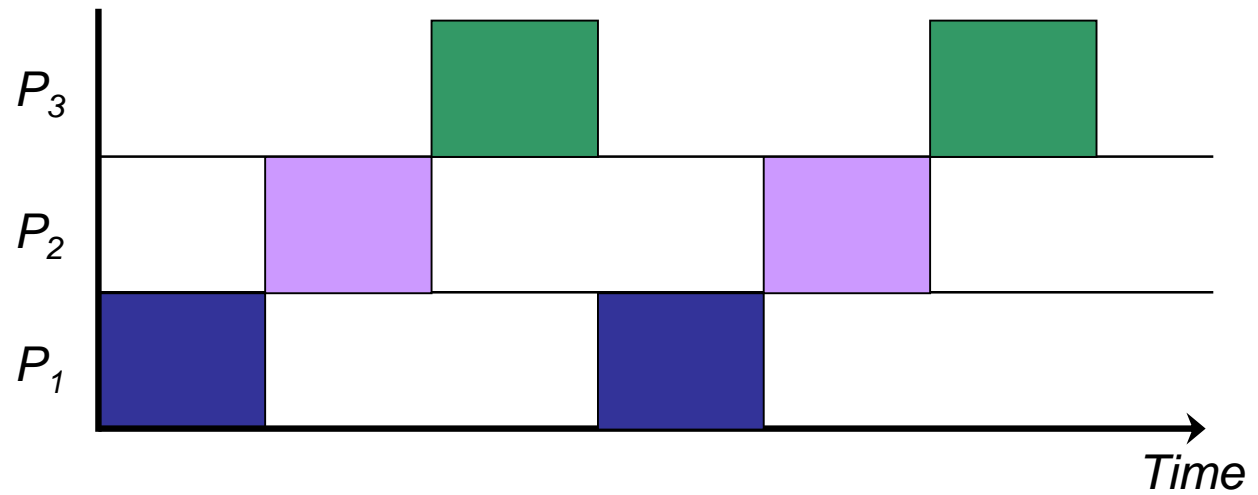
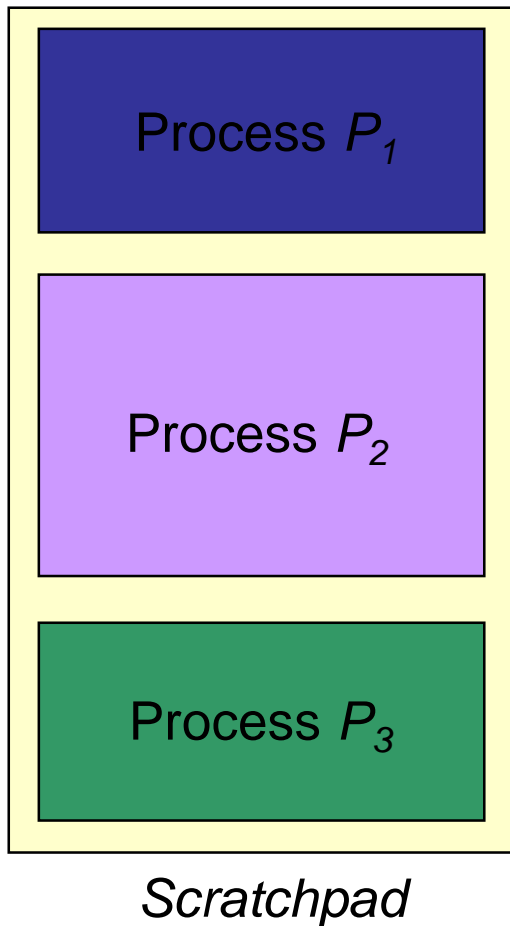
Fix SPM Allocation: Functions, Basic Blocks & Global Variables (13)

Comparison of ACET and $WCET_{EST}$ for Scratchpads and Caches



- SPMs are, in contrast to caches, perfectly predictable: $WCET_{EST}$ scales nicely with ACETs
- Only for larger memory sizes (beyond 2kB), caches outperform SPMs in terms of ACET

Multi-Process SPM Allocation: Partitioned SPM (1)



- Available SPM capacity is partitioned into disjoint areas during compile-time
- Each process P_i obtains a dedicated SPM partition for functions, (multi-) BBs & global variables of P_i
- *Expectation:* Good results for large SPMs

Multi-Process SPM Allocation: Partitioned SPM (2)

Energy Arrays of Single Processes

- Let P_1, \dots, P_N be the processes of a multi-process application
- S denotes the size of the available SPM capacity in bytes,
 $S' < S$ is a user-provided parameter that cuts the SPM into slices of S' bytes size each
- For each process P_i , we will determine an energy array $f'_i[x]$. $f'_i[x]$ denotes how much energy P_i consumes if P_i has x bytes of SPM at its disposal.
- $f'_i[x]$ is pre-computed for all sizes $x = S', 2S', 3S', \dots$ that are multiples of S' . This is achieved by repeated solution of the ILP for fix SPM allocation of functions, multi-BBs and global variables for each value of x above.

Multi-Process SPM Allocation: Partitioned SPM (3)

Energy Arrays for Multi-Process System and Partitioned SPM

- For a complete multi-process system consisting of processes P_1, \dots, P_N , its energy array $e'_N[x]$ has to be determined. $e'_N[x]$ denotes how much energy the entire multi-process system consumes if it has x bytes of SPM at its disposal.
- $e'_N[x] = \min\{ f'_1[x_1] + \dots + f'_N[x_N] \mid x_1 + \dots + x_N \leq x \}$
for all those values of x, x_1, \dots, x_N for which the individual f'_i are defined
- For a fixed available SPM size S and a multi-process system, an SPM allocation yielding the minimal $e'_N[S]$ has to be determined.

Multi-Process SPM Allocation: Partitioned SPM (4)

The `binmin` Function

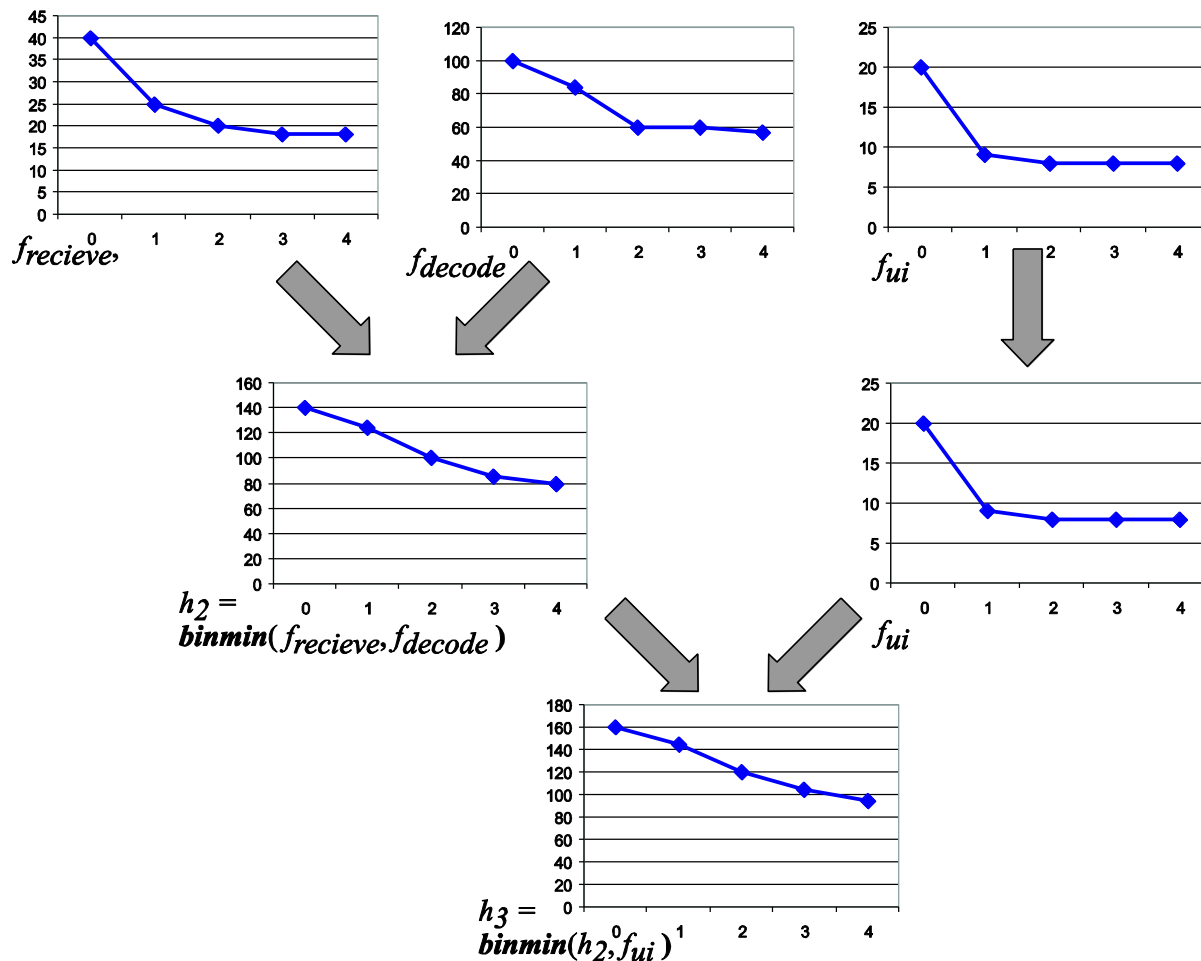
- Example: A multi-process system consists of 3 processes P_1 , P_2 and P_3 .
- $$e'_3[x] = \min\{ f'_1[x_1] + f'_2[x_2] + f'_3[x_3] \mid x_1 + x_2 + x_3 \leq x \}$$

$$= \text{binmin}(\text{binmin}(f'_1, f'_2), f'_3)$$
- `binmin` is a binary function that combines *two* energy arrays g and h and again computes an energy array:

$$\text{binmin}(g, h)[x] = \min\{ g[x_1] + h[x_2] \mid x_1 + x_2 \leq x \}$$
- Due to its associativity, computing the minimum over an N -fold sum in $e'_N[x]$ can be reduced to the binary `min` in `binmin`.

Multi-Process SPM Allocation: Partitioned SPM (5)

Example:
System with 3
processes (*receive*,
decode, *ui*)



Multi-Process SPM Allocation: Partitioned SPM (6)

Algorithm to Compute $\text{binmin}(g, h)$

- for ($x = 0; x \leq S; x += S'$)
 - int $min = \infty$;
 - for ($tmp = 0; tmp \leq x; tmp += S'$)
 - if ($g[tmp] + h[x - tmp] < min$)
 $min = g[tmp] + h[x - tmp]$;
 - $b[x] = min$;
- return b ;

Multi-Process SPM Allocation: Partitioned SPM (7)

Algorithm to Compute e'_N

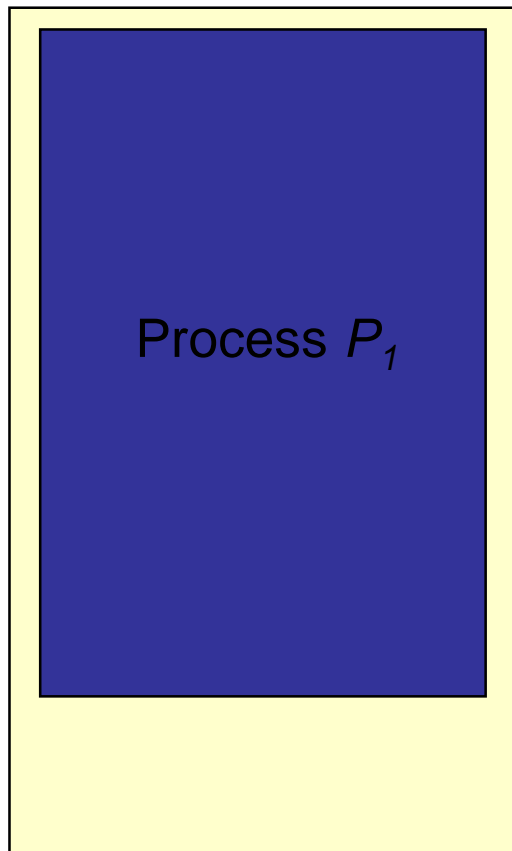
PartitionedSPM(f'_1, \dots, f'_N)

- if ($N > 2$)
 - $e'_{N-1} = \mathbf{PartitionedSPM}(f'_1, \dots, f'_{N-1});$
 - $e'_N = \mathbf{binmin}(e'_{N-1}, f'_N);$
- else
 - $e'_N = \mathbf{binmin}(f'_1, f'_2);$
- return e'_N ;

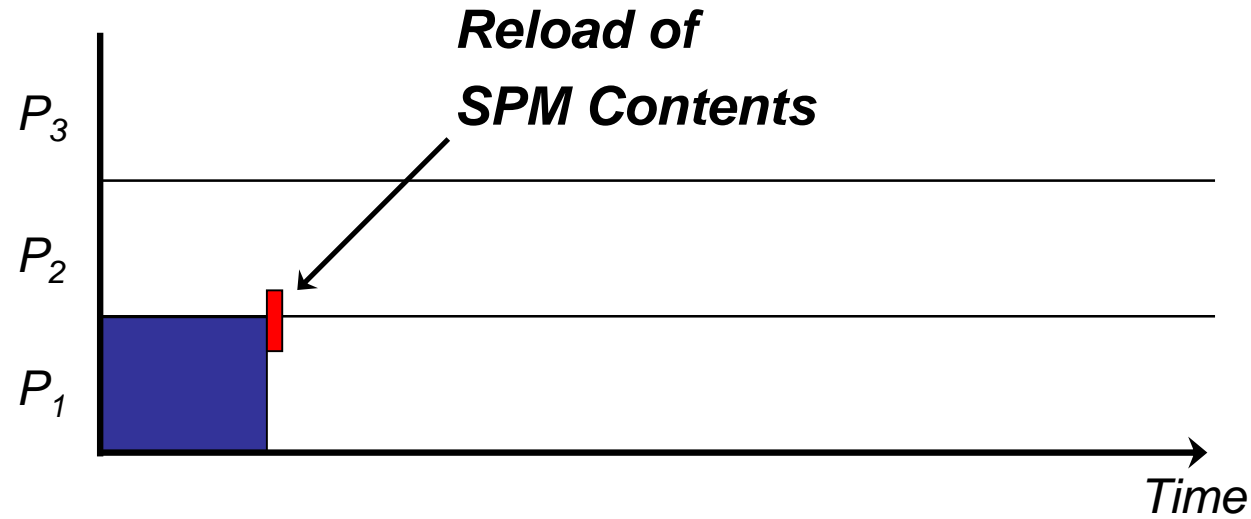
- **binmin** can easily be adapted such that it not only returns array $b[]$, but also that value of tmp for which $b[x]$ is minimal.

☞ This way, the algorithm not only computes e'_N but also the partition sizes x_i for all processes P_i .

Multi-Process SPM Allocation: Exclusive SPM (1)

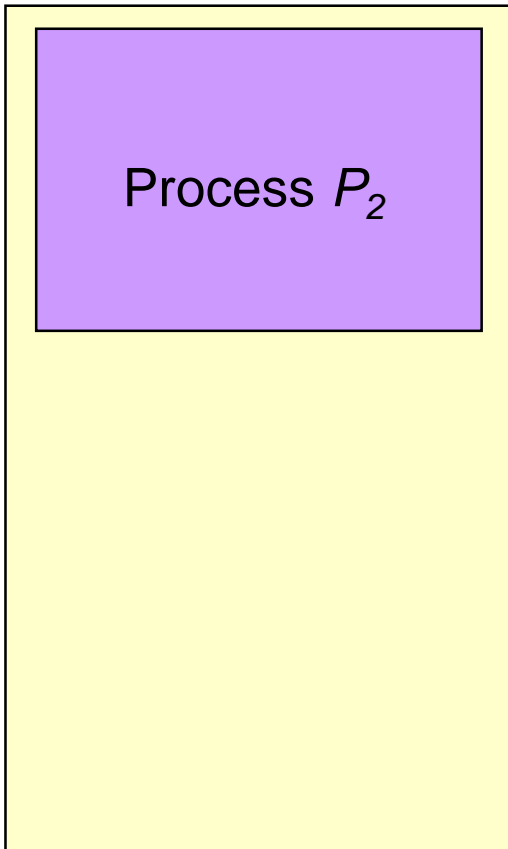


Scratchpad

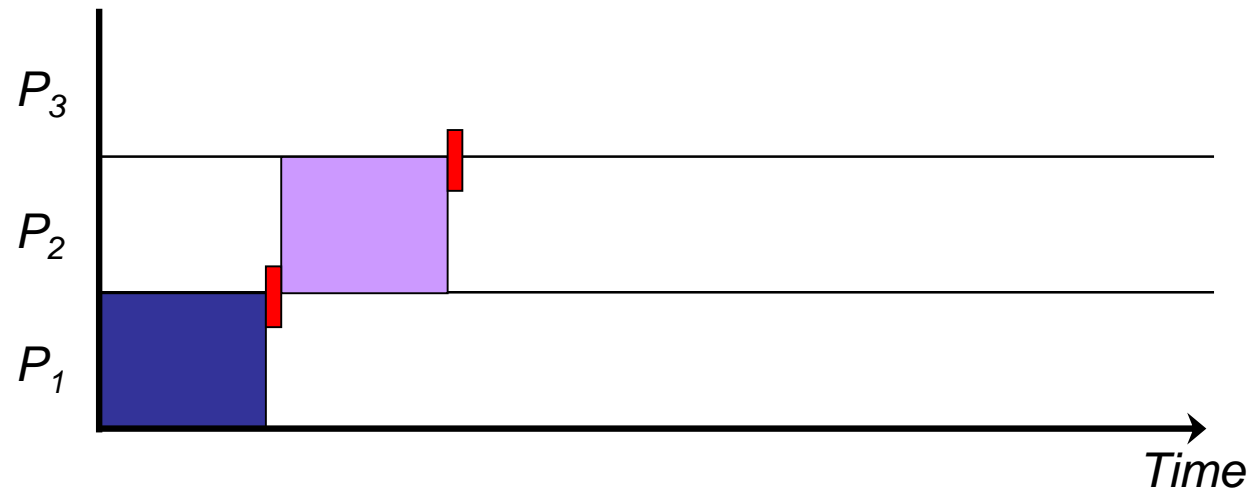


- Each process P_i can freely and exclusively use the complete SPM for its functions, (multi-) BBs & global variables
- During context switches, the SPM content must be stored and reloaded

Multi-Process SPM Allocation: Exclusive SPM (1)

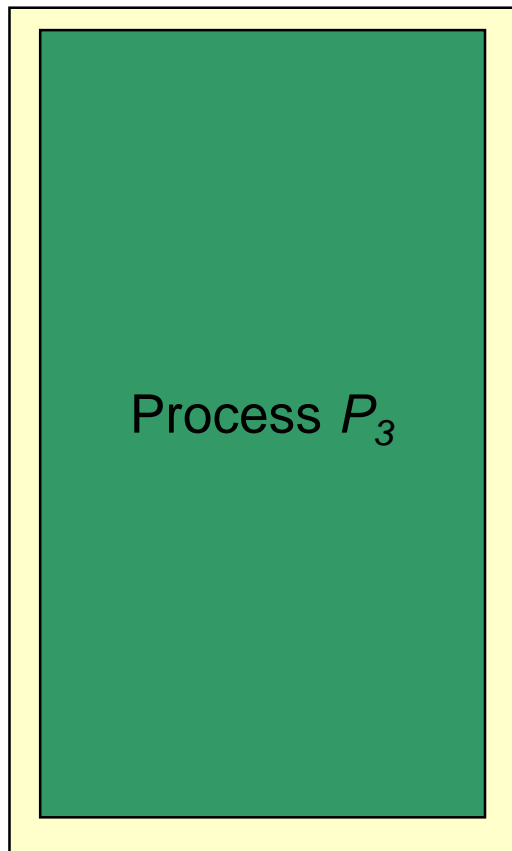


Scratchpad

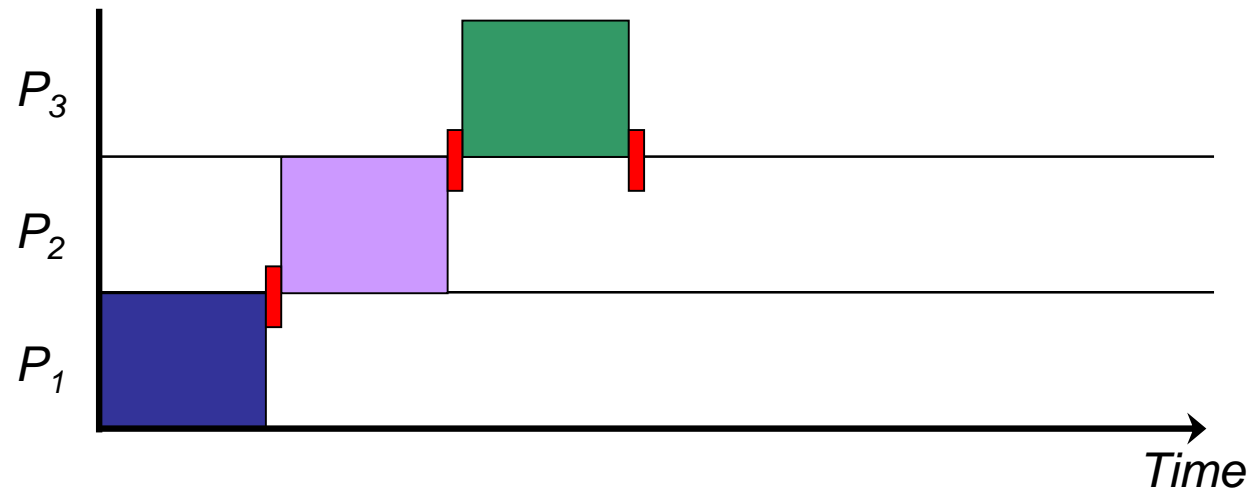


- Each process P_i can freely and exclusively use the complete SPM for its functions, (multi-) BBs & global variables
- During context switches, the SPM content must be stored and reloaded

Multi-Process SPM Allocation: Exclusive SPM (1)

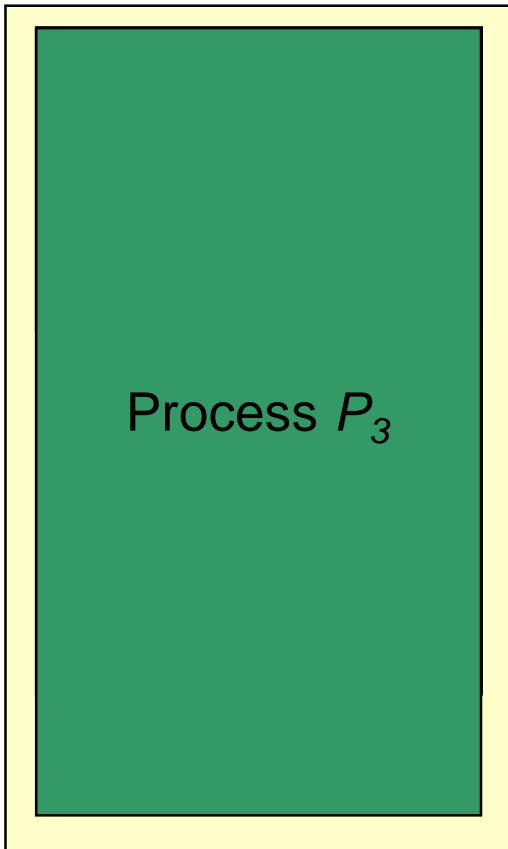


Scratchpad

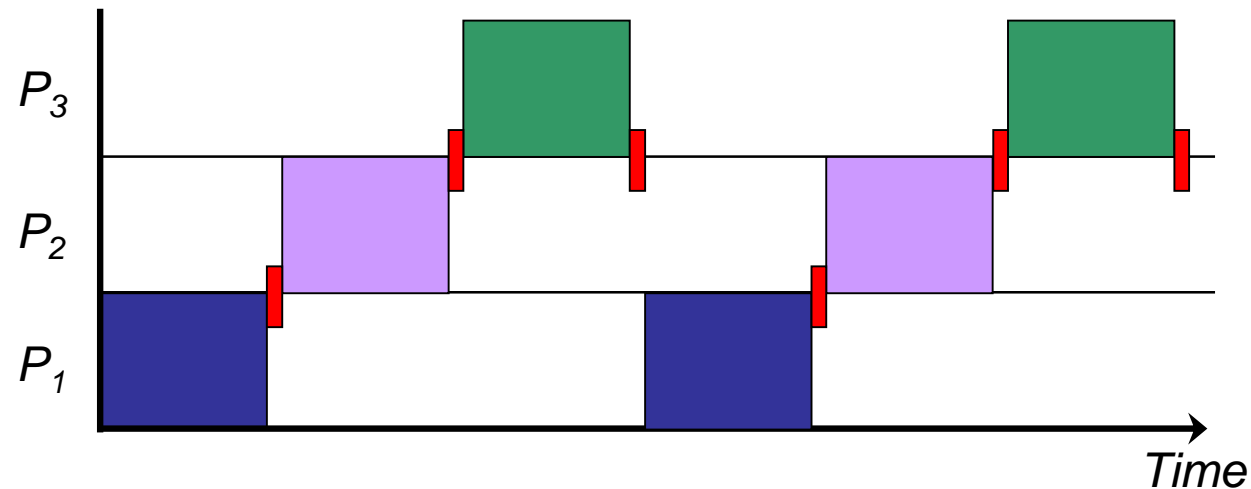


- Each process P_i can freely and exclusively use the complete SPM for its functions, (multi-) BBs & global variables
- During context switches, the SPM content must be stored and reloaded

Multi-Process SPM Allocation: Exclusive SPM (1)



Scratchpad



- Each process P_i can freely and exclusively use the complete SPM for its functions, (multi-) BBs & global variables
- During context switches, the SPM content must be stored and reloaded
- *Expectation:* Good results for small SPMs

Multi-Process SPM Allocation: Exclusive SPM (2)

Energy Arrays of Single Processes

- For each process P_i , s_i denotes its scheduling frequency. s_i is pre-computed using profiling.
- For each process P_i , we will determine an energy array $f_i''[x]$. $f_i''[x]$ denotes how much energy P_i consumes if P_i has x bytes of SPM at its disposal that need to be stored and reloaded during context switches.
- $f_i''[x]$ is pre-computed analogously for all sizes $x = S', 2S', 3S', \dots$:

$$f_i''[x] = f_i'[x] + s_i * (CE_{\text{SPM} \rightarrow \text{MM}}(x) + CE_{\text{MM} \rightarrow \text{SPM}}(x))$$
- $f_i''[x]$ = Energy consumed during the execution of P_i using x Bytes SPM + additional energy used to copy x bytes from SPM to main memory and back (*copy energy CE*), multiplied by the frequency of context switches for P_i .

Multi-Process SPM Allocation: Exclusive SPM (3)

Energy Arrays for Multi-Process System and Exclusive SPM

- For a complete multi-process system consisting of processes P_1, \dots, P_N , its energy array $e''_N[x]$ has to be determined. $e''_N[x]$ denotes how much energy the entire multi-process system consumes if it has x bytes of SPM at its disposal that (partially) need to be reloaded during context switches.

$$e''_N[x] = \sum_{P_i} \min\{ f''_i[x_j] \mid x_j \leq x \}$$

for all those values of x and x_j for which the individual f''_i are defined

- For a fixed available SPM size S and a multi-process system, an SPM allocation yielding the minimal $e''_N[S]$ has to be determined.

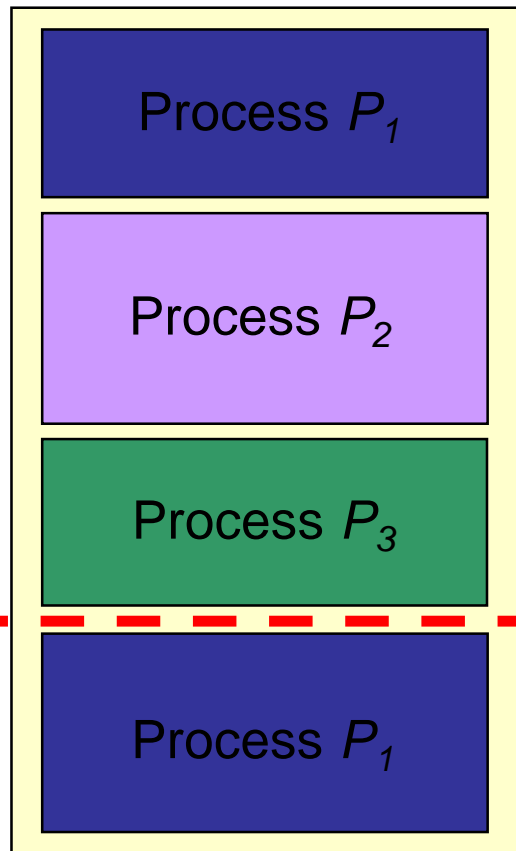
Multi-Process SPM Allocation: Exclusive SPM (4)

Algorithm to Compute e''_N

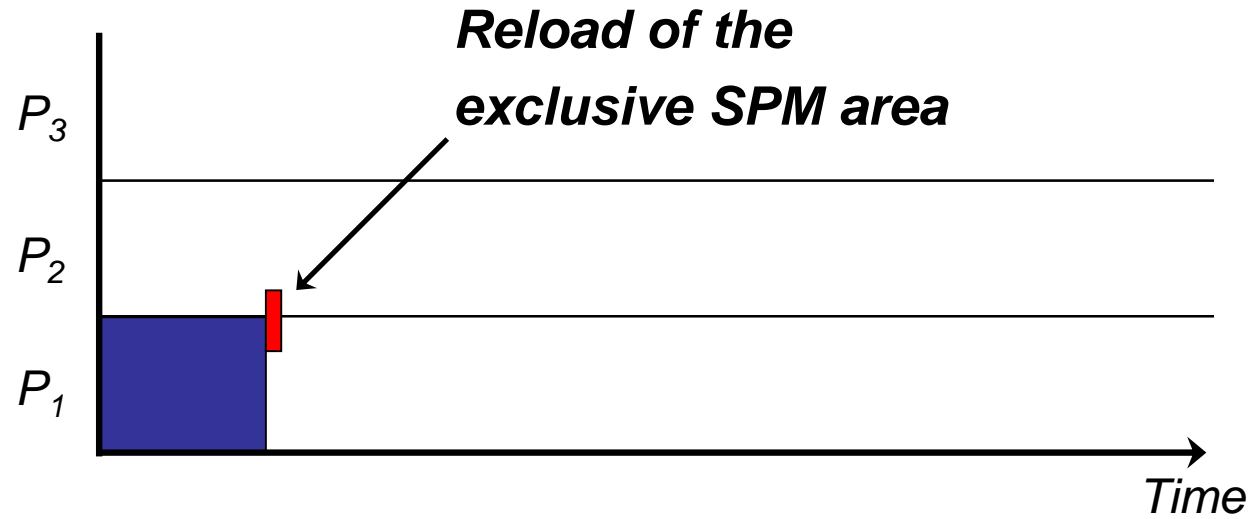
ExclusiveSPM(f''_1, \dots, f''_N)

- for (*<all processes P_i >*)
 - $prev_min[i] = \infty$;
- for ($x = 0$; $x \leq S$; $x += S'$)
 - $e''_N[x] = 0$;
 - for (*<all processes P_i >*)
 - $min[i] = (f''_i[x] < prev_min[i]) ? f''_i[x] : prev_min[i]$;
 - $e''_N[x] += min[i]$;
 - $prev_min[i] = min[i]$;
- return e''_N ;

Multi-Process SPM Allocation: Hybrid SPM (1)

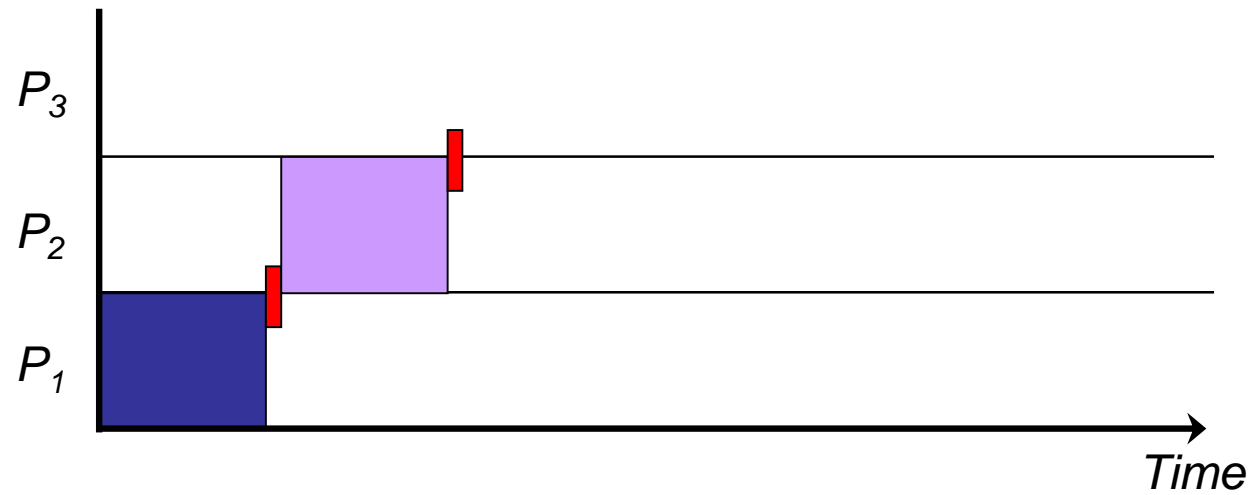
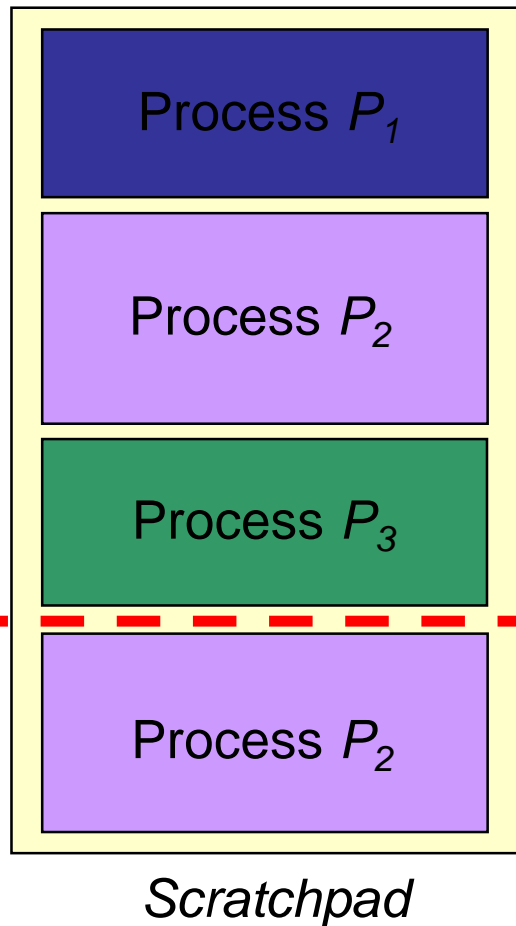


Scratchpad



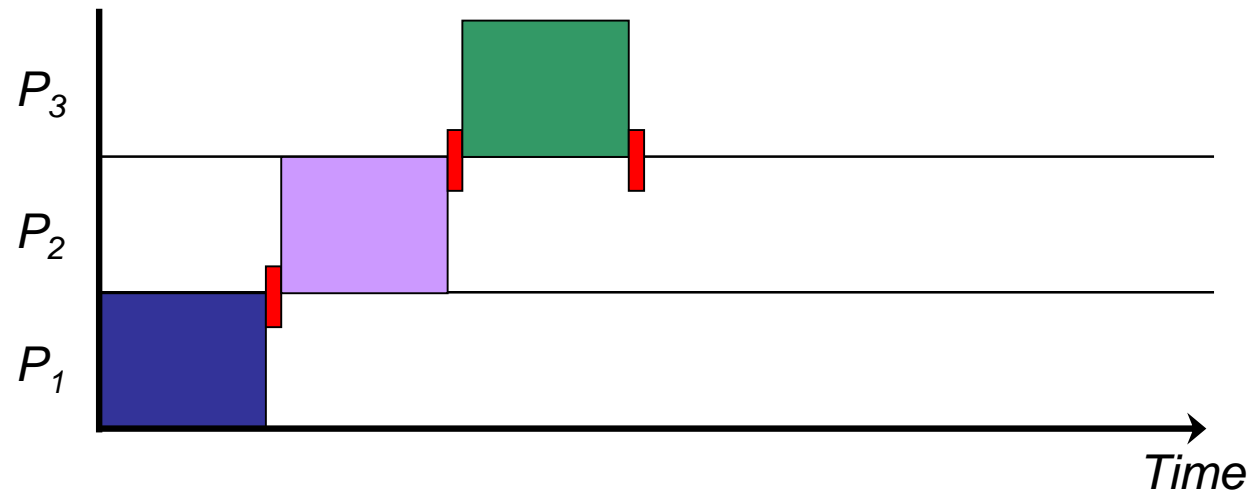
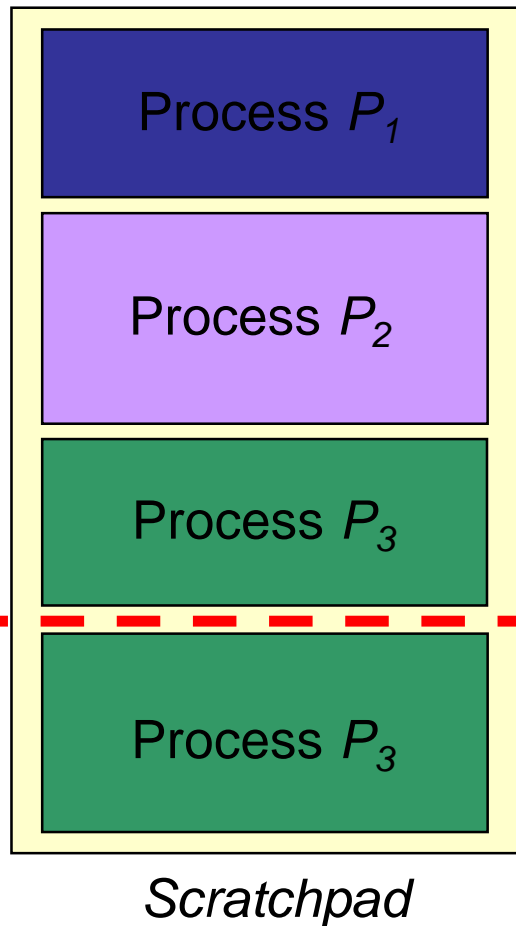
- SPM is grouped into a partitioned and an exclusive area.
- Each process P_i can make use of the exclusive area and of a dedicated part of the partitioned area.

Multi-Process SPM Allocation: Hybrid SPM (1)



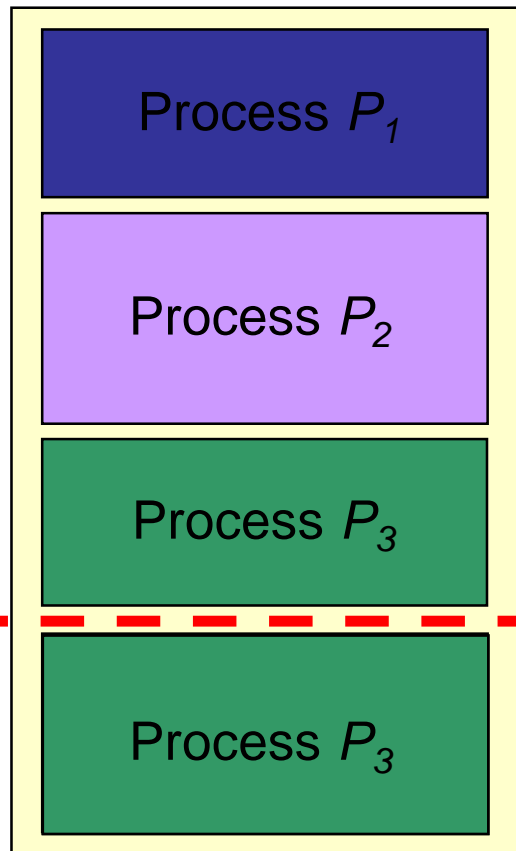
- SPM is grouped into a partitioned and an exclusive area.
- Each process P_i can make use of the exclusive area and of a dedicated part of the partitioned area.

Multi-Process SPM Allocation: Hybrid SPM (1)

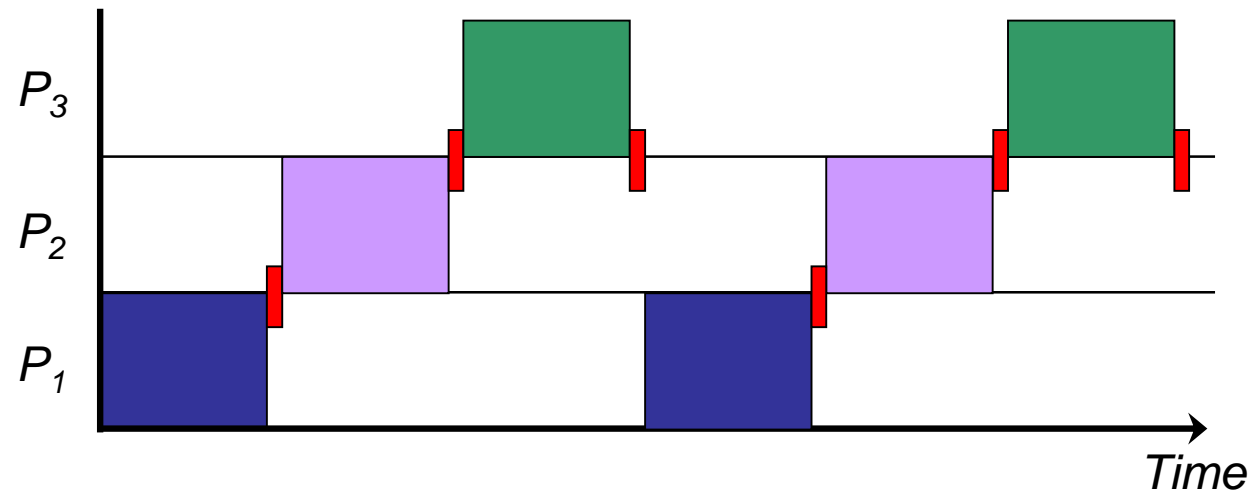


- SPM is grouped into a partitioned and an exclusive area.
- Each process P_i can make use of the exclusive area and of a dedicated part of the partitioned area.

Multi-Process SPM Allocation: Hybrid SPM (1)



Scratchpad



- SPM is grouped into a partitioned and an exclusive area.
- Each process P_i can make use of the exclusive area and of a dedicated part of the partitioned area.
- *Expectation*: Good results for all kinds of SPMs

Multi-Process SPM Allocation: Hybrid SPM (2)

Energy Arrays of Single Processes

- For each process P_i , we will determine an energy array $f_i'''[x, y]$. $f_i'''[x, y]$ denotes how much energy P_i consumes if P_i has x bytes of partitioned and y bytes of exclusive SPM to be reloaded during context switches at its disposal.
- $f_i'''[x, y]$ is again pre-computed for all sizes of x and $y = S', 2S', 3S', \dots$
This is achieved by solving a special variant of the ILP for fix SPM allocation (not shown here) that supports two different SPMs of sizes x and y .

Multi-Process SPM Allocation: Hybrid SPM (3)

Energy Arrays for Multi-Process System and Hybrid SPM

- For a complete multi-process system consisting of processes P_1, \dots, P_N , its energy array $e_N'''[x, y]$ has to be determined. $e_N'''[x, y]$ denotes how much energy the entire multi-process system consumes if it has x bytes partitioned and y bytes exclusive SPM at its disposal.
- $$e_N'''[x, y] = \min \{ f_1'''[x_1, y_1] + \dots + f_N'''[x_N, y_N] \mid x_1 + \dots + x_N \leq x \text{ and } y_i \leq y \text{ for each } i = 1, \dots, N \}$$
 for all those values of $x_1, \dots, x_N, y_1, \dots, y_N$, for which the individual f_i''' are defined
- For a fixed available SPM size S and a multi-process system, an SPM allocation yielding the minimal $e_N'''[x, y]$ has to be determined such that $x + y \leq S$ holds.

Multi-Process SPM Allocation: Hybrid SPM (4)

The `hybrid_binmin` Function

- Example: A multi-process system consists of 3 processes P_1 , P_2 and P_3 .
- $e_3'''[x, y] = \min\{ f_1'''[x_1, y_1] + f_2'''[x_2, y_2] + f_3'''[x_3, y_3] \mid x_1 + x_2 + x_3 \leq x \wedge y_1 \leq y \wedge y_2 \leq y \wedge y_3 \leq y \}$
 $= \text{hybrid_binmin}(\text{hybrid_binmin}(f_1''', f_2'''), f_3''')$
- `hybrid_binmin` is a binary function that combines *two* energy arrays g and h and again computes an energy array:

$$\text{hybrid_binmin}(g, h)[x, y] = \min\{ g[x_1, y_1] + h[x_2, y_2] \mid x_1 + x_2 \leq x \wedge y_1 \leq y \wedge y_2 \leq y \}$$
- Due to its associativity, computing the minimum over an N -fold sum in $e_N'''[x, y]$ can be reduced to the binary `min` in `hybrid_binmin`.

Multi-Process SPM Allocation: Hybrid SPM (5)

Algorithm to Compute `hybrid_binmin(g, h)`

- for ($y = 0; y \leq S; y += S'$)
 - int $min = \infty$;
 - for ($x = 0; x \leq S - y; x += S'$)
 - for ($tmp = 0; tmp \leq x; tmp += S'$)
 - if ($g[tmp, y] + h[x - tmp, y] < min$)
 $min = g[tmp, y] + h[x - tmp, y]$;
 - $b[x, y] = min$;
- return b ;

Multi-Process SPM Allocation: Hybrid SPM (6)

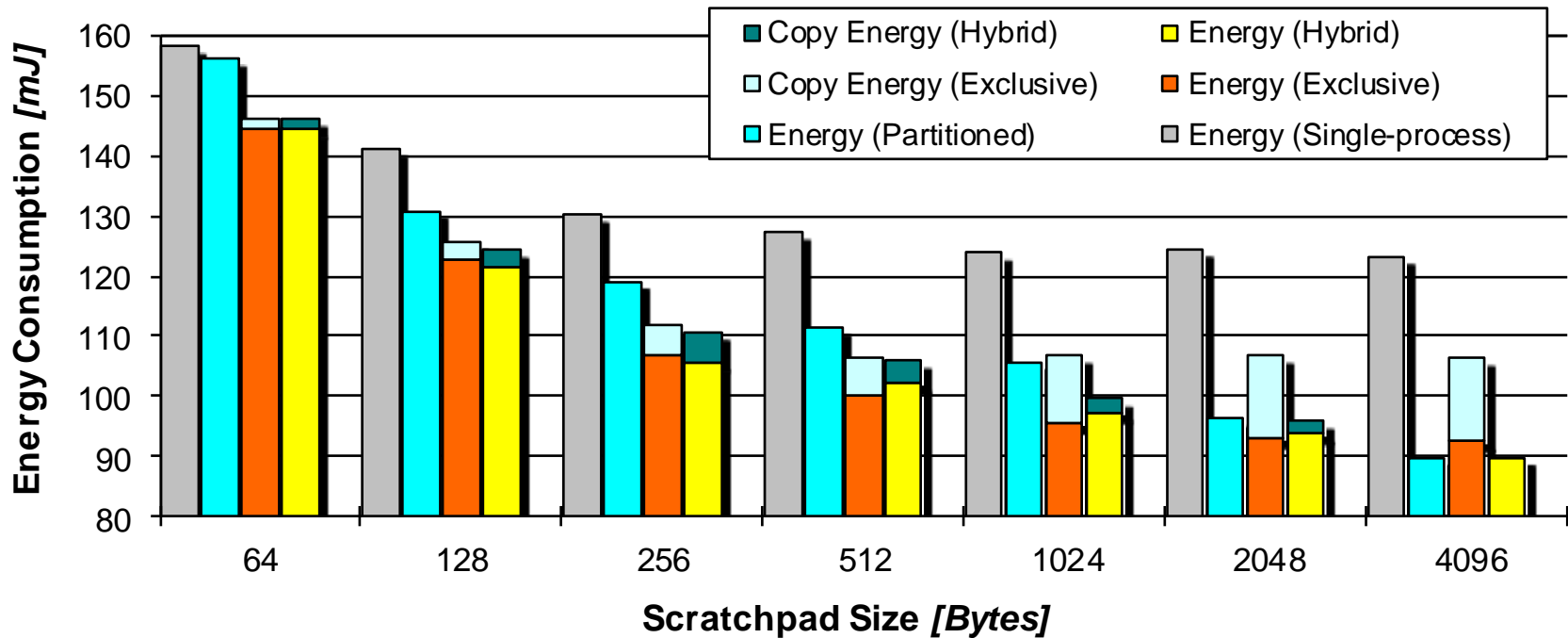
Algorithm to Compute e_N'''

HybridSPM(f_1''', \dots, f_N''')

- if ($N > 2$)
 - $e_{N-1}''' = \mathbf{HybridSPM}(f_1''', \dots, f_{N-1}''')$;
 - $e_N''' = \mathbf{hybrid_binmin}(e_{N-1}''', f_N''')$;
- else
 - $e_N''' = \mathbf{hybrid_binmin}(f_1''', f_2''')$;
- return e_N''' ;

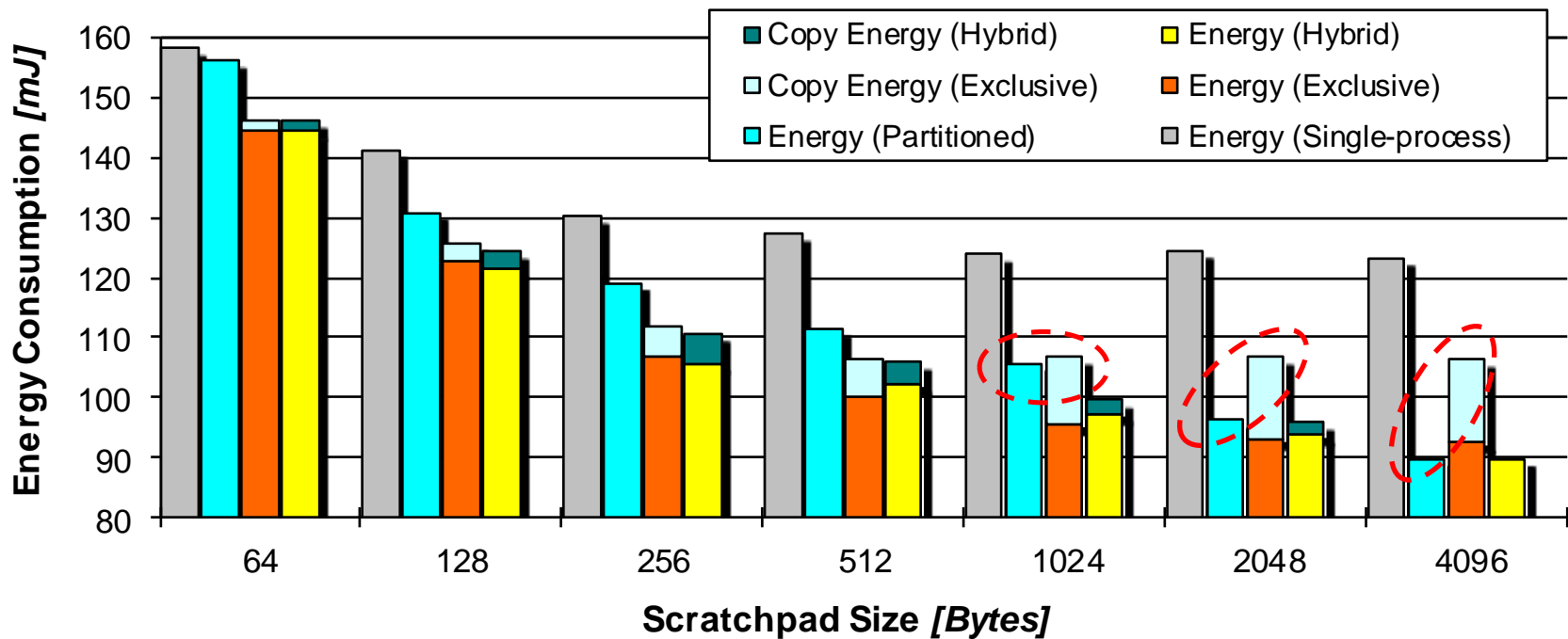
- Functionality in analogy to **PartitionedSPM** (👉 slide 83)

Multi-Process SPM Allocation: Results (1)



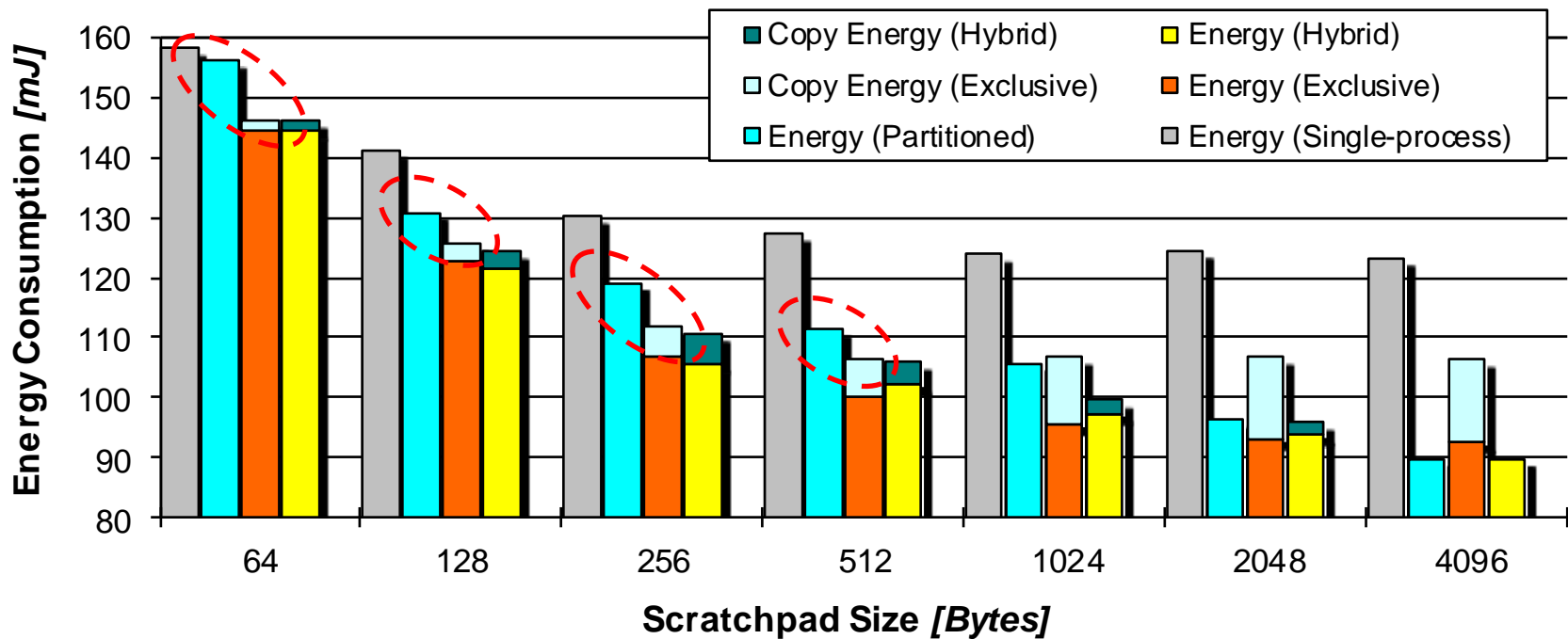
- *Here*: Media application (processes adpcm, g721, mpeg4, edge detection)
- *Energy (single-process)*: SPM is completely assigned to that single process using fix SPM allocation that reduces energy consumption most.

Multi-Process SPM Allocation: Results (2)



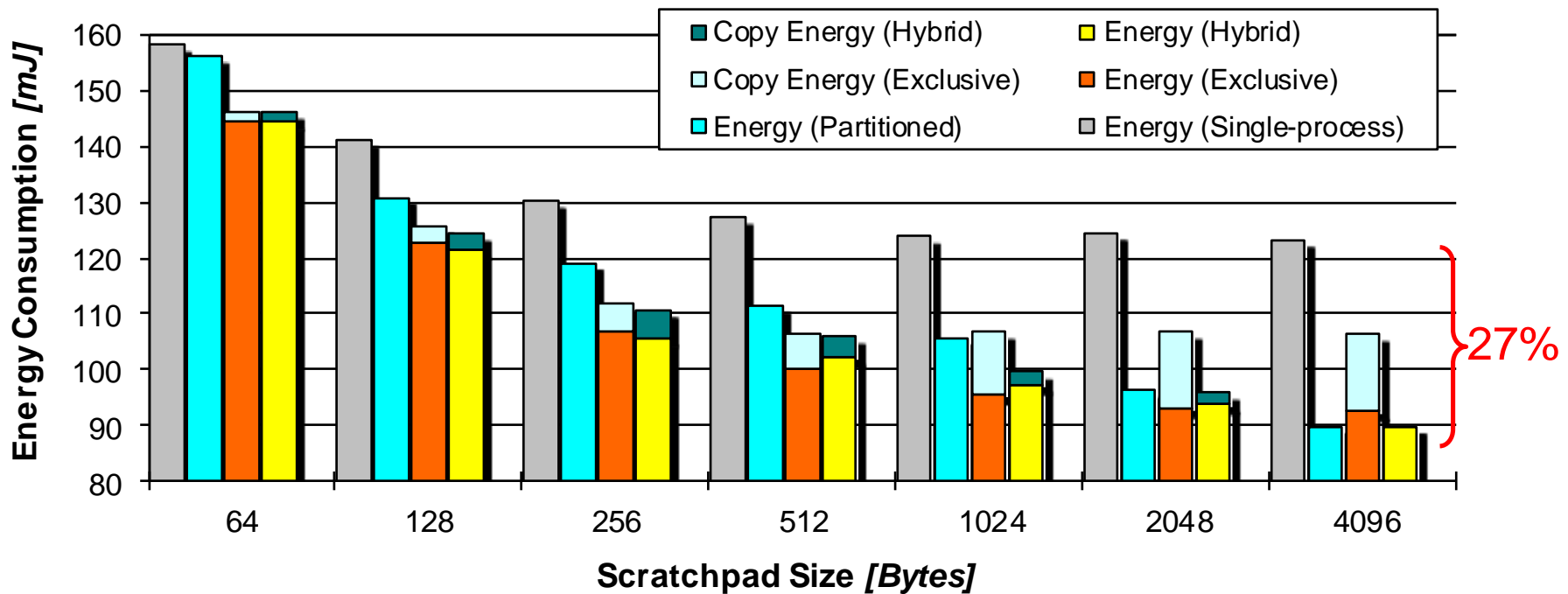
- *Partitioned SPM*: In fact beneficial for large memories
- Partitioning of the SPM outperforms exclusive SPM allocation from 1kB SPM size on

Multi-Process SPM Allocation: Results (3)



- *Exclusive SPM*: In fact beneficial for small memories
- Exclusive SPM allocation including reloading of SPM content outperforms partitioned SPMs up to 512 bytes SPM size

Multi-Process SPM Allocation: Results (4)



- *Hybrid SPM*: Is in fact the best allocation scheme for all SPM sizes
- At 4kB SPM: 27% energy savings compared to bar “single-process”
- Similar results also for “Video Phone” and “DSP” multi-process systems

References (1)

Code Generation for Network Processors

- J. Wagner. *Retargierbare Ausnutzung von Spezialoperationen für Eingebettete Systeme mit Hilfe bitgenauer Wertflussanalyse*. Dissertation, Dortmund 2006.


References (2)

Optimizations for Scratchpad Memories

- S. Steinke. *Untersuchung des Energieeinsparungspotenzials in eingebetteten Systemen durch energieoptimierende Compilertechnik*. Dissertation, Dortmund 2002.
- M. Verma, P. Marwedel. *Advanced Memory Optimization Techniques for Low-Power Embedded Processors*. Springer, 2007.
- M. Verma, K. Petzold, L. Wehmeyer et al. *Scratchpad Sharing Strategies for Multiprocess Embedded Systems: A first Approach*. 3rd IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia), Jersey City, September 2005.

Summary

Generation of Bit-Packet Operations for NPUs

- Conventional data flow analysis are not bit-true
- Bit-true data and value flow analysis via forward and backward simulation
- Detection of bit-Packets using  values of the BDVFA

Optimizations for Scratchpad Memories

- Scratchpads extremely beneficial regarding energy consumption, runtime and $WCET_{EST}$, as compared to caches and main memories
- Integer-linear programming (ILP) as optimization approach
- SPM contents: Functions, basic blocks and global variables
- SPM allocation for single- and multi-process systems