# Compilers for Embedded Systems

Summer Term 2022

Heiko Falk

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

Hamburg University of Technology

Institute of Embedded Systems

# Chapter 9

# WCET-Aware Compilation

# Outline

# Chapter Contents

# Software Design for Real-Time Systems

**Current Industrial Practice (Automotive, Avionics)**

1. Specification using graphical / high-level tools
2. Automatic generation of ANSI-C code
3. Compilation of binary machine code for a given processor architecture
4. Repeated executions / simulations of generated machine code, usage of “representative” input data
5. Time measurements provide *“observed execution times”*
6. Addition of safety margin (e.g., 20%) to greatest observed execution time: *“observed Worst-Case Execution Time”*
7. Observed WCET ≤ Real-time constraint? No: Go to 1

# Problems of this Design Flow

**Safety**

– No guarantee that observed WCET (even only approximately) matches the actual WCET

☞ No guarantee that a real-time system *always* terminates in time

**Design Time**

– How many iterations are required until step 7 successful?

☞ Depends on in how far steps 2-3 lead to the effective acceleration of the generated code in the worst case

☞ Try & Error until step 7 successful

# Current State of the Art in Compiler Construction

**Objective Function of Compiler Optimizations**

– Usually reduction of *Average-Case Execution Times (ACET)*: Accelerate a “typical” execution of a program using “typical” input data

☞ No statements about the impact of optimizations on WCETs possible

**Optimization Strategy**

– Naive: Current compilers lack precise ACET timing model

– Application of an optimization if “promising”

☞ ACET-related effects of optimizations unknown to compiler

☞ ACET optimizations potentially increase WCETs – Compilers often invoked without any optimizations for real-time systems

# Motivation

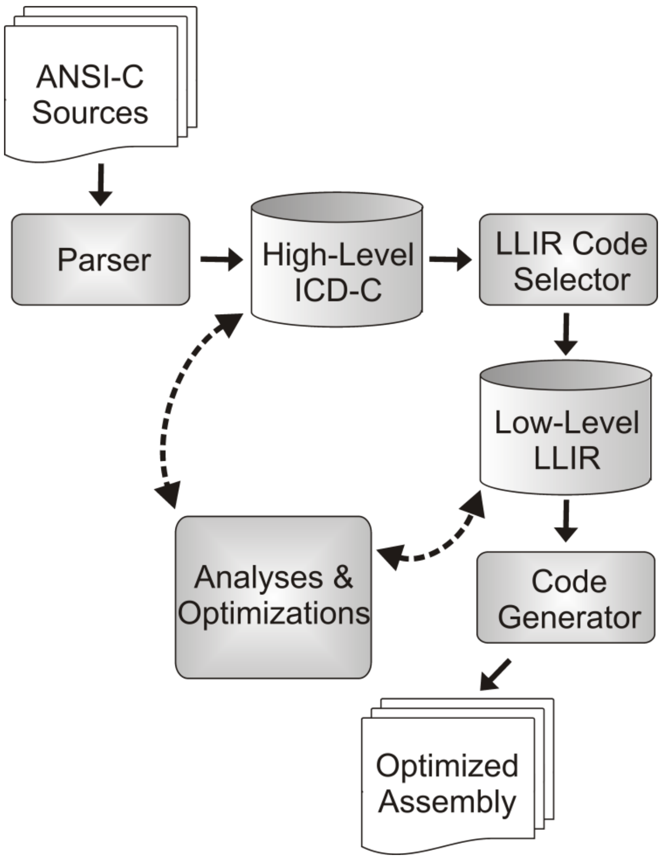
**Design of a Compiler that**

– considers $WCET_{EST}$ instead of average-case runtimes,
– allows formal guarantees on worst-case properties, instead of relying on observed execution times,
– applies fully automated optimizations to minimize $WCET_{EST}$

**Approach**

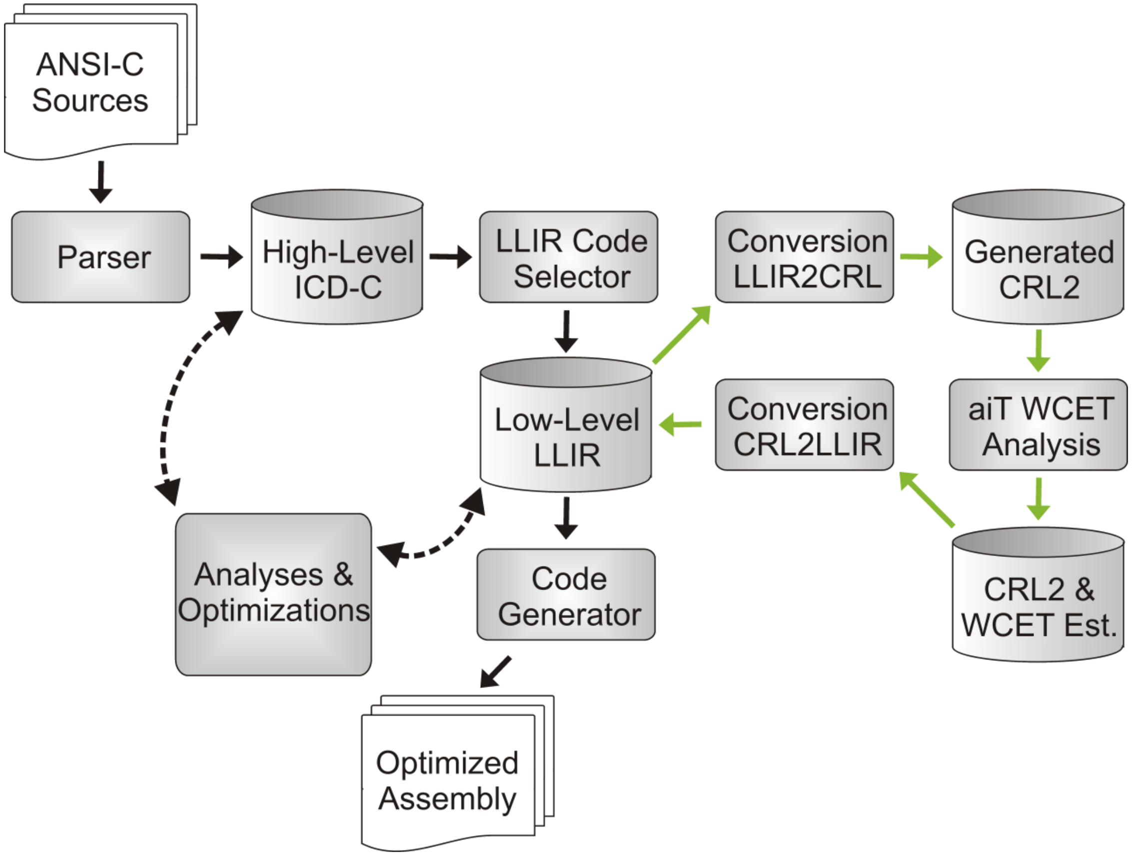
– Integration of a $WCET_{EST}$ timing model into compiler by coupling compiler back-end with static WCET analyzer.
– Exploitation of $WCET_{EST}$ timing model by novel optimizations explicitly aiming at $WCET_{EST}$ minimization.

# Integration of $WCET_{EST}$ Model into Compiler (1)




- Re-implementation of a WCET timing model in compiler makes no sense
- Instead: Tight integration of aiT (☞ *chapter 5)*
- Coupling inside processor-specific compiler back-end *(LLIR)*
- Seamless exchange of information via translation LLIR ↔ CRL2
- Transparent invocation of aiT inside the compiler
- Import of WCET-related data into compiler back-end

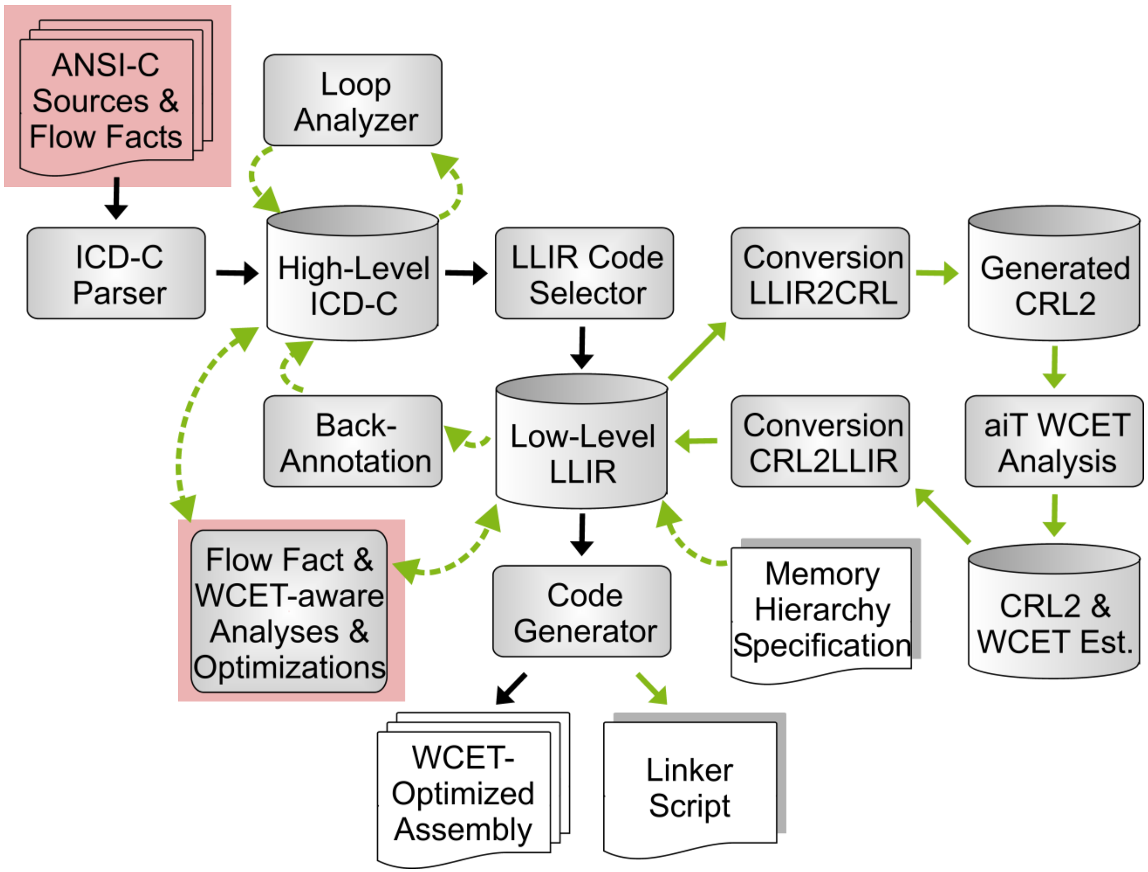
# Integration of $WCET_{EST}$ Model into Compiler (2)




Relevant WCET data:

- $WCET_{EST}$ of entire program, function or basic block
- Worst-case execution frequency per function, basic block or CFG edge
- Potential register contents
- Cache Hits / Misses per basic block

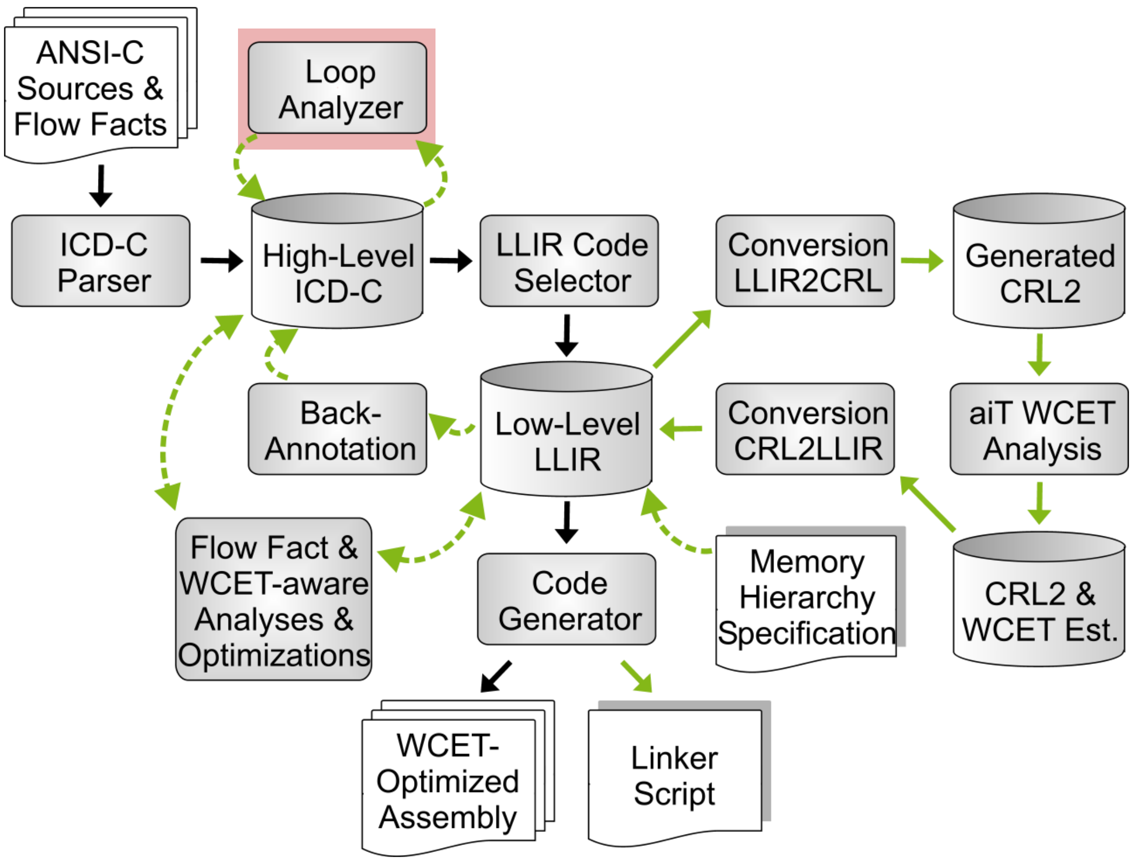
# WCC – The WCET-aware C Compiler (1)




**Flow Facts**

- WCET analysis: max. iteration counts & recursion depths
- WCC: Annotation directly in C source code: `_Pragma( "loopbound min 10 max 10" );`
- Automatic flow fact update during control flow-modifying optimizations

# WCC – The WCET-aware C Compiler (2)



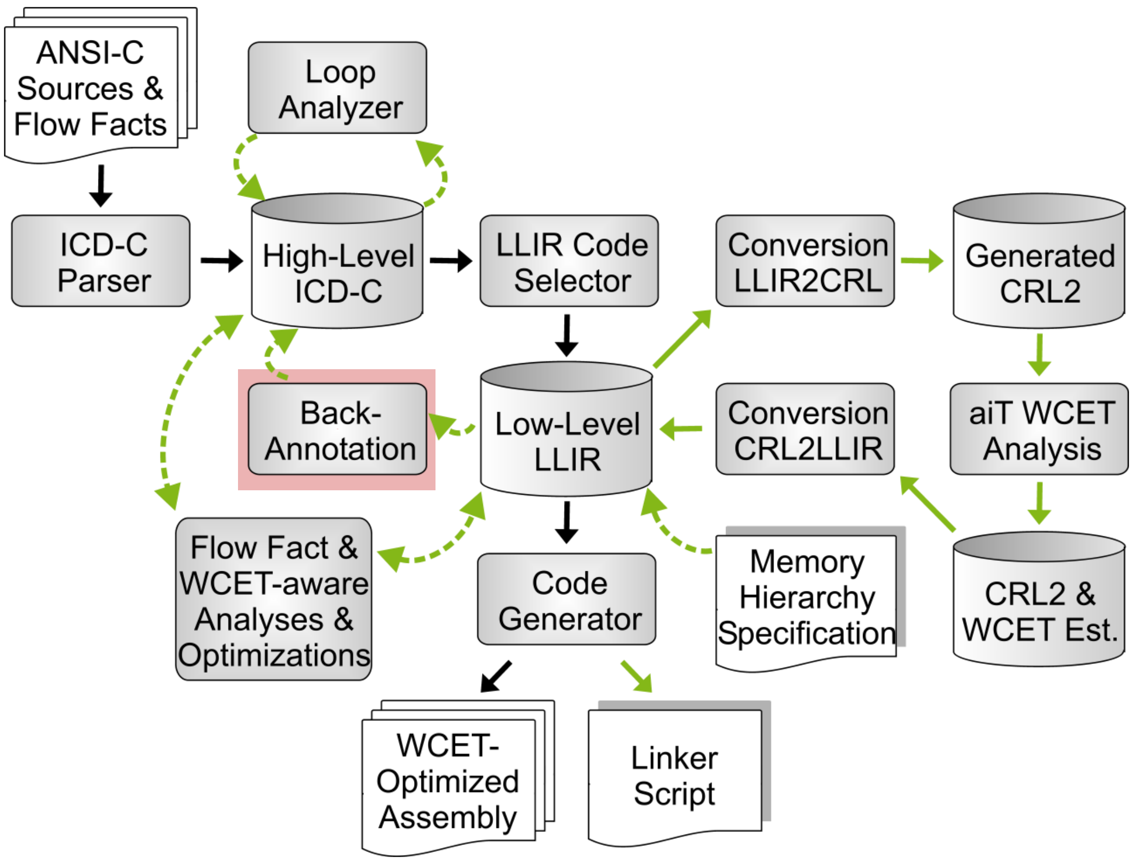

**Loop Analyzer**

– Manual annotation of Flow Facts tedious and error-prone

– WCC: Automated loop analysis that determines maximal iteration counts

– Partially bases on polyhedral models (☞ *chapter 4)*

# WCC – The WCET-aware C Compiler (3)



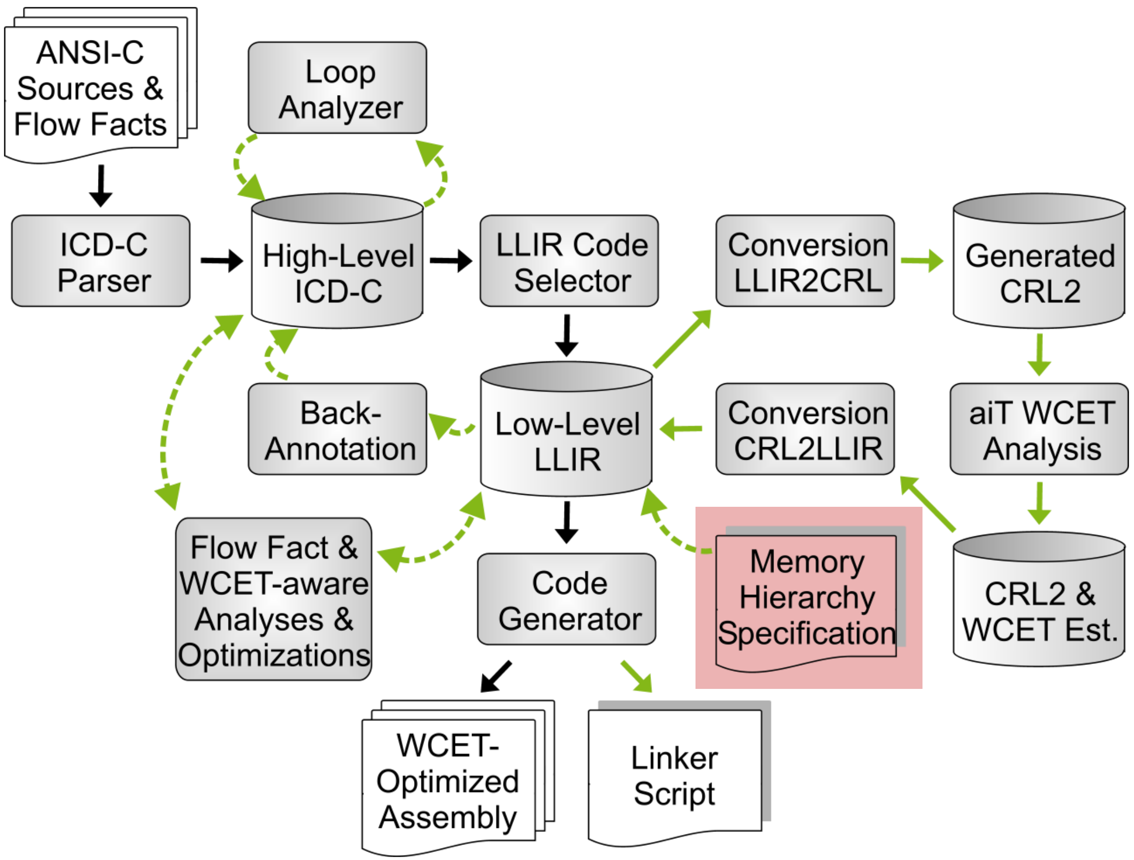

## Back-Annotation

- $WCET_{EST}$ data of aiT only available in back-end
- HIR optimizations have no accesses to $WCET_{EST}$ data

☞ $WCET_{EST}$ minimization at HIR level impossible

- WCC: Back-annotation translates $WCET_{EST}$ data from LIR to HIR

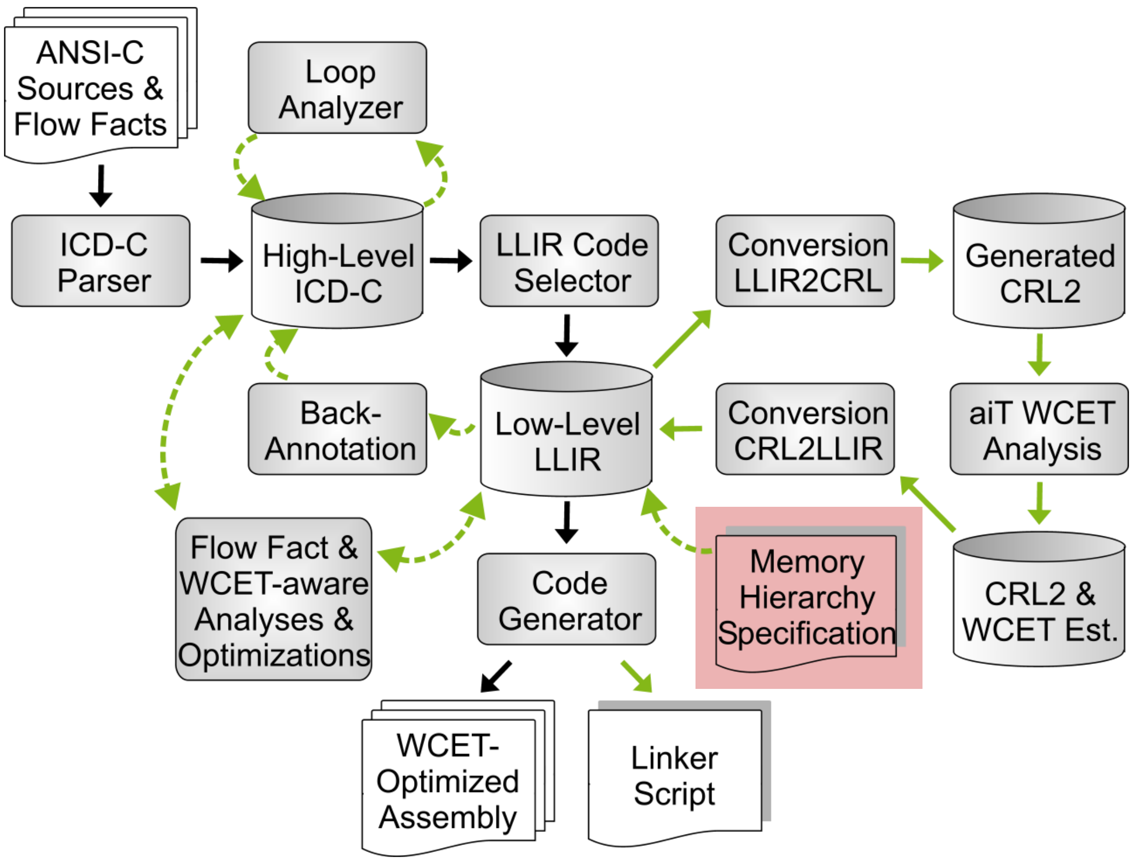
# WCC – The WCET-aware C Compiler (4)




**Memory Hierarchy**

- aiT operates on binary code using physical addresses
- WCC must provide correct physical addresses for code, data, branches and load/store operations to aiT
- WCC requires detailed knowledge about memories

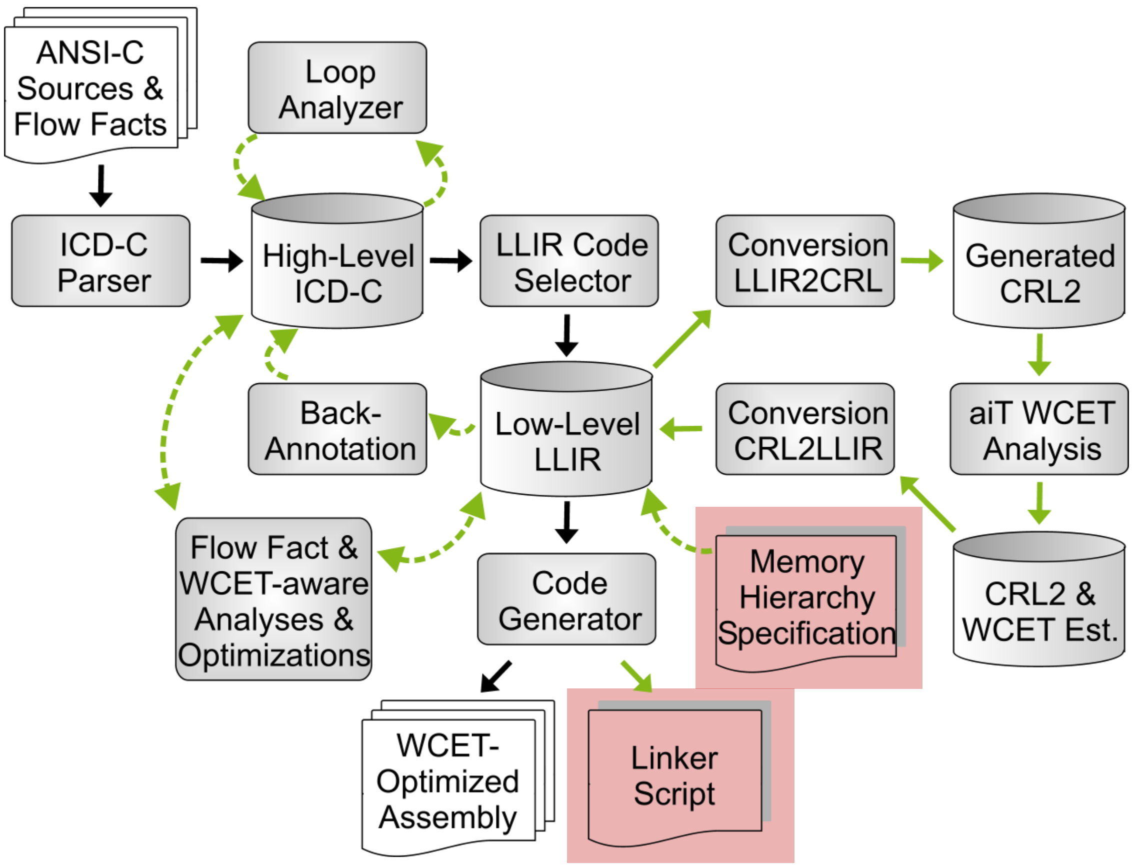
# WCC – The WCET-aware C Compiler (5)




**Memory Hierarchy**

- Memory regions, their start addresses, sizes, access latencies, access attributes (code, data, read-/writable, ...)
- SPM allocations also require this information for their optimizations

# WCC – The WCET-aware C Compiler (6)




**Memory Hierarchy**

- WCC decides on memory layout of code and data but produces no binary code
- Linker must generate binary code in strict compliance with WCC's memory layout
- WCC: Automatic generation of an adapted linker script

[http://www.tuhh.de/es/esd/research/wcc]

# Challenges during $WCET_{EST}$ Minimization

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

- WCET of a program = Length of the program's longest execution path (WCEP)
- $WCET_{EST}$ minimization: Optimization of only those parts of a program lying on the WCEP

☞ Code optimization apart the WCEP will not reduce $WCET_{EST}$

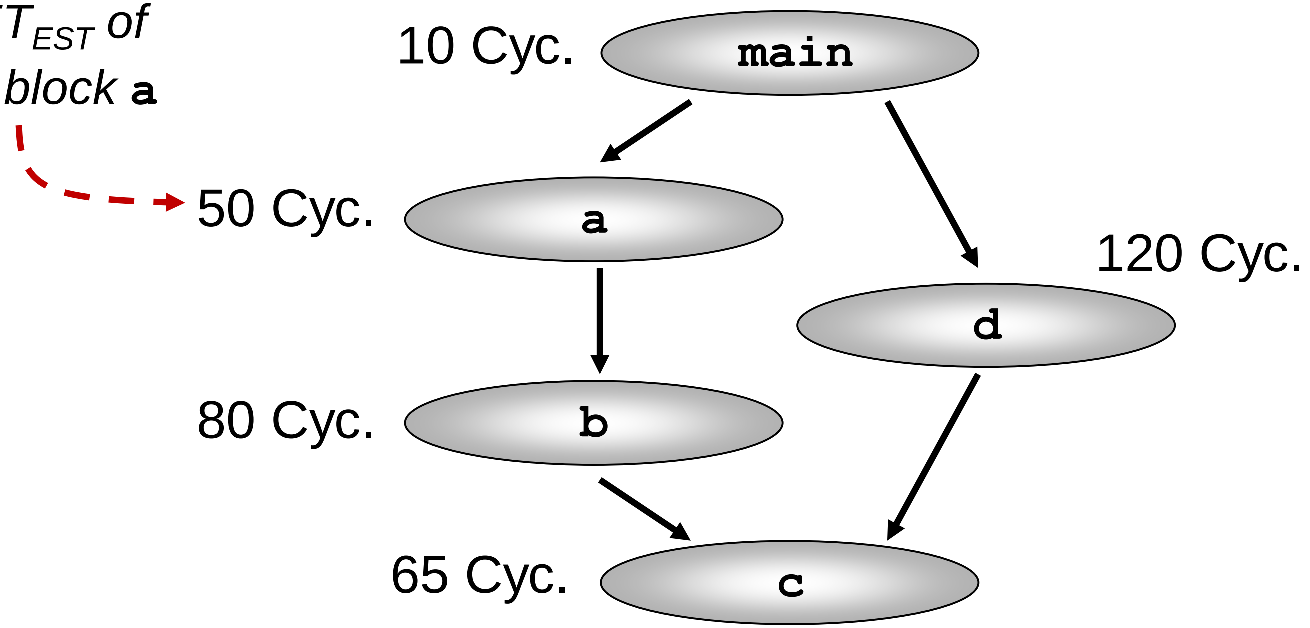
☞ Optimizations minimizing $WCET_{EST}$ require detailed knowledge of the WCEP!

☞ *WCET analyzer aiT provides such detailed information by means of execution frequencies of CFG edges.*

*But...*

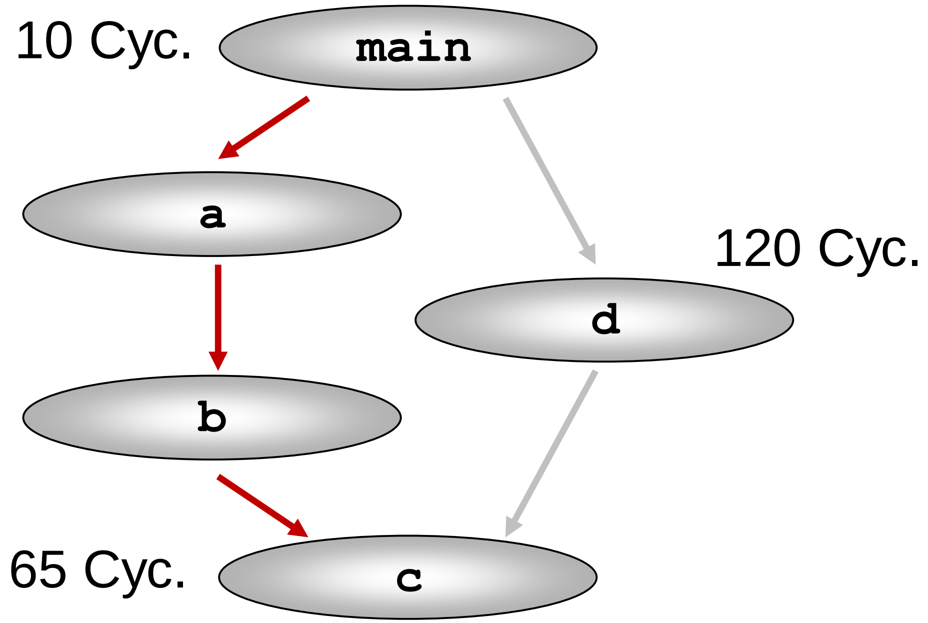
# Instability of the WCEP (1)

*$WCET_{EST}$ of basic block* **`a`**




– Example: Simple CFG with 5 basic blocks

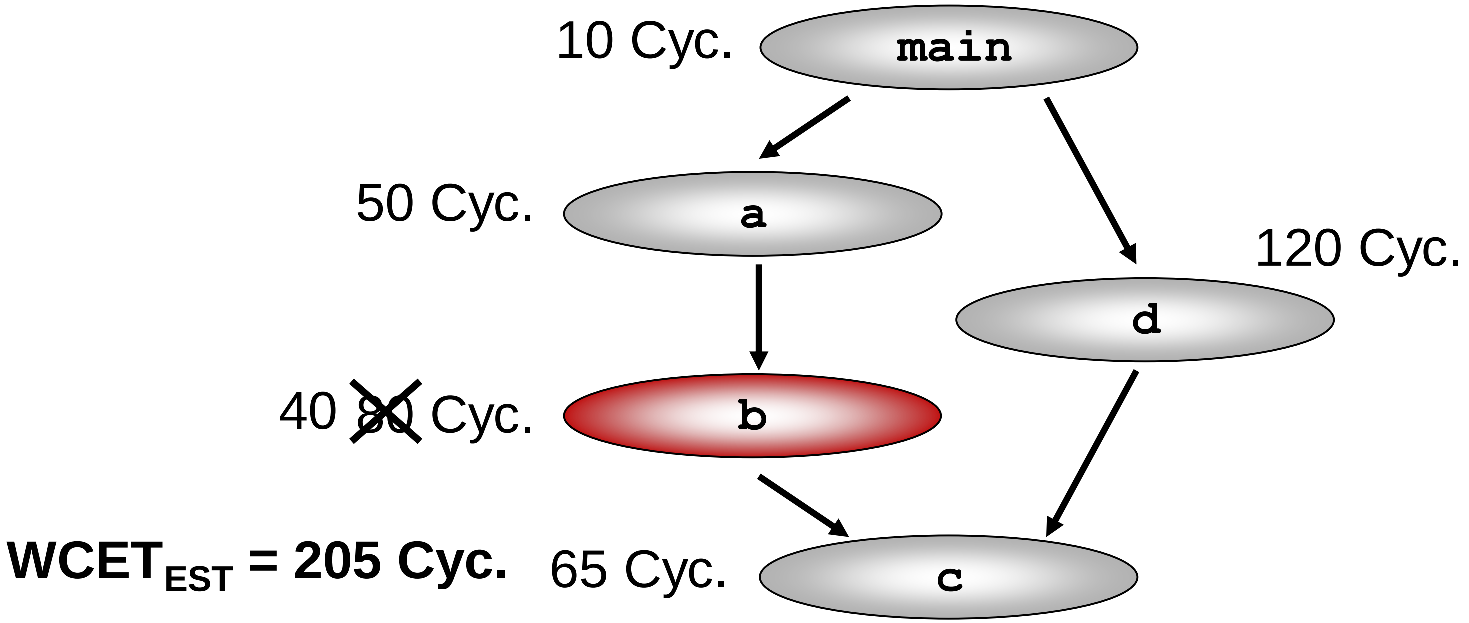
# Instability of the WCEP (2)




**$WCET_{EST}$ = 205 Cyc.**

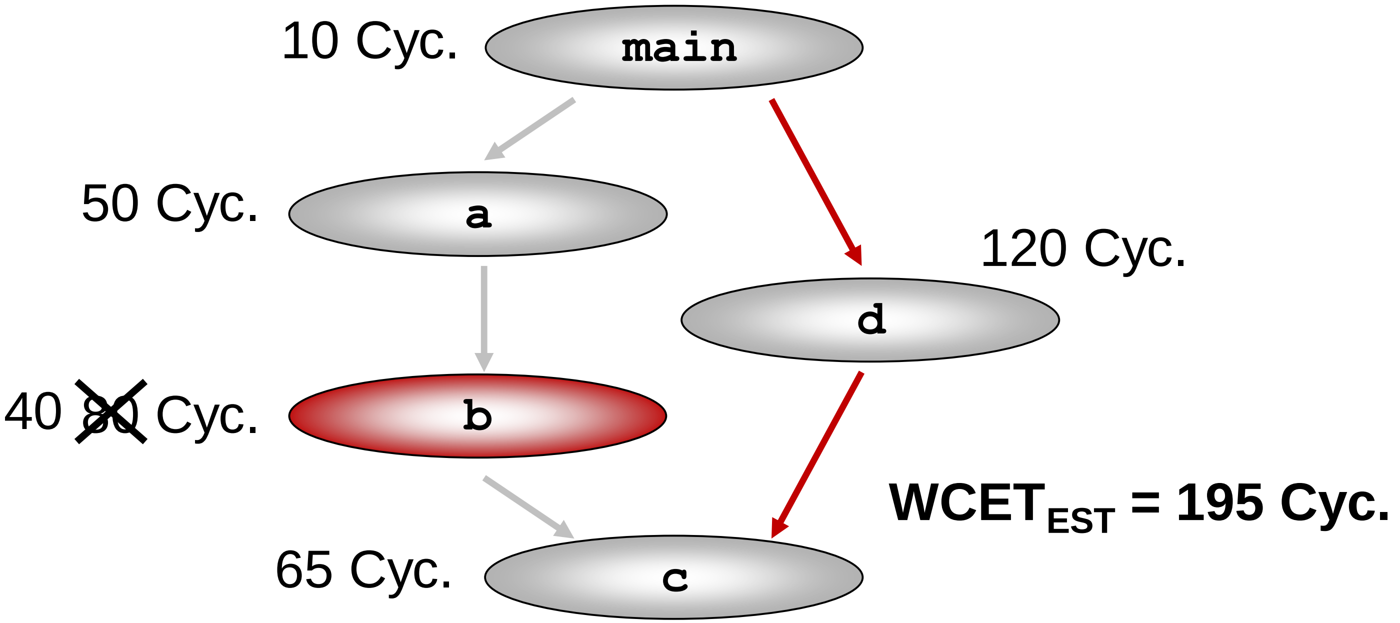
- Initial WCEP: **`main`**, **`a`**, **`b`**, **`c`**
- Length of WCEP = $WCET_{EST}$ = 205
- In the following: Optimization of **`b`**

# Instability of the WCEP (3)




- Initial WCEP: **`main`**, **`a`**, **`b`**, **`c`**
- Length of WCEP = $WCET_{EST}$ = 205
- In the following: Optimization of **`b`**

# Instability of the WCEP (4)




– Novel WCEP: **`main`**, **`d`**, **`c`**

– Novel $\text{WCET}_{\text{EST}}$: 195

☞ ***WCEP has changed due to an optimization!***

# Consequences for Compiler Optimizations

**WCET-Aware Optimizations...**

– ... always have to be aware that the WCEP can change after each individual optimization decision.

– ... should take the decision where to optimize something not only based on local information, but should always consider the global effects of an optimization decision.
*(The optimization of* `b` *in the previous example locally reduces the* $WCET_{EST}$ *of* `b` *by 40 cycles. But globally, only 10 cycles were saved!)*

☞ ***Challenge: To design novel compiler optimizations that fulfill the above requirements and that always consider the entire CFG and the current WCEP therein.***

# Chapter Contents

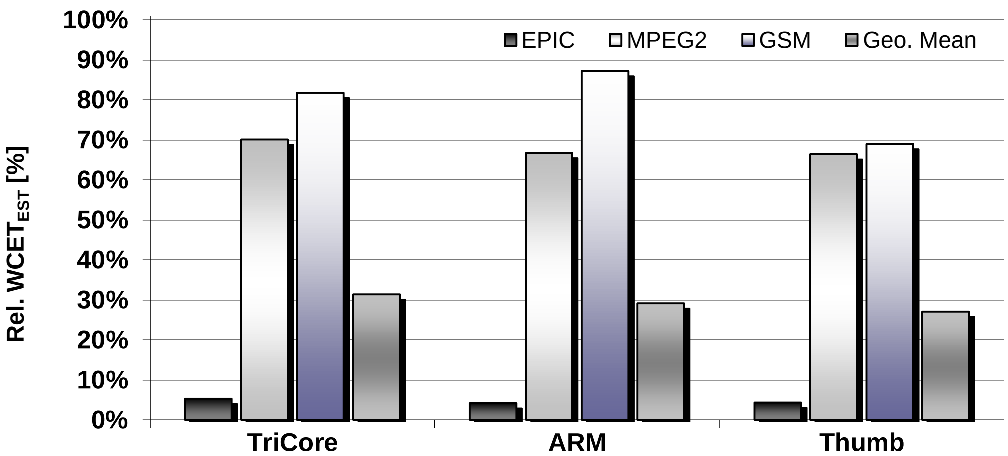
# Why Procedure Cloning and $\text{WCET}_{\text{EST}}$?

**Motivation** *(☞ cf. chapter 5)*

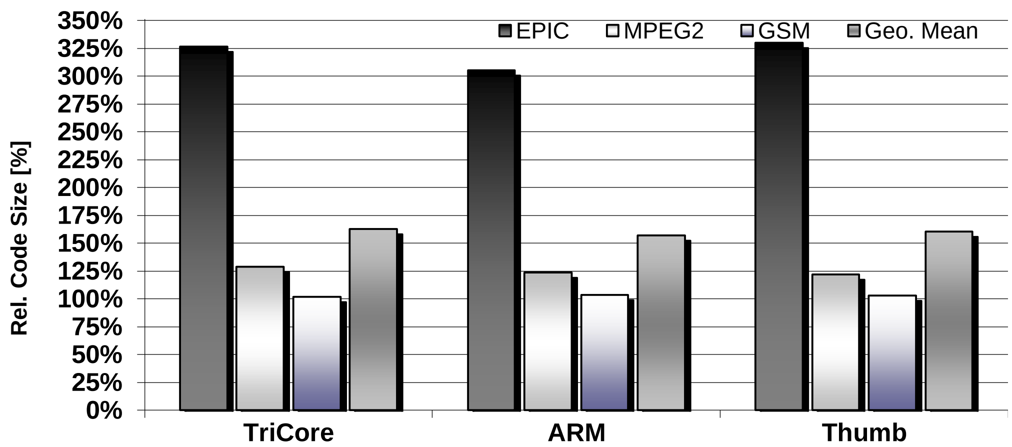
– Frequent occurrences of general-purpose functions in special-purpose contexts in embedded software
– Loop bounds are particularly often controlled by function parameters
– Loop bounds are particularly critical for WCET estimates
– Procedure Cloning allows the extremely precise annotation of loop bounds for WCET analysis

*[**P. Lokuciejewski.** Influence of Procedure Cloning on WCET Prediction. CODES+ISSS, Salzburg, 2007]*

# Results after Classical Cloning




– $WCET_{EST}$ improvements from 13% up to 95%!




– Code size increases from 2% up to 325%!

# Key Problems of Classical Cloning

- $WCET_{EST}$ of a program corresponds to length of WCEP
- Classical Procedure Cloning is fully unaware of WCEP
- Properties of functions that potentially yield $WCET_{EST}$ reductions (parameter-dependent loops) are not considered by the classical standard optimization

☞ Potential cloning of functions that do not lie on the WCEP

☞ Potential cloning of functions that do not contribute to a $WCET_{EST}$ reduction

☞ Unnecessary code size increases without any benefit in terms of $WCET_{EST}$ reduction

# WCET-aware Cloning (1)

**Input**

– Program *P* to be optimized, given in the form of an HIR

– Float value *maxFactor* that denotes the maximally acceptable code size increase

**Initialization**

*maxCodeSize* = getCodeSize( *P* ) * *maxFactor*;

**Phase 1 – Determination of the WCEP**

Perform a WCET analysis of *P*;

Determine set *F* of all original functions lying on the WCEP of *P*;

$wcet_{orig}$ = getWCET( *P* );

$cs_{orig}$ = getCodeSize( *P* );

# WCET-aware Cloning (2)

**Phase 2 – Determination of $WCET_{EST}$ Data per Function**

```
for ( <all functions f ∈ F> )
  if ( <f is called with constant value as some parameter p> &&
      ( <p is used as loop bound> ||
        <p is used in a condition of an if-statement> ||
        <p is argument in some other function call inside f> ) )
    // Cloning of f eventually beneficial w.r.t. WCET_EST
    HIR P' = P.copy();
    doCloning( P', f );                    // Try out cloning of f
    updateLoopBounds( P', f );
    deleteRedundantIfStmts( P', f );
    Perform WCET analysis of P';
    wcet_f = getWCET( P' );
    cs_f = getCodeSize( P' );
```

# WCET-aware Cloning (3)

**Phase 3 – Determination of that Function with highest Profit**

for ( <*all functions* $f \in F$> )

$profit_f = ( wcet_{orig} – wcet_f ) / ( cs_f – cs_{orig} )$;

Determine function $f_{opt}$ with maximal $profit_f$ AND

$cs_f \leq maxCodeSize$;

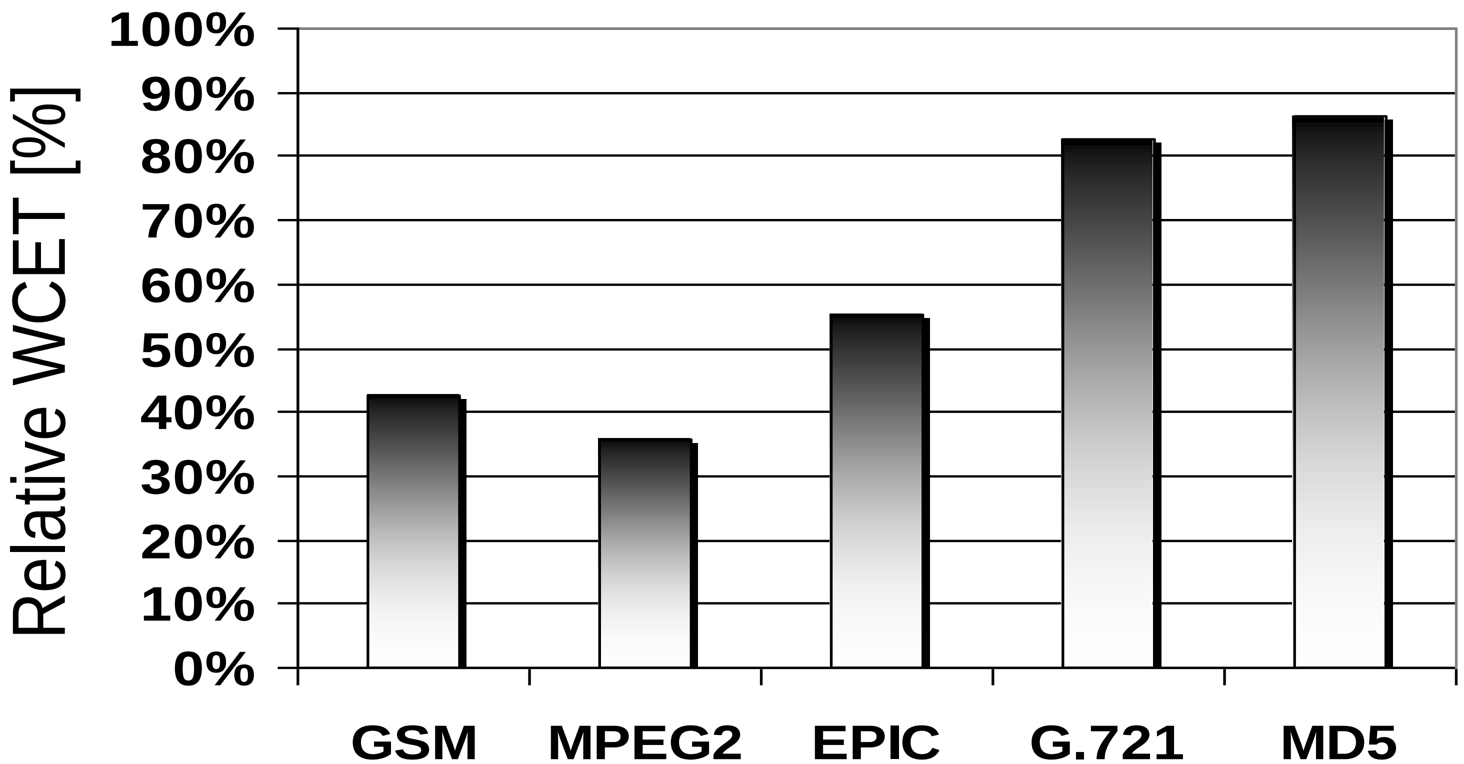
if ( <$f_{opt}$ exists> )

doCloning( $P$, $f_{opt}$ );

goto <*Phase 1*>;

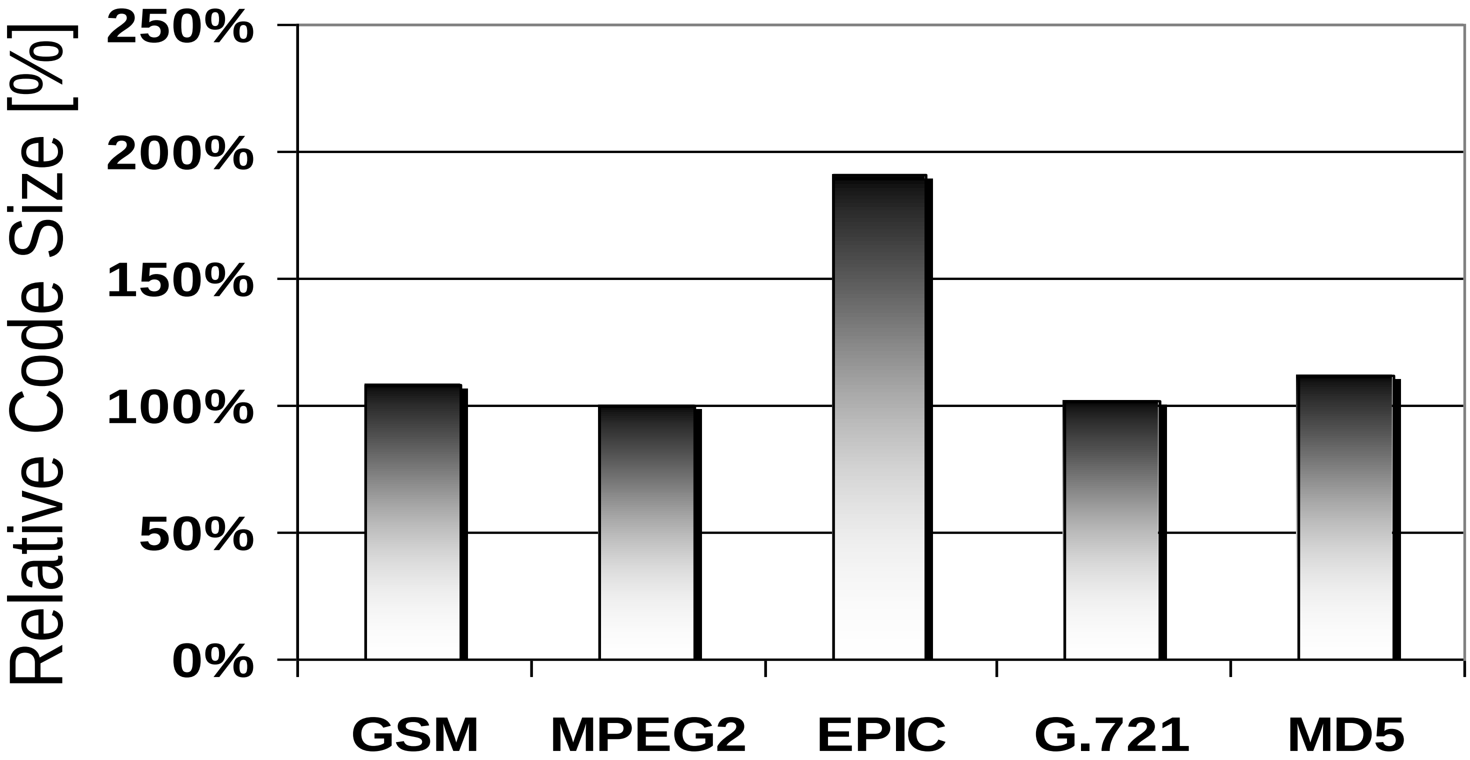
*[**P. Lokuciejewski.** WCET-Driven, Code-Size Critical Procedure Cloning. SCOPES, Munich, 2008]*

# Relative $WCET_{EST}$ after WCET-aware Cloning




- 100% = $WCET_{EST}$ without any procedure cloning
- $WCET_{EST}$ reductions from 14% up to 64%!

# Relative Code Sizes after WCET-aware Cloning

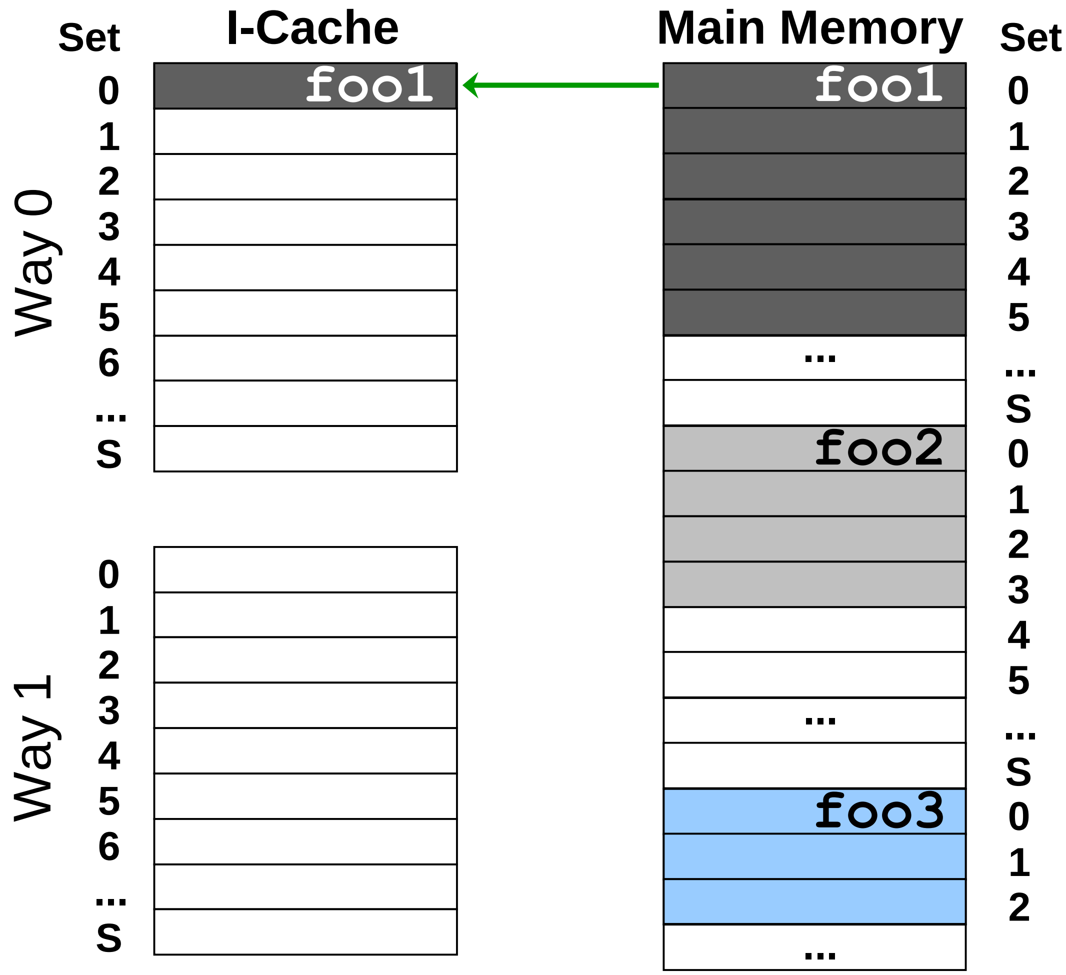



- 100% = Code size without any procedure cloning
- Code size increase of EPIC: 190% instead of 300%

# Eviction of Code from Instruction Caches

- Caches exploit locality of memory accesses
  - *Spatial Locality:* Memory accesses target a spatially small memory region that should be kept in the cache completely
  - *Temporal Locality:* In a short period of time, spatially scattered memory regions are accessed so that these regions should be kept in the cache
- Poor layout of code (or data) in memory can lead to a bad cache performance if temporal locality is high:
- Scattered memory regions with high temporal locality can, when arranged badly in memory, evict themselves repeatedly from the cache, thus yielding many cache misses – so-called *conflict misses*.

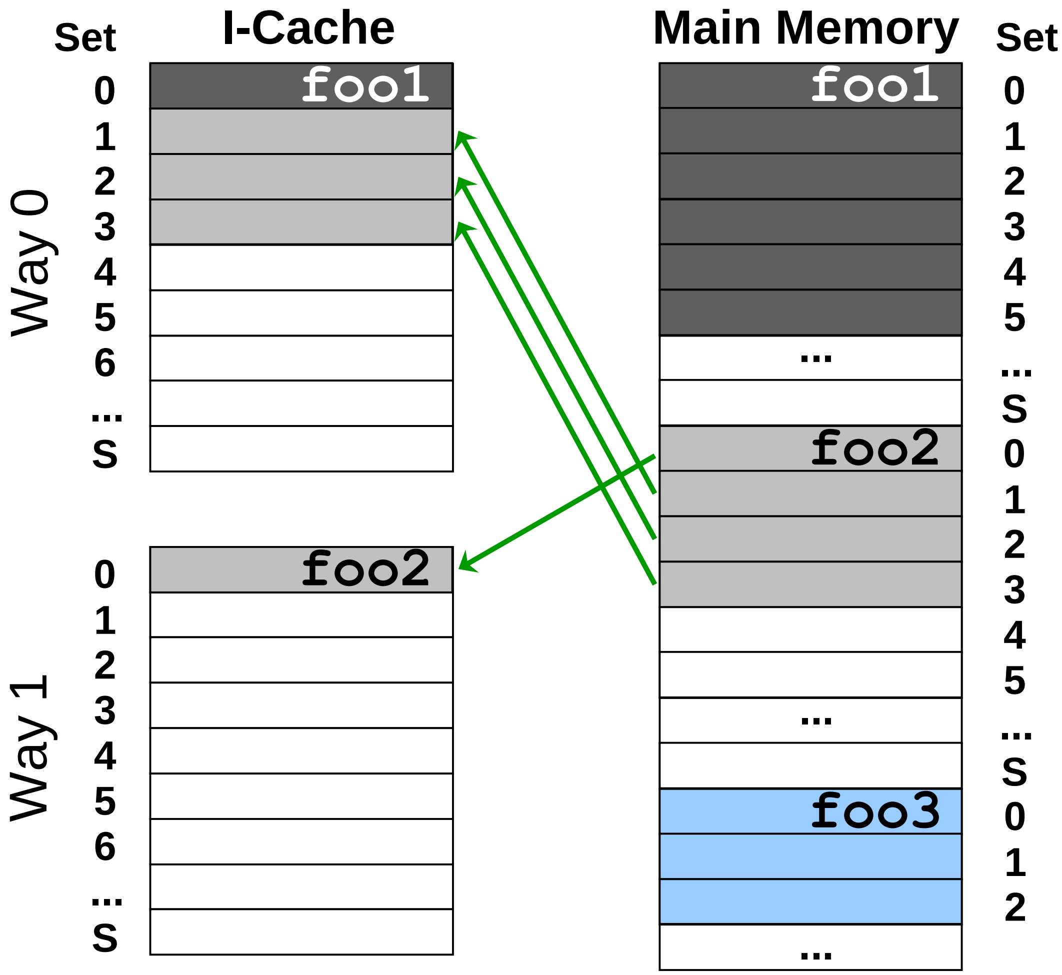
# Example of I-Cache Evictions (1)




```
void foo1()
{
  for (i=0; i<n; i++) {
    foo2();
    foo3();
    // More Code
  }
}
```

*Here:* 2-way set-associative I-Cache

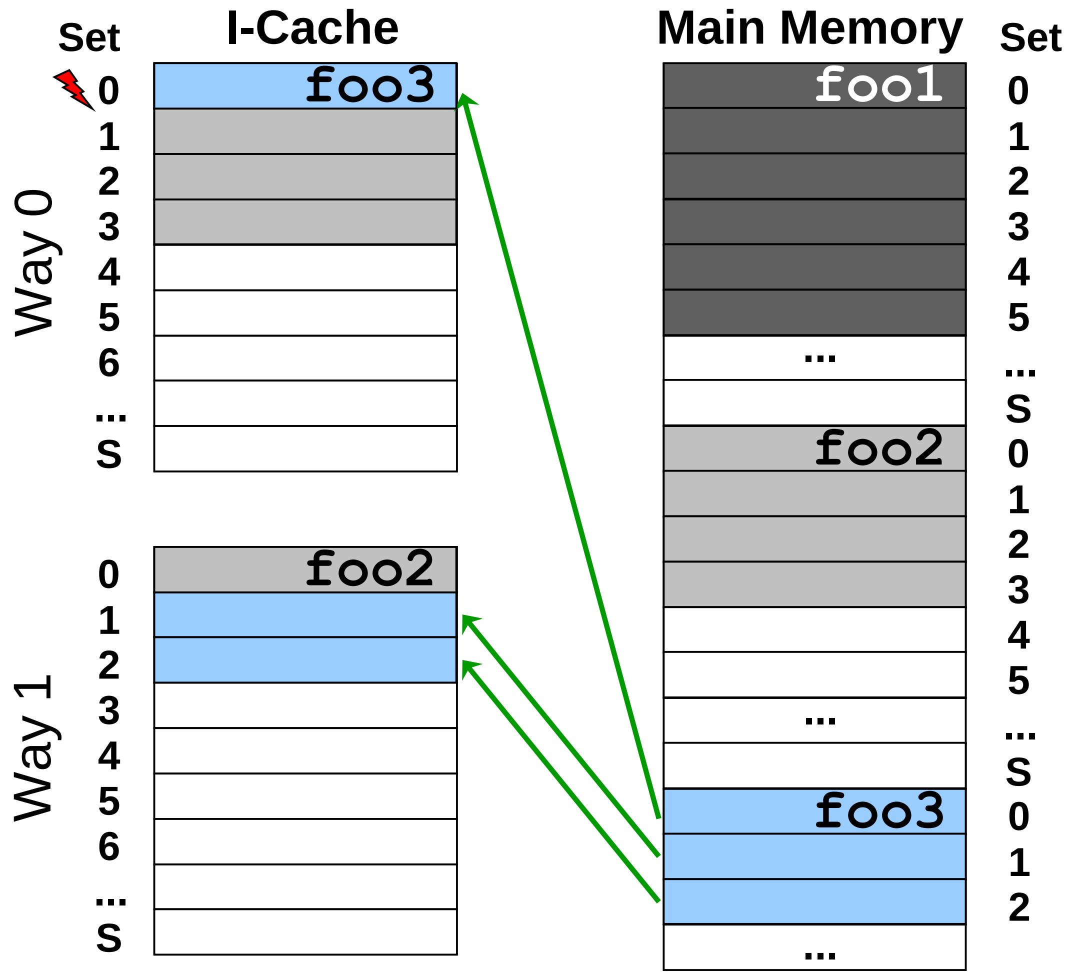
# Example of I-Cache Evictions (2)




```
void foo1()
{
  for (i=0; i<n; i++) {
    foo2();
    foo3();
    // More Code
  }
}
```

*Here:* 2-way set-associative I-Cache

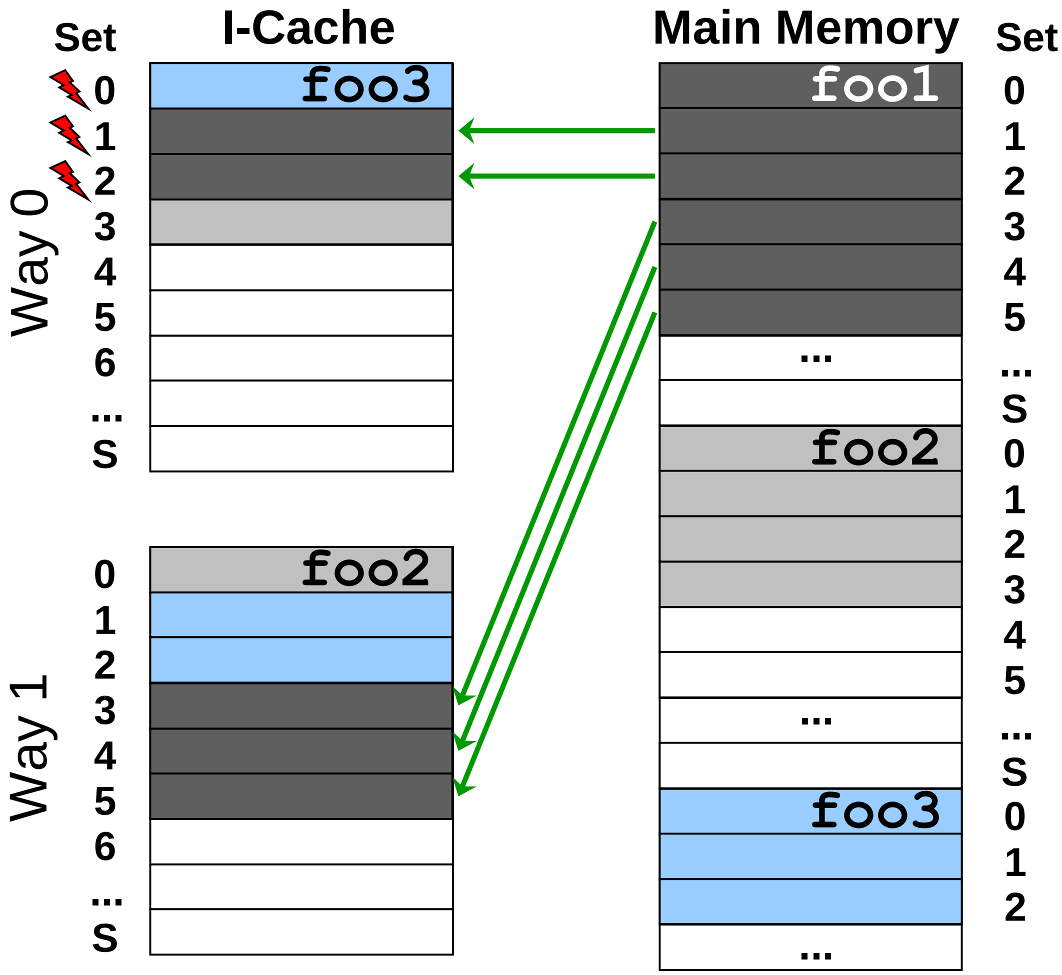
# Example of I-Cache Evictions (3)




```
void foo1()
{
   for (i=0; i<n; i++) {
      foo2();
      foo3();
      // More Code
   }
}
```

*Here:* 2-way set-associative I-Cache

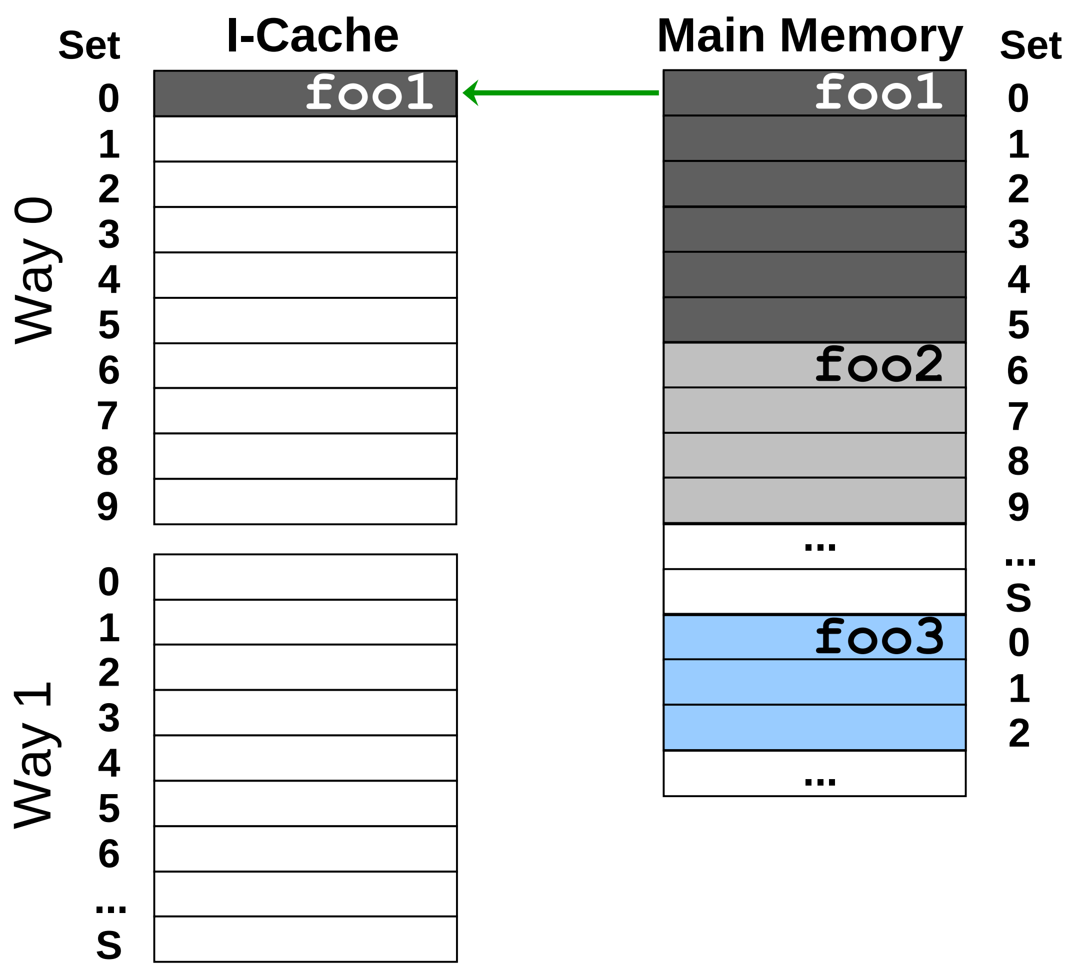
# Example of I-Cache Evictions (4)




```
void foo1()
{
  for (i=0; i<n; i++) {
    foo2();
    foo3();
    // More Code
  }
}
```

*Here:* 2-way set-associative I-Cache

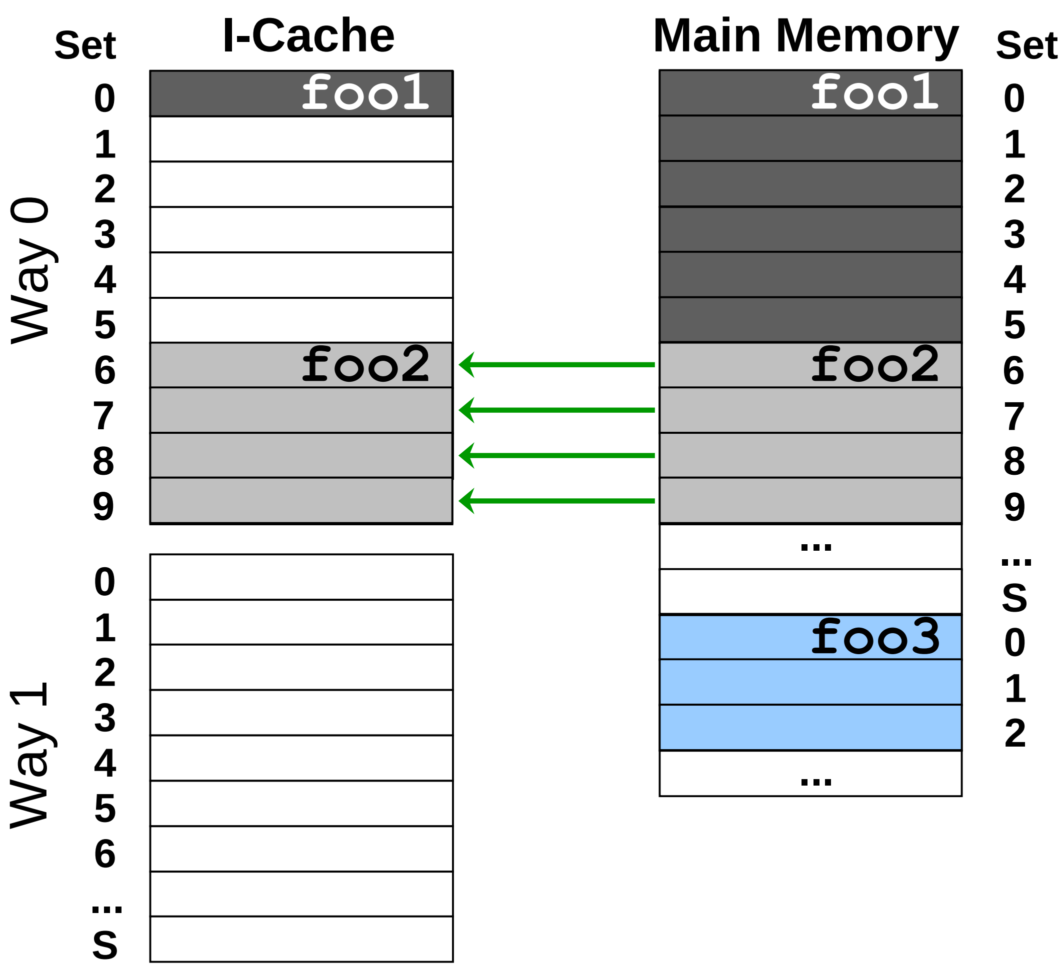
# A better Memory Layout without Evictions (1)




```
void foo1()
{
  for (i=0; i<n; i++) {
    foo2();
    foo3();
    // More Code
  }
}
```

*Here:* 2-way set-associative I-Cache

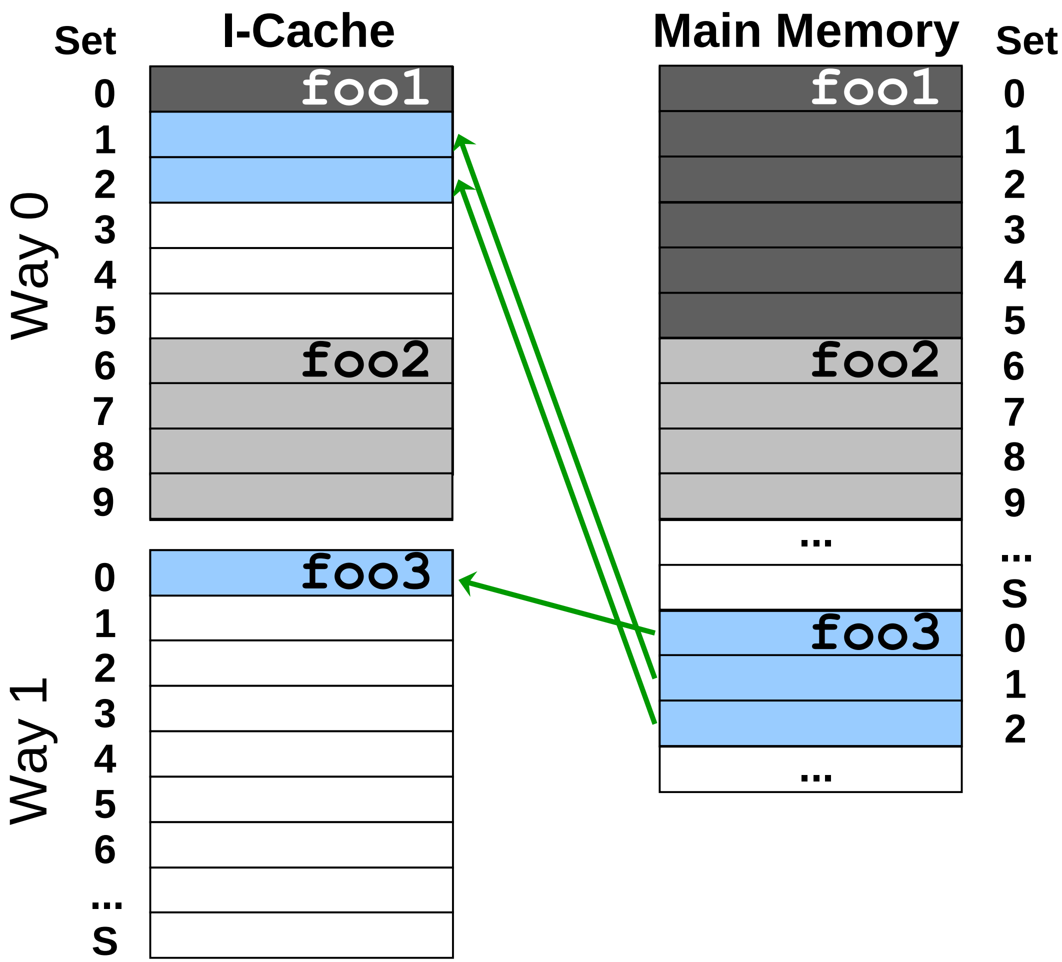
# A better Memory Layout without Evictions (2)




```
void foo1()
{
  for (i=0; i<n; i++) {
    foo2();
    foo3();
    // More Code
  }
}
```

*Here:* 2-way set-associative I-Cache

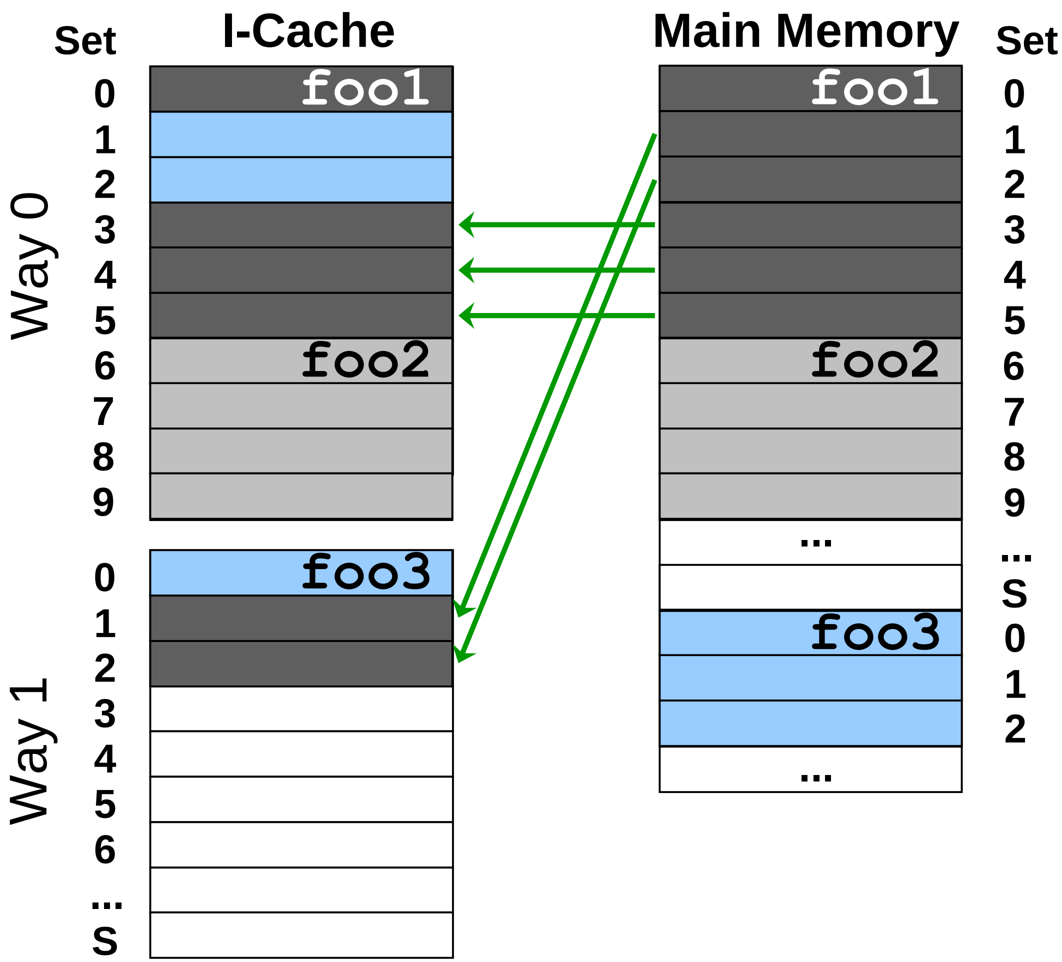
# A better Memory Layout without Evictions (3)




```
void foo1()
{
   for (i=0; i<n; i++) {
      foo2();
      foo3();
      // More Code
   }
}
```

_Here:_ 2-way set-associative I-Cache

# A better Memory Layout without Evictions (4)




```
void foo1()
{
   for (i=0; i<n; i++) {
      foo2();
      foo3();
      // More Code
   }
}
```

*Here:* 2-way set-associative I-Cache

# Procedure Positioning using Call Graphs

**Definition *(Call Graph)*:**

The *Call Graph* is an undirected weighted graph $G = (V, E, w)$ with

- $V$ contains a node $v$ per function of a program
- $E$ contains an edge $e = \{v, w\}$ if a function $v$ calls function $w$
- Each edge $e = \{v, w\}$ is weighted with the frequency $w(e)$ how often $v$ and $w$ call themselves

**Concept of WCET-aware Procedure Positioning**

- Generate call graphs with edge weights equal to worst-case call frequencies as determined during static WCET analysis
- Repeatedly place two functions with high edge weights consecutively in memory

# WCET-aware Procedure Positioning (1)

**Input**

– Program $P$ to be optimized, given in the form of an LIR

**Initialization**

Perform a WCET analysis of $P$;

Generate call graph $G_{orig} = (V_{orig}, E_{orig}, w_{orig})$ of $P$ based on WCET data;

Generate call graph $G_{new} = (V_{new}, E_{new}, w_{new}) = G_{orig}$.copy();

*[**P. Lokuciejewski et al.** WCET-driven Cache-based Procedure Positioning Optimizations. ECRTS, Prague, 2008]*

# WCET-aware Procedure Positioning (2)

**Optimization Loop**

do
  $wcet_{current}$ = getWCET( *P* );
  for ( <*all edges e* = {*v*, *w*} $\in$ $E_{new}$,
      *sorted in descending order w.r.t.* $w_{new}$> )
    if ( Positioning( *e*, $G_{new}$, $G_{orig}$, *P*, $wcet_{current}$ ) == true )
      *// If contiguous placement of nodes v and w in memory reduces*
      *// $WCET_{EST}$, terminate for-loop and continue with do-while-loop.*
      break;
while ( <*P was modified during last iteration*> );

# WCET-aware Procedure Positioning (3)

**Positioning(** $e = \{v, w\} \in E_{new}, G_{new}, G_{orig}, P, wcet_{current}$ **)**

Generate LIR *P'* with *v* and *w* placed contiguously in memory;

Perform a WCET analysis of *P'*;

$wcet_{new}$ = getWCET( *P'* );

if ( $wcet_{new} < wcet_{current}$ )

  *P* = *P'*;

  Merge nodes *v* and *w* in $G_{new}$;

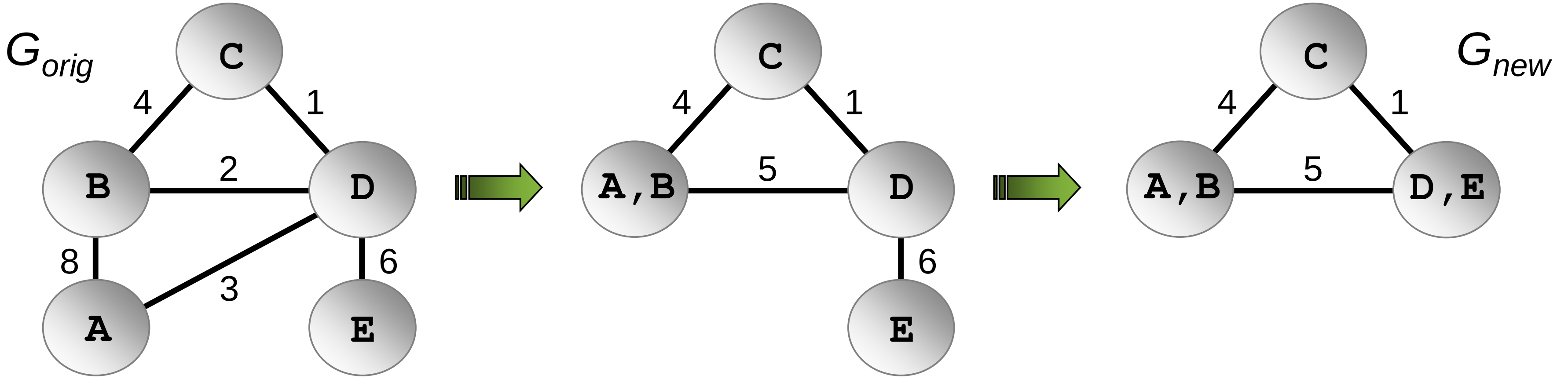
  Update $w_{new}$ based on novel WCET data;

  return true;

else

  return false;

# Merging of Nodes




**Contiguous Placement of merged Nodes in Memory**

– Problem: How shall (**A**, **B**) and (**D**, **E**) be placed in the next step?

– $G_{orig}$ reveals that **A** and **D** should be placed contiguously

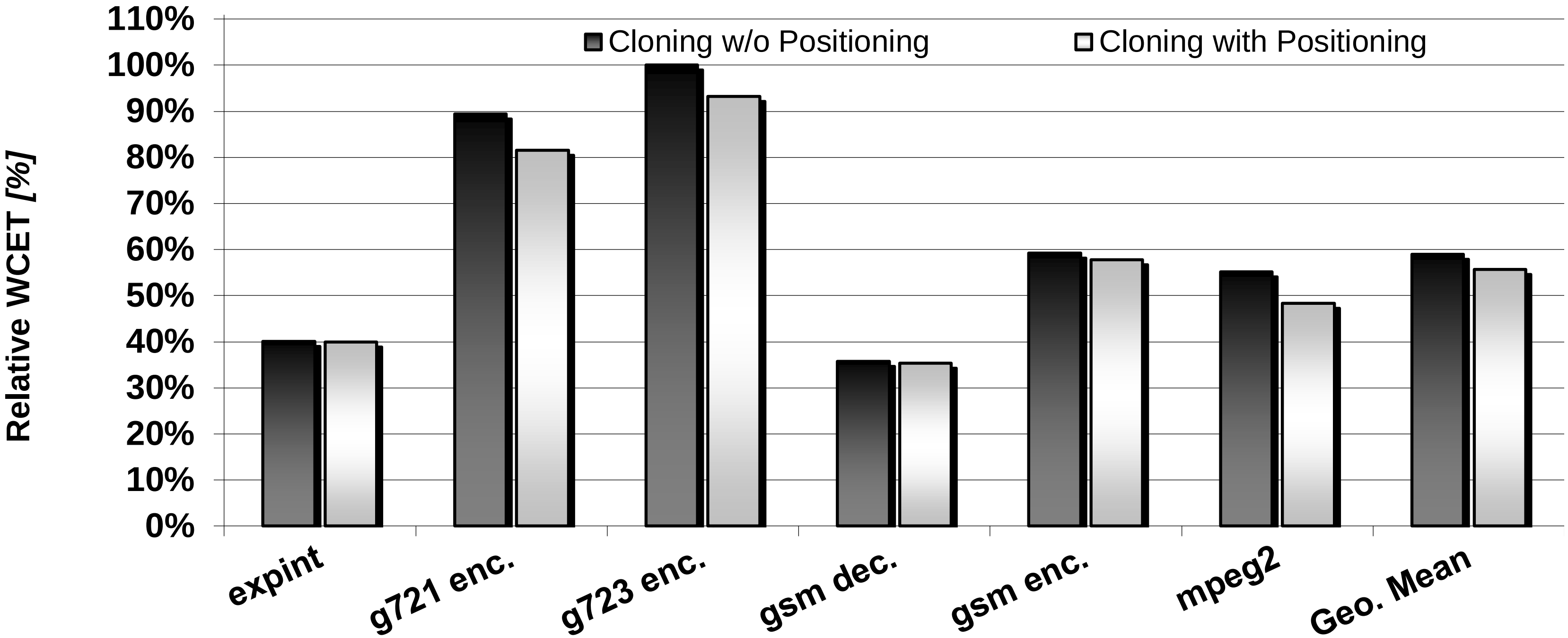
☞ Best placement is (**B**, **A**, **D**, **E**).

☞ That's why $G_{orig}$ is kept throughout the positioning algorithm!

# Properties

- Algorithm greedily places two nodes of the current graph $G_{new}$ contiguously in memory in one iteration
- Here, always those two nodes are considered which call themselves most frequently according to $w_{new}$
- Instable WCEPs are considered by the algorithm, because a WCET analysis is done for each placement, and because the edge weights $w_{new}$ are updated according to this novel WCET data
- Since WCET-aware Procedure Cloning places the novel clones at the end of a program, it makes sense to combine WCET-aware Cloning and Positioning

# Relative $WCET_{EST}$ after WCET-Cloning & Positioning




– 100% = $WCET_{EST}$ w/o Procedure Cloning and Positioning
– I-Cache: 16kB, 2-way set-associative, LRU replacement
– WCET-Positioning of clones: additional $WCET_{EST}$ reduction by up to 7% compared to cloning w/o positioning

***Caution:*** *Don't compare this diagram with slide 30, since I-cache was disabled in slide 30!*

# Conclusions

**Comparison with Consequences for $\mathbf{WCET_{EST}}$ Optimizations**

WCET-aware optimizations...

- ... mandatorily need detailed knowledge of the WCEP
- 👍 *WCET-Cloning and WCET-Positioning both consider the WCEP*
- ... always have to be aware that the WCEP can change after each individual optimization decision
- 👍 *Both optimizations update the WCEP after each modification of the code*
- ... should take the decision where to optimize something not only based on local information, but should always consider the global effects of an optimization decision
- 👎 *WCET-Cloning and WCET-Positioning are both greedy heuristics that are driven solely by local data per function*

# Chapter Contents

# Register Allocation by Graph Coloring

1. **Build:** Create interference graph $G = (V, E)$ with
   $V$ = {*virtual registers*} ∪ {*K physical processor registers*},
   $e = (v, w) \in E \Leftrightarrow$ VREGs *v* and *w* may never share the same PHREG,
   i.e. *v* and *w* interfere
2. **Simplify:** Remove all nodes $v \in V$ with degree $< K$
3. **Spill:** After step 2, each node of *G* has degree $\geq K$. Select one node $v \in V$; mark *v* as *potential spill*; remove *v* from *G*
4. **Repeat** Simplify and Spill until $G = \emptyset$
5. **Select:** Re-insert nodes *v* into *G* in reverse order; if there is a free color $k_v$, color *v*; otherwise, mark *v* as *actual spill*
6. **Generate Spill Code** before/after actual spills; go to step 1 if #VREGS > 0

*[**A. W. Appel.** Modern compiler implementation in C. 2004]*

# Problem of Standard Graph Coloring

3. **Spill:** After step 2, each node of *G* has degree ≥ *K*. Select one node *v* ∈ *V*; mark *v* as *potential spill*; remove *v* from *G*

***Which node v should be selected as potential spill?***

Common graph coloring implementations select ...

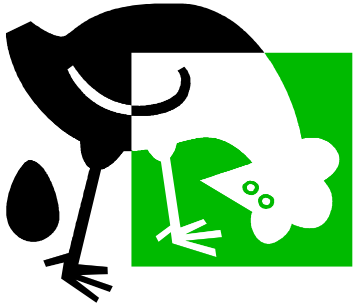
– ... the first node *v* according to the order in which VREGs were generated during code generation,

– ... the node with highest degree in the interference graph,

– ... a node with high degree, with few DEFs/USEs, not in some inner loop – maybe depending on profiling data.

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

# A Chicken-Egg Problem



**A WCET-aware Register Allocator...**

– ... relies on WCET data provided by WCET analysis using aiT,
– ... but cannot obtain WCET data since code containing virtual registers is not executable and thus not analyzable!

**The Way Out**

– Start by marking all VREGs as actual spill
  ☞ Code has lousy quality, but is fully analyzable
– Perform WCET analysis, get WCEP *P*
– Apply standard graph coloring to all VREGs of that basic block $b \in P$ with most executions of spill code in the worst case
– Re-evaluate novel WCEP

# WCET-aware Graph Coloring (1)

```
LLIR WCET_GC_RA( LLIR P )
{
  // Iterate until current WCEP is fully allocated.
  while ( true )
  {
    // Copy P, spill all VREGs of P′ onto stack.
    LLIR P′ = P.copy();
    P′.spillAllVREGs();

    // Compute Worst-Case Execution Path for fully spilled LIR.
    set<basic_blocks> WCEP = computeWCEP( P′ );

    // If there are no more VREGs, the allocation loop is over.
    if ( getVREGs( WCEP ) == ∅ )
      break;
```

# WCET-aware Graph Coloring (2)

```
    // Determine that block on the WCEP with highest product of
    // Worst-Case Execution Count * spilling instructions.
    basic_block b′ = getMaxSpillCodeBlock( WCEP );
    basic_block b = getBlockOfOriginalP( b′ );

    // Collect all VREGs of this most critical block.
    list<virtualRegister> vregs = getVREGs( b );

    // Sort VREGs by #occurrences, apply standard graph coloring.
    vregs.sort( occurrences of VREG in b );
    traditionalGraphColoring( P, vregs );
  }

  // Allocate all remaining VREGs not lying on the WCEP.
  traditionalGraphColoring( P, getVREGs( P ) );
  return P;
}
```

# Properties (1)

- Algorithm simultaneously handles a program *P* to be register allocated, and a copy *P'* where all VREGs are completely spilled to memory in order to enable WCET analysis.
- Register allocation is done basic block-wise along the WCEP.
- After the allocation of one basic block, the WCEP in *P'* is recomputed.
- In one iteration of the algorithm: Allocation of all VREGs occurring in that basic block *b* that contains many spill instructions in *P'* and that is executed very often.

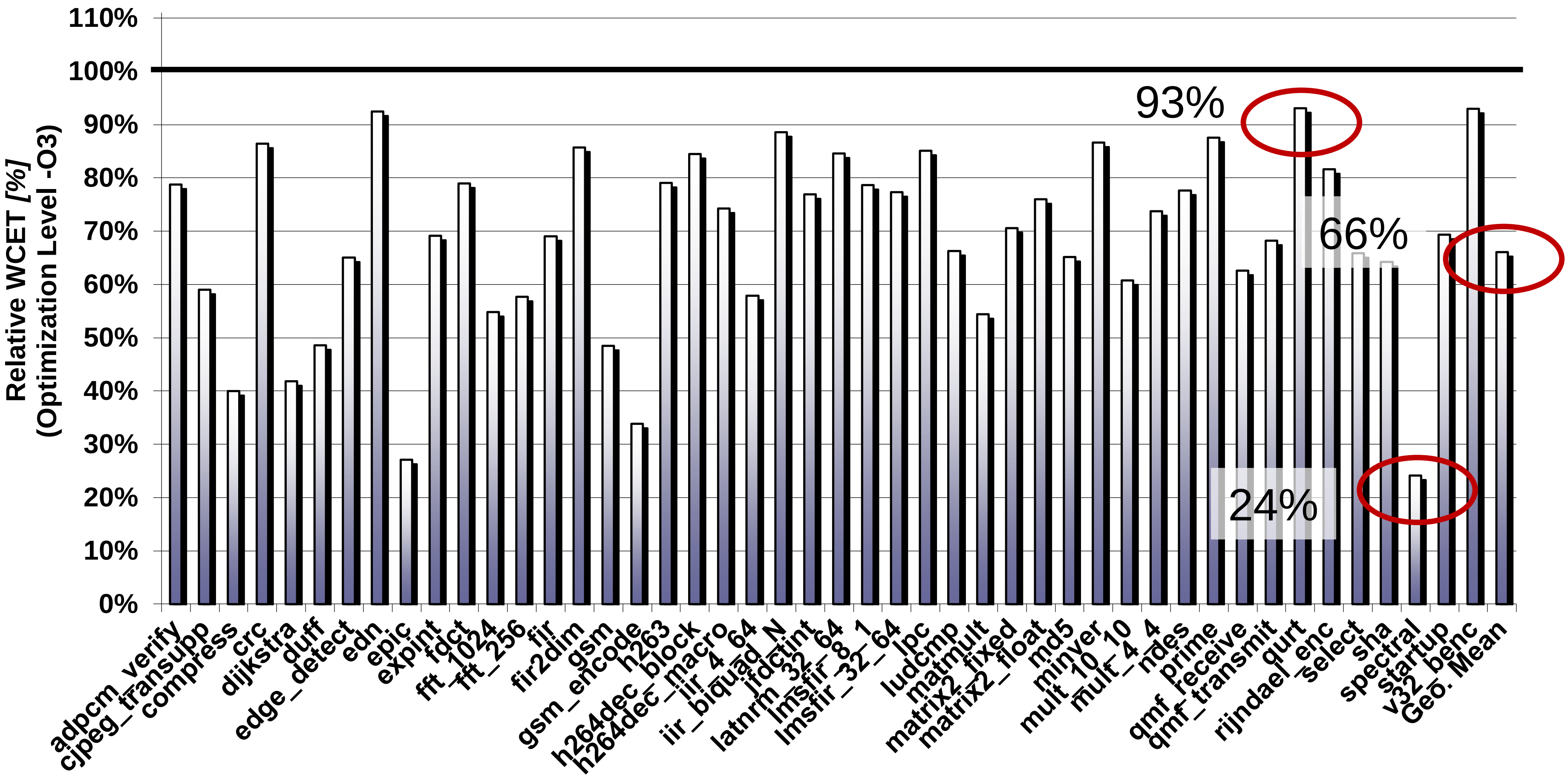
☞ The VREGs of this most timing-critical block *b* should be kept in PHREGs if possible.

# Properties (2)

– Spilling of VREGs in *b* cannot be avoided in general. If spilling is required in *b*, spill only those VREGs *v* of *b* that occur least frequently in *b*, since few occurrences of *v* in *b* imply few spill instructions inside *b*.
– Register allocation itself, i.e., assigning colors to *b*'s VREGs, and spill code generation are handed over to standard graph coloring.
– After the allocation loop, the WCEP is completely allocated. But there may still be some VREGs in blocks besides the WCEP.
☞ One final run of standard graph coloring in order to catch all those remaining VREGs.

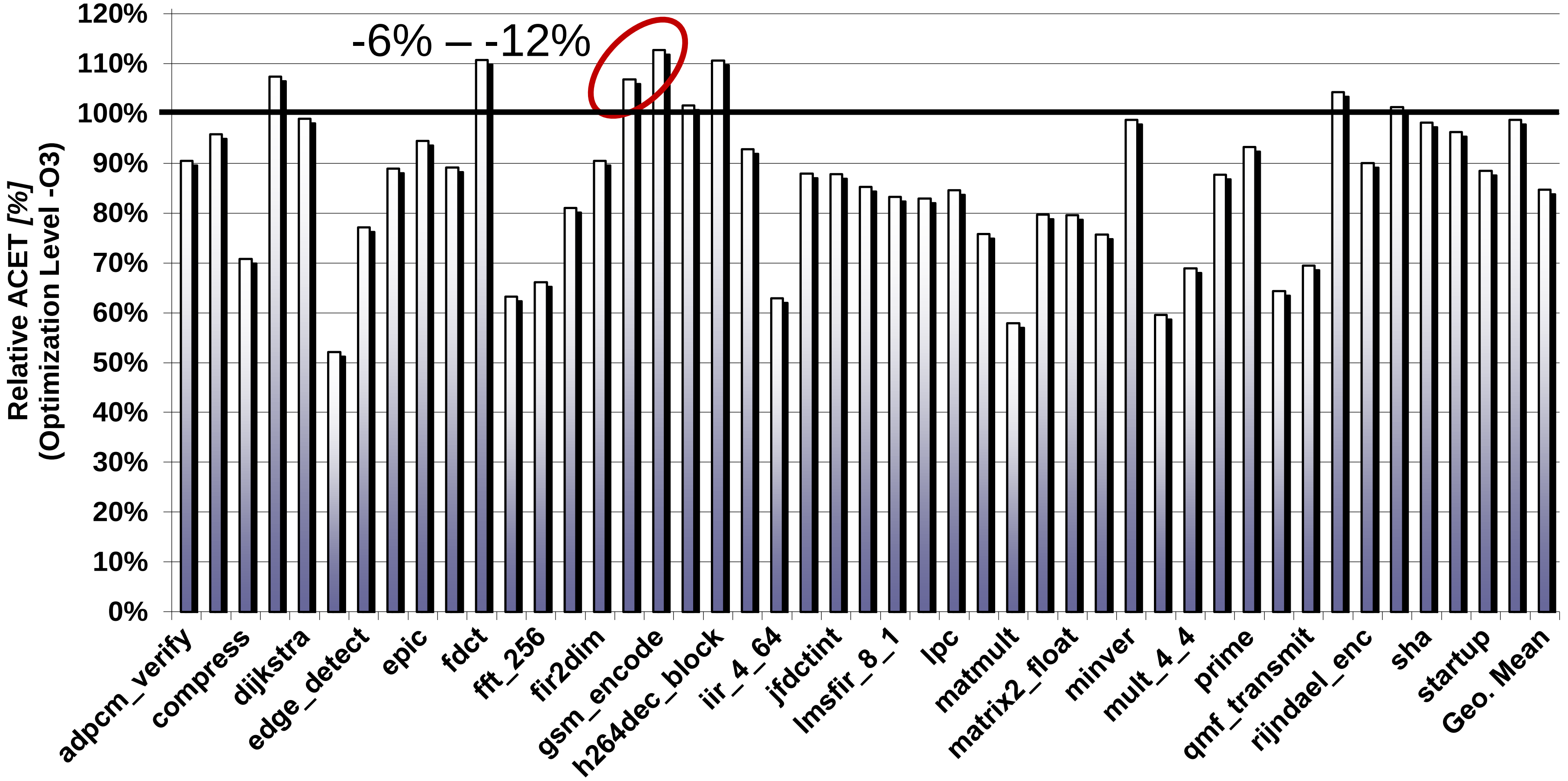
*[**H. Falk.** WCET-aware Register Allocation based on Graph Coloring. DAC, San Francisco, 2009]*

# Relative $WCET_{EST}$ after WCET-aware Graph Coloring




100% = $WCET_{EST}$ using Standard Graph Coloring (highest degree)

# Relative ACET after WCET-aware Graph Coloring




100% = ACET using Standard Graph Coloring (highest degree)

# Discussion

- $WCET_{EST}$ reductions from 6.9% up to 75.9%, on average 33.9%.
- Allocation of all 46 benchmarks: total of 1,979 WCET analyses.
- Run-time of WCET graph coloring: 12:15 hours for all 46 benchmarks
  ☞ 16 minutes per benchmark on average
- ACET reductions of up to 47.9%, but decreases of up to 12.7%. On average, 15.2% ACET reduction.
- Benchmarks behave very different w.r.t. $WCET_{EST}$ and ACET:
  **`gsm`** family: 51.5% – 66.2% $WCET_{EST}$ reduction
  6.8% – 12.7% ACET degradation
- *Reason:* WCET graph coloring avoids spilling along WCEP but inserts spill code at other places in the CFG which are frequently executed in an average-case scenario.

# Conclusions

**Comparison with Consequences for $\text{WCET}_{\text{EST}}$ Optimizations**

WCET-aware optimizations...

– ... mandatorily need detailed knowledge of the WCEP

👍 *WCET graph coloring considers the WCEP*

– ... always have to be aware that the WCEP can change after each individual optimization decision

👍 *WCET graph coloring updates the WCEP after each modification of the code*

– ... should take the decision where to optimize something not only based on local information, but should always consider the global effects of an optimization decision

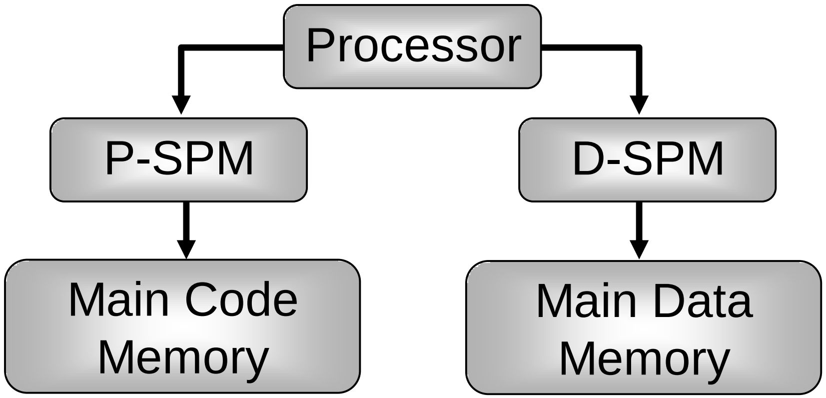
👎 *WCET graph coloring is greedy heuristic that is driven solely by local data per basic block*

# Chapter Contents

## 9. WCET-Aware Compilation

- Introduction
  - Integration of a WCET Timing Model into a Compiler
  - Challenges for WCET-Aware Optimization
- Procedure Cloning & Positioning
  - WCET-Aware Procedure Cloning
  - Procedure Positioning for Cache Miss Reduction
- Register Allocation
  - Problem of Classical Graph Coloring
  - WCET-Aware Graph Coloring
- Scratchpad Allocation of Data and Code
  - Allocation of global Data
  - Allocation of Basic Blocks

# In the Following: Harvard Architectures




- Separate busses and memories for code and data
- Scratchpad allocation of code and data can be solved independently from each other

☞ Two separate ILPs for these optimizations

☞ Non-Harvard architectures with unified busses and memories for code and data: Straightforward combination of both ILPs

# ILP for WCET-aware SPM Allocation of Data

**Goal**

- Determine set of data objects (global variables or static local variables) to be allocated to the data SPM,
- such that selected data objects lead to overall minimization of $WCET_{EST}$
- under consideration of switching WCEPs.

**Approach**

- Integer-linear programming (ILP)

☞ Optimality of results

- *Notation:* Upper-case letters ≅ constants,
  lower-case letters ≅ variables

# Decision Variables & Costs

- ***Binary decision variables per data object:***

$$y_i = \begin{cases} 1 & \text{if data object } d_i \text{ is assigned to } mem_{spm} \\ 0 & \text{if data object } d_i \text{ is assigned to } mem_{main} \end{cases}$$

- ***Costs of basic block $b_j$:***

$$c_j = C_j - \sum_{d_i \in \text{data objects}} G_{i,j} * y_i$$

$c_j$ models the $\text{WCET}_{\text{EST}}$ of $b_j$, depending on whether the data objects accessed by $b_j$ are allocated to main memory or SPM, resp.
$C_j$: $b_j$'s $\text{WCET}_{\text{EST}}$ if all data objects reside in main memory
$G_{i,j}$: WCET reduction of $b_j$ if data object $d_i$ is assigned to SPM

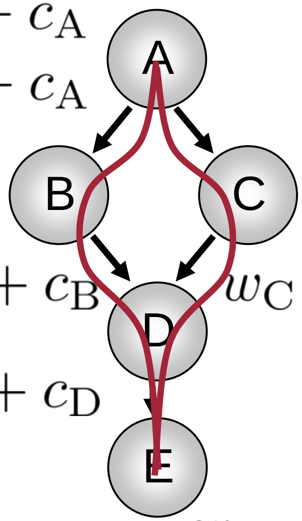
# Intraprocedural Control Flow (1)

*[**V. Suhendra et al.** WCET Centric Data Allocation to Scratchpad Memory. RTSS, Miami, 2005]*

– ***Modeling of a function's control flow:***

<u>*Acyclic sub-graphs:*</u>

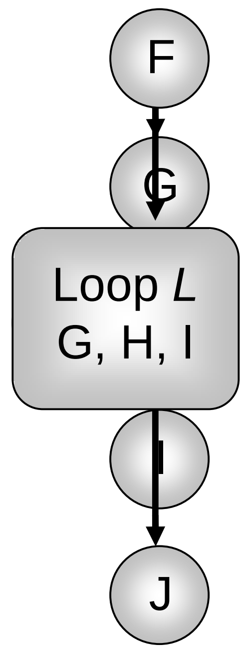
$$w_A \geq w_B + c_A$$
$$w_A \geq w_C + c_A$$




$$w_B \geq w_D + c_B \qquad w_C \geq w_D + c_C$$
$$w_D \geq w_E + c_D$$
$$w_E = c_E$$

$w_A$ = WCET of <u>*longest*</u> path starting at A

<u>*(Reducible) Loops:*</u>




- Treat body of inner-most loop $L$ like acyclic sub-graph
- Fold loop $L$
- Costs of $L$:

  $c_L = w_G * C^L_{max}$
- Continue with next innermost loop

# Intraprocedural Control Flow (2)

- ***Modeling of a function's control flow:***
  - For sink nodes $b_j$ of an acyclic sub-graph, $w_j$ is set to the costs $c_j$
  - For all other nodes $b_j$ of an acyclic sub-graph, the WCET of the paths starting in $b_j$ must be greater or equal than the WCET of each successor $b_{succ}$, PLUS the costs $c_j$ of $b_j$
  - For each successor $b_{succ}$ of $b_j$, one constraint is created in the ILP
  - ☞ Variable $w_j$ actually models all paths starting in $b_j$. Due to the ≥ operator in the inequations, the maximum over all paths starting in $w_j$ is modeled
  - 👍 Potential changes of the WCEP from one successor $b_{succ1}$ to another successor $b_{succ2}$ of $b_j$ are considered by construction

# Intraprocedural Control Flow (3)

- ***Modeling of a function's control flow:***
  - Reducible loops *L* have exactly one entry basic block $b^L_{entry}$
  - By "ignoring" of the back-edge of a reducible loop *L*, the CFG of the loop body becomes acyclic
  - Create constraints for acyclic loop body as shown on slide 65 (left part)
  - ☞ Variable $w^L_{entry}$ models $WCET_{EST}$ of the entire body of loop *L* if it is executed *exactly once*
  - Multiplication of $w^L_{entry}$ by the maximal number $C^L_{max}$ of iterations of *L* provides $WCET_{EST}$ for *all executions* of the loop
  - ☞ Costs of *L* are equal to the $WCET_{EST}$ of *L* for all executions

# Interprocedural Control Flow

- ***Modeling function calls:***
  - Each function $F$ has dedicated entry BB $b^F_{entry}$
  - Variable $w^F_{entry}$ models $\text{WCET}_{\text{EST}}$ of the longest path in $F$ that starts in $b^F_{entry}$
  - ☞ $w^F_{entry}$ models $\text{WCET}_{\text{EST}}$ of $F$ for exactly 1 execution of $F$
  - ☞ If $F'$ calls function $F$: Add $w^F_{entry}$ to $\text{WCET}_{\text{EST}}$ of $F'$

- ***Function calls in basic block $b_j$:***
  - "Call penalty" for calling basic block:

$$cp_j = \begin{cases} w^F_{entry} & \text{if } b_j \text{ calls } F \\ 0 & \text{else} \end{cases}$$

  - ILP constraint per basic block:

$$\forall (b_j, b_{succ}) : w_j \geq w_{succ} + c_j + cp_j$$

# Scratchpad Capacity and Objective Function

- ***Scratchpad capacity constraint:***

$$\sum_{d_i \in \text{data objects}} (S_i * y_i) \leq S_{spm}$$

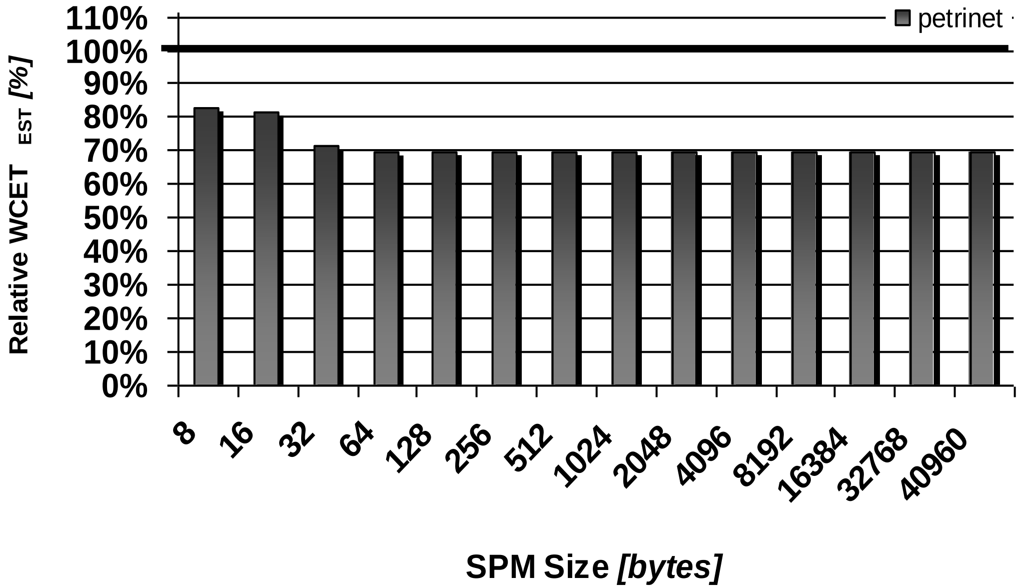
The sum of the sizes of all data objects allocated onto the SPM is less than or equal to the totally available SPM capacity.

- ***Objective function:***
  - $w^F_{entry}$ models $\text{WCET}_{\text{EST}}$ of $F$ if $F$ is executed exactly once
  - Variable $w^{\texttt{main}}_{entry}$ models $\text{WCET}_{\text{EST}}$ of the entire program

☞ $w^{\texttt{main}}_{entry} \rightsquigarrow min.$

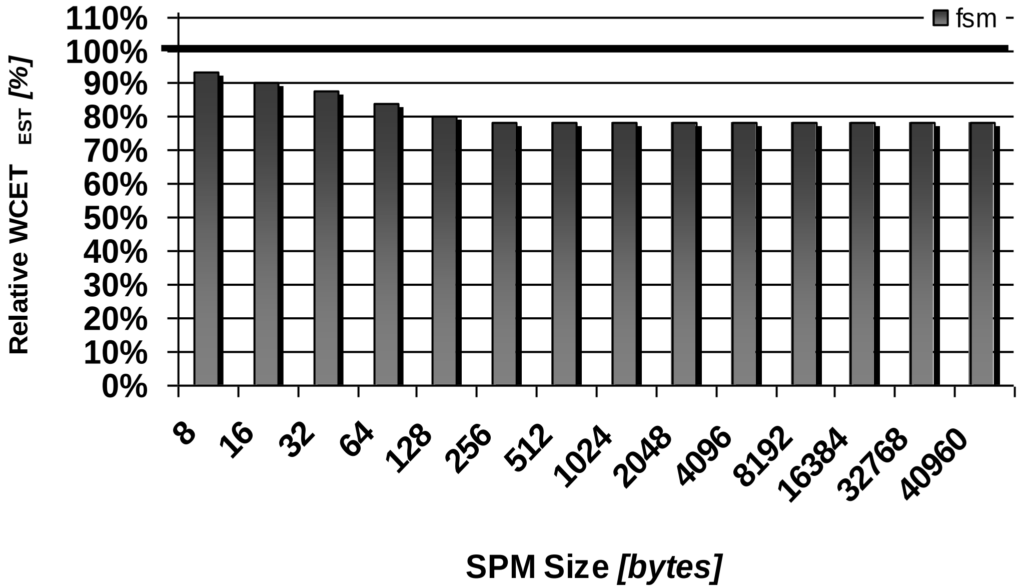
*[**F. Rotthowe.** Scratchpad Allocation of Data for Worst-Case Execution Time Minimization (in German). Diploma Thesis, TU Dortmund, 2008]*

# Rel. $WCET_{EST}$ after D-SPM Allocation of `petrinet`



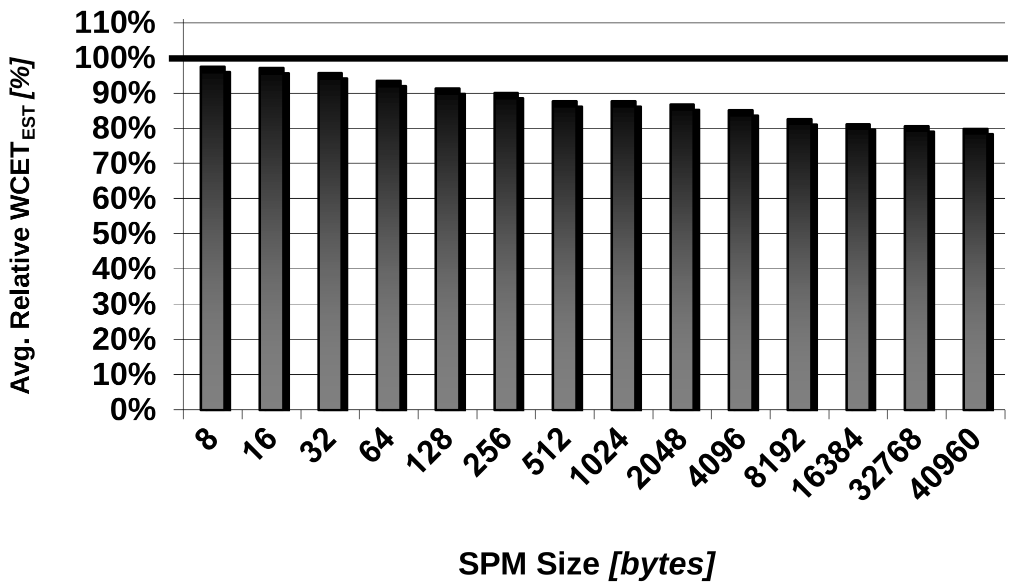

- Notable $WCET_{EST}$ reductions already for SPMs of only a few bytes
- 6 global variables of 72 bytes size in total
- $WCET_{EST}$ reductions by 28.6% for 32 bytes SPM
- X-Axis: Absolute SPM sizes
- Y-Axis: 100% = $WCET_{EST}$ when not using SPM at all

# Rel. $WCET_{EST}$ after D-SPM Allocation of `fsm`




- More steady $WCET_{EST}$ reductions for increasing SPM sizes
- 98 global variables à 4 bytes size each
- $WCET_{EST}$ reductions by 21.4% for 256 bytes SPM
- X-Axis: Absolute SPM sizes
- Y-Axis: 100% = $WCET_{EST}$ when not using SPM at all

# Rel. $WCET_{EST}$ after D-SPM Allocation of 14 Benchmarks




- Steady $WCET_{EST}$ reductions for increasing SPM sizes
- $WCET_{EST}$ reductions from 2.7% – 20.6%
- X-Axis: Absolute SPM sizes
- Y-Axis: 100% = $WCET_{EST}$ when not using SPM at all

# ILP for WCET-aware SPM Allocation of Code

**Goal**

- Determine set of basic blocks to be allocated to the SPM
- such that selected basic blocks lead to overall minimization of $WCET_{EST}$
- under consideration of switching WCEPs.

**Approach**

- Integer-linear programming (ILP)

☞ Optimality of results

- *Notation:* Upper-case letters ≅ constants,
  lower-case letters ≅ variables

# Decision Variables & Costs

– ***Binary decision variables per basic block:***

$$x_i = \begin{cases} 1 & \text{if basic block } b_i \text{ is assigned to } mem_{spm} \\ 0 & \text{if basic block } b_i \text{ is assigned to } mem_{main} \end{cases}$$

– ***Costs of basic block $b_i$:***

$$c_i = C^i_{main} * (1 - x_i) + C^i_{spm} * x_i$$

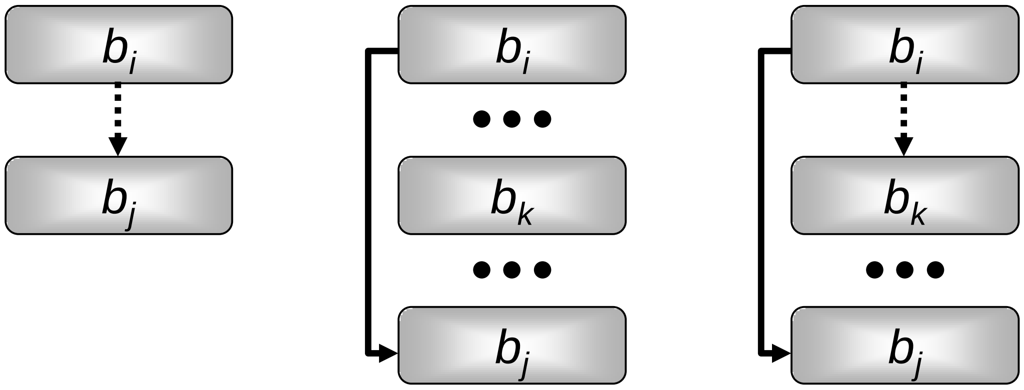
$c_i$ models the $\text{WCET}_{\text{EST}}$ of $b_i$ if it is allocated to main memory or SPM, resp.

– ***Modeling of the intraprocedural control flow:***
As before in the WCET-aware SPM allocation of data

# Cross-Memory Jumps

- ***Allocation of consecutive basic blocks:***
  - Allocation of consecutive basic blocks in the CFG to different memories requires adaption/insertion of dedicated jumping code
  - Cross-memory jumps are costly: Often need more than 1 instruction
  - Jumping code: *Variable* overhead in terms of $WCET_{EST}$ and code size, depending on decision variables *(☞ cf. chapter 7)*

- ***Jump Scenarios:***



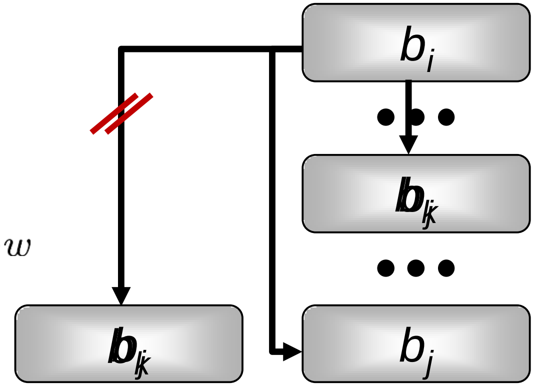

***a) Implicit*** ***b) Unconditional*** ***c) Conditional***

# Penalties for Cross-Memory Jumps

- ***Jump Penalty (⊗ ≅ Boolean XOR):***
  - *<u>Penalty for implicit jumps:</u>* $jp^i_{impl}$

    $$jp^i_{impl} = (x_i \otimes x_j) * P_{high}$$

    High penalty if basic blocks *i* and *j* are placed in different memories
  - *<u>Penalty for unconditional jumps:</u>* $jp^i_{uncond}$
    - If $b_i$ and $b_j$ in different memories: $P_{high}$
    - If $b_i$ and $b_j$ adjacent in same memory: 0
    - If $b_i$ and $b_j$ not adjacent in same memory: $P_{low}$




$$jp^i_{uncond} = (x_i \otimes x_j) * P_{high} + \overline{(x_i \otimes x_j)} * (1 - \prod_{b_k \in \text{JS b)}} (x_i \otimes x_k)) * P_{low}$$

  - *<u>Conditional jumps:</u>* Obvious combination of $jp^i_{impl}$ and $jp^i_{uncond}$

# Jump Penalties and Interprocedural Control Flow

– ***Jump penalties for basic block $b_i$:***

$$jp_i = \begin{cases} jp^i_{impl} & \text{if Jump Scenario of } b_i \text{ is } \mathit{implicit} \\ jp^i_{uncond} & \text{if Jump Scenario of } b_i \text{ is } \mathit{unconditional} \\ jp^i_{cond} & \text{if Jump Scenario of } b_i \text{ is } \mathit{conditional} \\ 0 & \text{else} \end{cases}$$

– ***Penalty for function calls for basic block $b_i$:***

$$cp_i = \begin{cases} w^F_{entry} + (x_i \otimes x^F_{entry}) * P_{high} & \text{if } b_i \text{ calls } F \\ \qquad +(1 - (x_i \otimes x^F_{entry})) * P_{low} & \\ 0 & \text{else} \end{cases}$$

If block $b_i$ calls function $F$: Add $\text{WCET}_{\text{EST}}$ of $F$ to $\text{WCET}_{\text{EST}}$ of $b_i$. Furthermore, add $P_{high}$ if function call is cross-memory call. If function call stays in the same memory, add only $P_{low}$.

# Sizes of Basic Blocks

- ***Constraint for all successors $b_{succ}$ of $b_i$:***

  $$\forall(b_i, b_{succ}) : w_i \geq w_{succ} + c_i + cp_i + jp_i$$

- ***Size of a basic block $b_i$:***
  - Size of $b_i$ depends on actual jumping code for $b_i$
  - Size of jumping code of $b_i$ depends on jump scenario:

  $$s_i = \begin{cases} (x_i \wedge \overline{x_j}) * S_{impl} & \text{if Jump Scenario of } b_i \text{ is } \mathit{implicit} \\ (x_i \wedge \overline{x_j}) * S_{uncond} & \text{if Jump Scenario of } b_i \text{ is } \mathit{unconditional} \\ (x_i \wedge \overline{x_k}) * S_{impl} + & \text{if Jump Scenario of } b_i \text{ is } \mathit{conditional} \\ \qquad (x_i \wedge \overline{x_j}) * S_{uncond} & \\ (x_i \wedge \overline{x^F_{entry}}) * S_{call} & \text{if } b_i \text{ calls } F \\ 0 & \text{else} \end{cases}$$

  - Total size of basic block $b_i$:
    Size $S_i$ of $b_i$ without any jumping code <u>*plus*</u>
    Size $s_i$ of $b_i$'s jumping code

# Scratchpad Capacity and Objective Function

- ***Scratchpad capacity constraint:***

  $$\sum_{b_i} (S_i * x_i + s_i) \leq S_{spm}$$

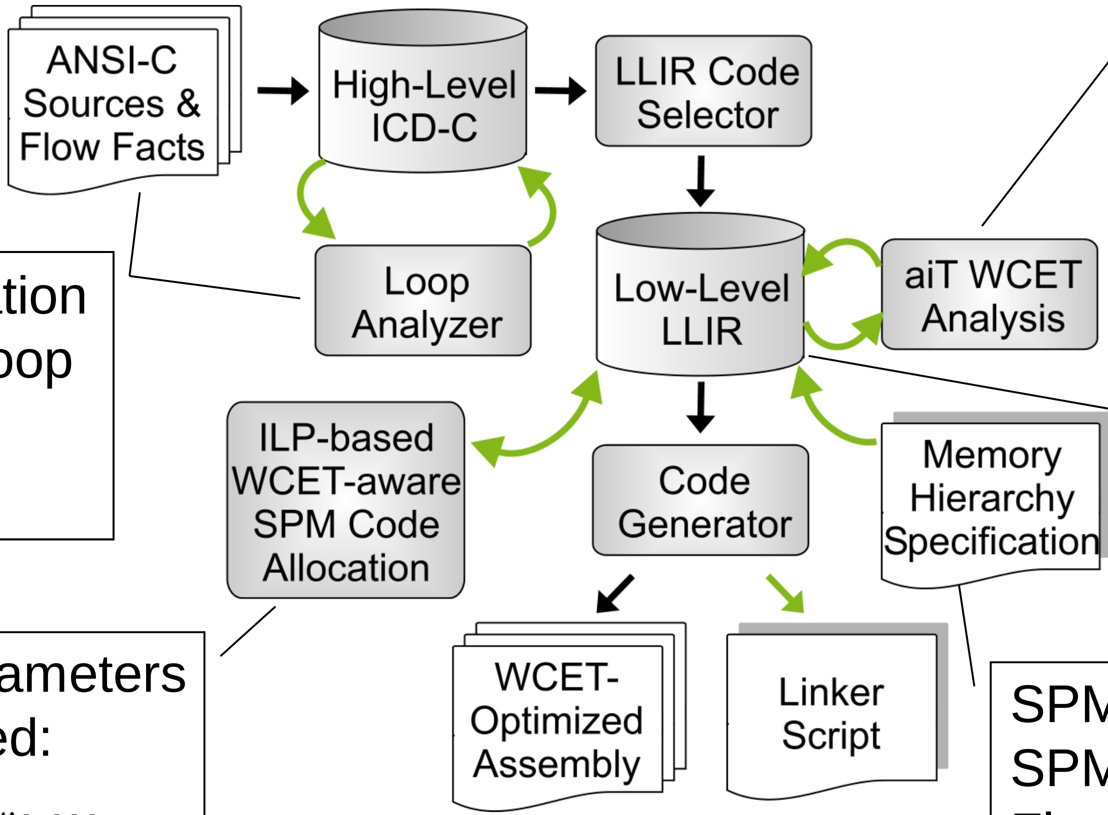
  The sum of the sizes of all basic blocks allocated onto the SPM without jumping code, plus the size of jumping code in $b_i$ is less than or equal to the totally available SPM capacity.

- ***Objective function:***

  $$w_{entry}^{\texttt{main}} \rightsquigarrow min.$$

*[**H. Falk, J. C. Kleinsorge.** Optimal Static WCET-aware Scratchpad Allocation of Program Code. DAC, San Francisco, 2009]*

# Determination of the Constants of the ILPs (1)




WCET$_{EST}$ of BB $b_i$ for SPM and main memory: $C^i_{main}$, $C^i_{spm}$

Max. iteration count of loop $L$: $C^L_{max}$

Size of BB $b_i$: $S_i$

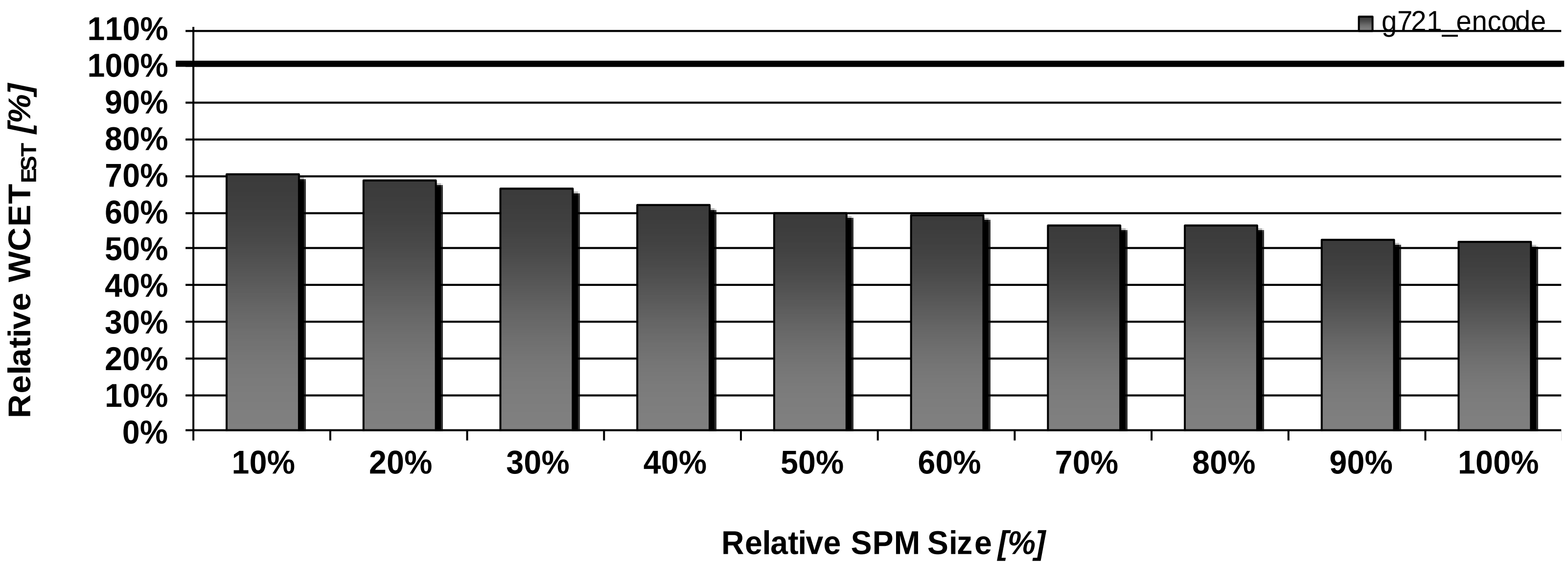
Other parameters hard-coded: $P_{high}, P_{low}, \ldots$

SPM size $S_{spm}$ = 47 kB
SPM access = 1 cycle
Flash access = 6 cycles

# Determination of the Constants of the ILPs (2)

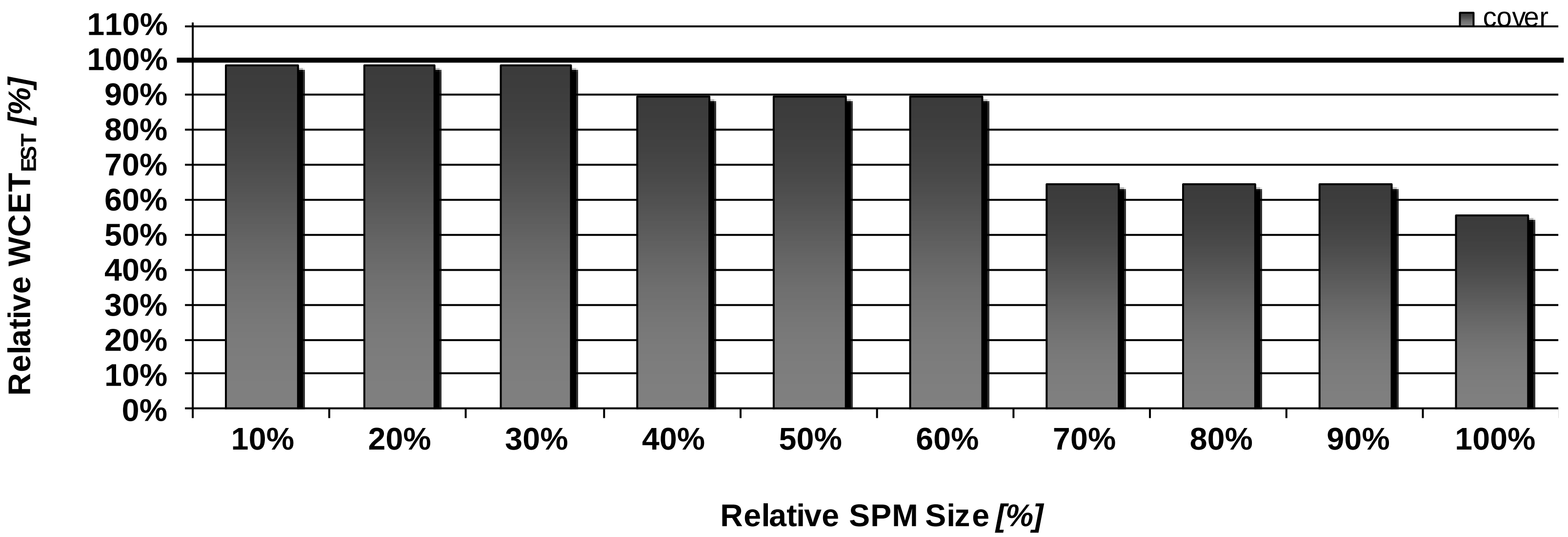
- *WCET$_{EST}$ $C^i_{main}$, $C^i_{spm}$ per basic block $b_i$ for both memories:*
  Determined by two WCET analyses, one in which all basic blocks lie in the SPM, one with all blocks in main memory.
- *Max. iteration count of loops $C^L_{max}$:*
  Either annotated in the source code using flow facts, or determined by WCC's automatic loop bound analysis.
- *Size $S_i$ of a basic block without jumping code:*
  By simple enumeration of all LIR operations
- *Size $S_{spm}$ of the scratchpad:*
  Taken from WCC's memory hierarchy specifications
- *Remaining parameters determined experimentally:*
  $P_{high}$ = 16 $P_{low}$ = 8
  $S_{impl}$ = $S_{uncond}$ = 10 $S_{call}$ = 12

# Rel. $WCET_{EST}$ after P-SPM Allocation of `g721_encode`



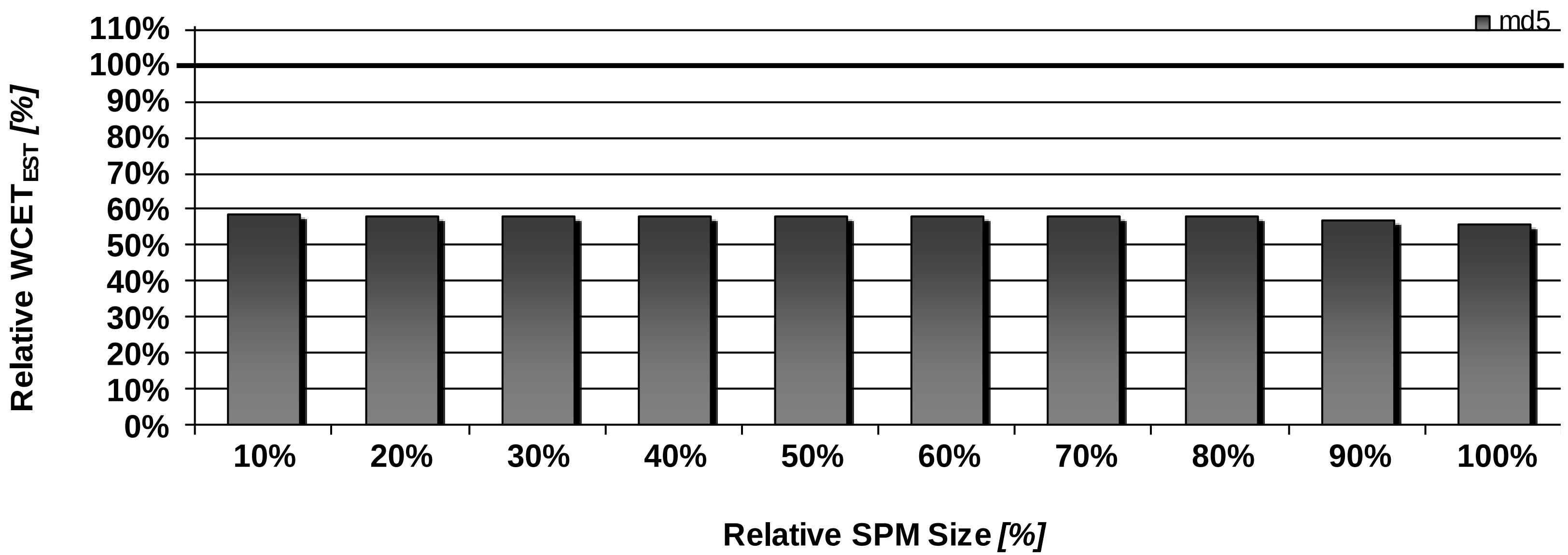

- Steady $WCET_{EST}$ decreases for increasing SPM sizes
- $WCET_{EST}$ reductions from 29% – 48%
- X-Axis: SPM size = *x*% of benchmark's code size
- Y-Axis: 100% = $WCET_{EST}$ when not using SPM at all

# Rel. $WCET_{EST}$ after P-SPM Allocation of `cover`




- Stepwise $WCET_{EST}$ decreases: Useful content allocated to SPM only at 40%, 70% and 100% relative SPM size
- $WCET_{EST}$ reductions of 10%, 35% and 44%
- X-Axis: SPM size = *x*% of benchmark's code size
- Y-Axis: 100% = $WCET_{EST}$ when not using SPM at all

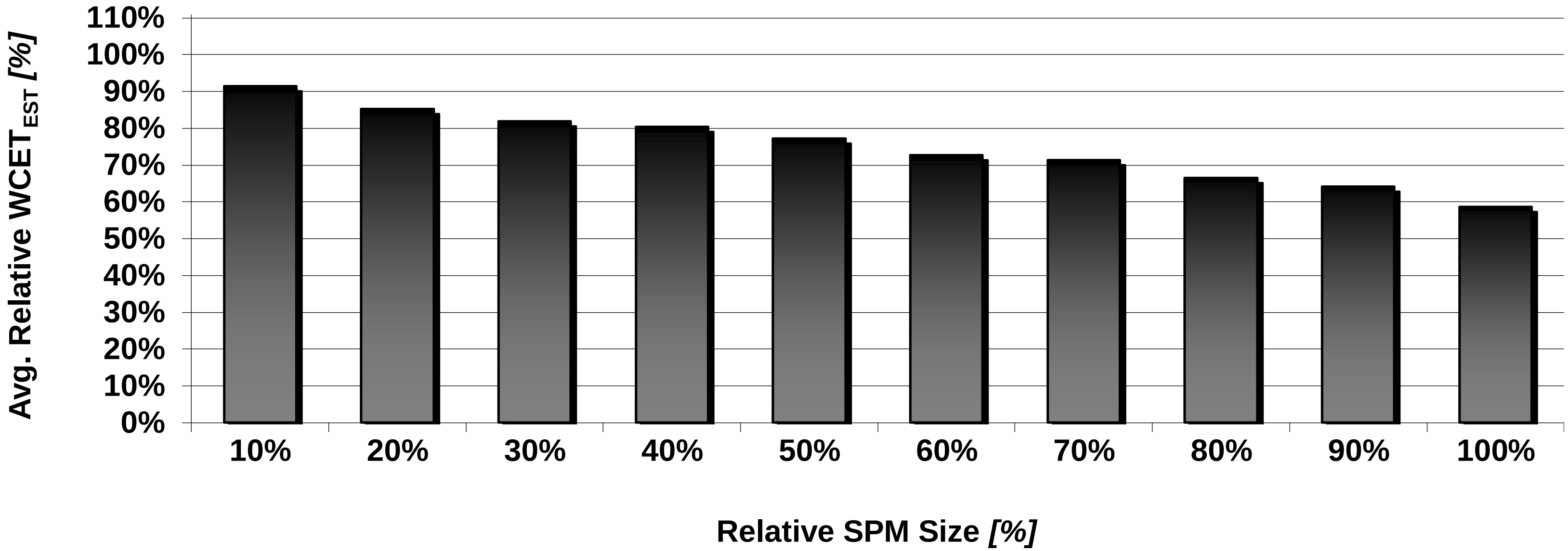
# Rel. $WCET_{EST}$ after P-SPM Allocation of `md5`




- Almost invariable $WCET_{EST}$ reductions for all SPM sizes: 40% - 44%
- ILP clearly finds tiny but time-critical hot-spot of **`md5`** and allocates it to SPM

- X-Axis: SPM size = *x*% of benchmark’s code size
- Y-Axis: 100% = $WCET_{EST}$ when not using SPM at all

# Rel. $WCET_{EST}$ after P-SPM Allocation of 73 Benchmarks




- Steady $WCET_{EST}$ reductions for increasing SPM sizes
- $WCET_{EST}$ reductions from 8% – 41%

- X-Axis: SPM size = *x*% of benchmark's code size
- Y-Axis: 100% = $WCET_{EST}$ when not using SPM at all

# Conclusions

**Comparison with Consequences for $\text{WCET}_{\text{EST}}$ Optimizations**

WCET-aware optimizations...

– ... mandatorily need detailed knowledge of the WCEP

👍 *SPM allocations consider the WCEP*

– ... always have to be aware that the WCEP can change after each individual optimization decision

👍 *SPM allocations inherently capture WCEP changes inside the ILP*

– ... should take the decision where to optimize something not only based on local information, but should always consider the global effects of an optimization decision

👍 *Objective functions of the ILPs model the global WCET of a program that is subject to minimization.*

# References

**WCC Compiler Infrastructure**

– H. Falk, P. Lokuciejewski. *A compiler framework for the reduction of worst-case execution times*. Springer Real-Time Systems 46(2), October 2010.

# Summary

**Compilers for $\text{WCET}_{\text{EST}}$ Minimization**

– Integration of a formal WCET timing model into compiler
– Challenge: To consider unstable WCEPs in the course of optimizations

**WCET-aware Optimizations**

– Procedure Cloning & Positioning: Greedy heuristics that determine current WCEP via repeated WCET analyses
– Register allocation: cyclic dependencies between register allocation and WCET analysis; graph coloring along the always current WCEP
– Scratchpad allocations: Inherent modelling of WCEP in the ILPs; eliminates need for repeated WCET analyses during optimization