Topic 2
WCET-aware compilation and optimization

Heiko Falk

Institute for Embedded Systems/Real-Time Systems
Faculty of Engineering and Computer Science
Ulm University
Outline

1. Introduction and Motivation
   - Hard Real-Time Systems and the Worst-Case Execution Time
   - Current State of the Art in SW-Design for Hard RT-Systems
   - Static WCET Analysis
   - The WCET-aware C Compiler WCC
   - Challenges for WCET-aware Compiler Optimizations

2. Procedure Positioning

3. Scratchpad Allocation
Hard Real-Time Systems

Real-Time Systems
– Computation of a function \( f(x) \) must complete within given time bounds
– Too late computation of \( f(x) \Rightarrow \text{incorrect computation of } f(x) \)

Hard Real-Time Systems
– Late computation of \( f(x) \Rightarrow \text{catastrophic consequences} \)
  (loss of human life, environmental damages, ...)
– Examples: Electronics in transportation systems, controllers in power plants, medical systems, ...
– Real-Time Systems are usually also Embedded Systems

Worst-Case Execution Times (\textit{WCET})

**Requirements**
- **Safeness:** $\text{WCET} \leq \text{WCET}_{\text{EST}}$
- **Tightness:** $\text{WCET}_{\text{EST}} - \text{WECT} \rightarrow \text{minimal}$

\textit{WCET in general not computable}

[Borrowed from Reinhard Wilhelm]
Software Development Flow for Hard Real-Time Systems

Current Industrial Practice (Automotive, Avionics)
1. Specification using graphical / high-level tools
2. Automatic generation of ANSI-C code
3. Translation into executable 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: goto 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 Design

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 WCETs possible

Optimization Strategy
- Naive: Current compilers lack precise ACET timing model
- Application of an optimization if “promising”
  Effect of optimizations on a program’s ACET fully unknown to the compiler itself.
Motivation of this Summer School’s Topic

Design of a Compiler that
– considers $\text{WCET}_{\text{EST}}$ 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 $\text{WCET}_{\text{EST}}$.

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

- WCET in general not computable
- Derivation of safe upper bounds ($\text{WCET}_{\text{EST}}$) possible for particular classes of programs if *Flow Facts* are provided

- AbsInt’s WCET Analyzer aiT
- Safe static WCET estimation

[http://www.absint.com/ait]
Workflow of the WCET Analyzer aiT (1)

- **Input**: Binary executable $P$ to be analyzed.
- **exec2crl**: Disassembler, translates $P$ into aiT’s intermediate format CRL2.
- **Value analysis**: Computes possible contents of processor registers for any point in time during $P$’s execution.

**Note**: $P$ is never executed by aiT! $P$ is “only” analyzed.
Workflow of the WCET Analyzer aiT (2)

- **Loop bound analysis**: Tries to determine lower and upper bounds for the number of iterations of each loop in $P$.
- **Cache analysis**: Makes use of a formal cache model, classifies each memory access in $P$ as definite cache hit, definite cache miss or unknown.
Workflow of the WCET Analyzer aiT (3)

- **Pipeline analysis**: Includes an accurate model of the processor’s pipeline. Depending on the pipeline’s initial state, possible cache states etc., possible states of the pipeline at the end of each basic block of $P$ are determined. Result of the pipeline analysis is the $\text{WCET}_{\text{EST}}$ of each individual basic block.
Workflow of the WCET Analyzer aiT (4)

- **Path analysis**: Models all possible execution paths within $P$ under consideration of the $\text{WCET}_{\text{EST}}$ of all basic blocks. Determines the longest possible execution path within $P$ which leads to the overall $\text{WCET}_{\text{EST}}$ of $P$. Result of the path analysis is e.g. the length of this longest path, i.e. $P$‘s $\text{WCET}_{\text{EST}}$. 
Complexity Issues (1)

- **Problem**: WCET analysis is not computable with today’s machines! If WCET were computable, one could decide in $O(1)$ if WCET $< \infty$ and thus solve the halting problem.

- **Reason**: It is not computable how long $P$ stays in loops. Automatic loop bound analysis is applicable only for simple classes of loops. (Analogously for recursive function calls.)
Complexity Issues (2)

- **Solution**: The user of aiT must mandatorily provide information about e.g. minimal and maximal iteration bounds of loops and recursion depths.

- **Annotation file**: Contains such user-provided annotations (“Flow Facts”) and is – besides the program \( P \) to be analyzed – another external input to aiT.
Control Flow Graphs

Control Flow Graph (CFG)
- Fundamental code representation during path analysis

**Definition:** \( CFG = (V, E, s) \) is a directed graph with
- \( V = \{ b_1, \ldots, b_n \} \) (set of all basic blocks)
- \( E = \{ (b_i \rightarrow b_j) | \text{basic block } b_j \text{ can be executed immediately after } b_i \} \)
- \( s \in V = \) the unique start node of the CFG

```c
int fun2() {
  for (; a1<30; a1++ ) {
    for (; b1<a1; b1++ )
      printf( "%d
", b1 );
    printf( "%d
", a1 );
  }
  return a1; }
```
Path Analysis using Implicit Path Enumeration

- **Question:** Given a CFG $G$ and the $WCET_{EST}$ per node, which is the longest path from $s$ to an end node in $G$ w.r.t. $WCET_{EST}$?

- **Approach:** Use an ILP to model $G$ where the objective function to be maximized represents the length of the longest path. This ILP has to reflect all possible paths in $G$. It does so *not* by *explicitly* listing all such paths. Instead, they are represented *implicitly*.

- **Implicit Path Enumeration Technique (IPET):** Use integer variables per node and edge in $G$. A variable represents the maximal possible flow through an edge or node. Constraints ensure that for each node $v$, the flow entering $v$ is equal to the flow leaving $v$ (*Kirchhoff’s rule*). Objective function models maximal flow through $G$’s nodes, weighted by the $WCET_{EST}$ per node.
IPET – Example (1)

```c
int main()
{
    int i, j = 0;

    _Pragma( "loopbound min 100 max 100" );
    for ( i = 0; i < 100; i++ )
        if ( i < 50 )
            j += i;
        else
            j += (i * 13) % 42;
    return j;
}
```

CFG and node WCETs:
IPET – Example (2)

ILP constraints:

/* The program is executed exactly once */
xm = 1;
/* Constraint for flow leaving _main */
xm = xm1;
/* Constraint for flow entering _L1 */
x1 = xm1 + x61;
/* Constraint for flow leaving _L1 */
x1 = x12 + x13;
/* Constraint for flow entering _L2 */
x2 = x12;
/* Constraint for flow entering _L3 */
x3 = x13;
...
IPET – Example (3)

ILP constraints:

...  
/* Constraint for lower loop bound of _L1 */  
\textit{x1} \geq 101 \textit{x12}; 
/* Constraint for upper loop bound of _L1 */  
\textit{x1} \leq 101 \textit{x12}; 

/* Objective function = WCET \rightarrow \text{max.} */ 
\textbf{MAX}: 21 \textit{xm} + 27 \textit{x1} + 20 \textit{x2} + 2 \textit{x3} +  
2 \textit{x4} + 20 \textit{x5} + 13 \textit{x6};
IPET – Example (4)

ILP solution:

Value of objective function: 6268

Actual values of the variables:

\[
\begin{align*}
\text{xm} & : 1 \\
\text{x1} & : 101 \\
\text{x2} & : 1 \\
\text{x3} & : 100 \\
\text{x4} & : 0 \\
\text{x5} & : 100 \\
\text{x6} & : 100 \\
\text{xm1} & : 1 \\
\text{x61} & : 100 \\
\text{x12} & : 1 \\
\text{x13} & : 100 \\
\text{x34} & : 0 \\
\text{x35} & : 100 \\
\text{x46} & : 0 \\
\text{x56} & : 100 \\
\end{align*}
\]

Program’s $WCET_{EST}$

Worst-Case Execution Count of Basic Blocks ($WCEC$)

Worst-case flow through CFG edges
Integration of WCET into WCC Compiler (1)

- Supported processors: Infineon TriCore TC1796 and TC1797; ARM7
- Tight integration of aiT
- 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 into WCC Compiler (2)

Relevant WCET data:
- $WCET_{EST}$ of entire program, function of basic block
- Worst-Case execution frequency per function, basic block or CFG edge
- Potential register contents
- Cache hits / misses per basic block

Issues during WCET\textsubscript{EST} Minimization

The Worst-Case Execution Path (WCEP)
- WCET of a program = Length of the program’s longest execution path (WCEP)
- WCET\textsubscript{EST} Minimization: Optimization of only those parts of a program lying on the WCEP
- Code optimization apart the WCEP will not reduce WCET\textsubscript{EST}

Optimizerizations minimizing WCET\textsubscript{EST} require detailed knowledge of the WCEP...

\textbf{WCET analyzer aiT provides such detailed information by means of execution frequencies of CFG edges, but...}
Instability of the WCEP (1)

$\text{WCET}_{\text{EST}}$ of basic block $a$

- $10$ Cyc.
- $50$ Cyc.
- $80$ Cyc.
- $65$ Cyc.
- $120$ Cyc.
Instability of the WCEP (2)

- Initial WCEP: `main, a, b, c`
- Length of WCEP = $\text{WCET}_{\text{EST}}$: 205
- In the following: optimization of `b`

$\text{WCET}_{\text{EST}} = 205$ Cyc.
Instability of the WCEP (3)

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

WCET_{EST} = 205 Cyc.
Instability of the WCEP (4)

- Novel WCEP: main, d, c
- Novel WCET<sub>EST</sub>: 195

\[ WCET\text{ has changed due to an optimization! } \]
Issues during $\text{WCET}_{\text{EST}}$ Minimization

The Worst-Case Execution Path (WCEP)
- $\text{WCET}$ of a program = Length of the program’s longest execution path (WCEP)
- $\text{WCET}_{\text{EST}}$ Minimization: Optimization of only those parts of a program lying on the WCEP
- Code optimization apart the WCEP will not reduce $\text{WCET}_{\text{EST}}$

$\mathcal{F}$ Optimizations minimizing $\text{WCET}_{\text{EST}}$ require detailed knowledge of the WCEP...

$\mathcal{F}$ ... and its changes in the course of an optimization.
Outline

1. Introduction and Motivation
2. Procedure Positioning
   - Impact of Code Memory Layout on Cache Performance
   - WCET-aware Procedure Positioning
3. Scratchpad Allocation
Eviction of Code from Instruction Caches

- Caches exploit the locality of memory references
  - *Spatial locality:* Memory references access a very limited range of addresses, so that this address range should be kept in the cache.
  - *Temporal locality:* In a short period of time, spatially distributed addresses are frequently accessed, so that these distributed addresses should be kept in the cache.
- Bad placement of code (or data) in memory can lead to poor cache performance when temporal locality is high
- Addresses exhibiting a high temporal locality can evict themselves from the cache if they are placed badly, leading to a high number of *cache conflict misses.*
Example for I-Cache Eviction (1)

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

Here: 2-way set-associative I-Cache
Example for I-Cache Eviction (2)

Here: 2-way set-associative I-Cache

```c
void foo1()
{
    for (i=0; i<n; i++) {
        foo2();
        foo3();
        // More Code
    }
}
```
Example for I-Cache Eviction (3)

Here: 2-way set-associative I-Cache
Example for I-Cache Eviction (4)

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

Here: 2-way set-associative I-Cache
A better Layout without Evictions (1)

<table>
<thead>
<tr>
<th>Set</th>
<th>I-Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>foo1</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set</th>
<th>Main Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>foo1</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>foo2</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>foo3</td>
</tr>
</tbody>
</table>

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

Here: 2-way set-associative I-Cache
A better 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 Layout without Evictions (3)

void foo1()
{
    for (i=0; i<n; i++) {
        foo2();
        foo3();
    // More Code
    }
}
A better 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:** 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 $e = \{u, v\}$ if a function $u$ calls function $v$
- Each edge $e = \{u, v\}$ is weighted with the frequency $w(e)$ how often $u$ and $v$ 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 a low-level intermediate format

Initialization
- Perform WCET analysis of $P$;
- Generate call graph $G_{\text{orig}} = (V_{\text{orig}}, E_{\text{orig}}, w_{\text{orig}})$ of $P$ based on WCET data;
- Generate call graph $G_{\text{new}} = (V_{\text{new}}, E_{\text{new}}, w_{\text{new}}) = G_{\text{orig}}.\text{copy}()$;

[P. Lokuciejewski et al., WCET-driven Cache-based Procedure Positioning Optimizations, ECRTS, Prague, 2008]
WCET-aware Procedure Positioning (2)

Optimization Loop

do
  \[ \text{wcet}_{\text{current}} = \text{getWCET}(P); \]
  for ( \text{all edges } e = \{u, v\} \in E_{\text{new}}, \]
    \text{sorted in descending order w.r.t. } w_{\text{new}} \}
  \]
  if ( \text{Positioning}(e, G_{\text{new}}, G_{\text{orig}}, P, \text{wcet}_{\text{current}}) == \text{true} )
    \]
    \text{// If contiguous placement of nodes } u \text{ and } v \text{ in memory reduces } \]
    \]
    \text{WCET}_{\text{EST}}, \text{ terminate for-loop and continue with do-while-loop.} \]
  break;
while ( \text{<P was modified during last iteration>} );
WCET-aware Procedure Positioning (3)

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

Generate LLIR $P'$ with $u$ and $v$ placed contiguously in memory;
Perform WCET analysis of $P'$;
$wcet_{new} = \text{getWCET}( P' )$;
if ( $wcet_{new} < wcet_{current}$ )
  $P = P'$;
  Merge nodes $u$ and $v$ 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 should (A, B) and (D, E) be placed in the next step?
– \( G_{\text{orig}} \) reveals that A and D should be placed contiguously

\( G_{\text{new}} \)

© H. Falk | 7/8/2014
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
- 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
Relative $\text{WCET}_{\text{EST}}$ after WCET-Cloning & Positioning

- 100% = $\text{WCET}_{\text{EST}}$ w/o Procedure Positioning
- I-Cache: 16kB, 2-way set-associative, LRU replacement
- WCET-Positioning: $\text{WCET}_{\text{EST}}$ reduction by up to 22%
- Heuristic: Works on initial call graph only, does not call aiT iteratively
Outline

1. Introduction and Motivation
2. Procedure Positioning
3. Scratchpad Allocation
   - SPM Allocation of Data
   - Results
Caches vs. Scratchpad Memories (SPM)

**Caches:**
- Hardware-controlled
- Cache contents difficult to predict
- Latencies of memory accesses highly variable
- $WCET_{EST}$ often imprecise
- Caches often deactivated in hard real-time systems

**Scratchpads:**
- No autonomous hardware
- SPM seamlessly integrated in processor’s address space
- Latencies of memory accesses constant
- $WCET_{EST}$ extremely precise
- SPM contents to be defined by compiler
Timing Predictability of Caches & SPMs (G.721)

SPMs are – in contrast to caches – highly predictable: $\text{WCET}_{\text{EST}}$ scale with ACETs

[L. Wehmeyer, P. Marwedel, Influence of Memory Hierarchies on Predictability for Time Constrained Embedded Software, DATE, Munich, 2005]
ILP for WET-aware SPM Allocation of Data

Goal
– Determine set of data objects (global variables or static local variables) to be allocated to the SPM
– such that selected data objects lead to overall minimization of WCET\textsubscript{EST}
– under consideration of switching WCEPs.

Approach
– Integer-linear programming (ILP)
  ⚫ Optimality of results

– In the following: uppercase = constants, lowercase = variables
Decision Variables & Costs

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

  \[ y_i = \begin{cases} 
  1 & \text{if data object } d_i \text{ is assigned to } \text{mem}_{spm} \\
  0 & \text{if data object } d_i \text{ is assigned to } \text{mem}_{main} 
\end{cases} \]

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

  \[ c_j = C_j - \sum_{d_i \in \text{data objects}} G_{i,j} \cdot y_i \]

  \( c_j \) models the \( WCET_{EST} \) 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 \( WCET_{EST} \) 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

- **Modeling of a function’s control flow:**

**Acyclic sub-graphs:**

\[ w_A \geq w_B + c_A \]
\[ w_A \geq w_C + c_A \]
\[ w_B \geq w_D + c_B \]
\[ w_C \geq w_D + c_C \]
\[ w_D \geq w_E + c_D \]
\[ w_E = c_E \]

\[ w_A = \text{WCET of longest path starting at } A \]

**Reducible Loops:**

- Treat body of innermost loop \( L \) like acyclic sub-graph
- Fold loop \( L \)
- Costs of \( L \):
  \[ c_L = w_G \cdot C^L_{\text{max}} \]
- Continue with next innermost loop

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

- **Modeling function calls:**
  - Each function $F$ has dedicated entry BB $b_{entry}^F$

  $\Downarrow w_{entry}^F$ models WCET_{EST} of $F$ if $F$ is executed exactly once

  $\Downarrow$ If function $F$ is called: add $w_{entry}^F$ to WCET_{EST} of caller
Call Penalties

- **Call penalties for basic block** $b_j$:

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

  If $b_j$ calls $F$, add $\text{WCET}_{\text{EST}}$ of $F$ to call penalty.

- **Final control flow constraints per basic block** $b_j$:

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

  Add call penalty to variable $w_j$ modeling $\text{WCET}_{\text{EST}}$ of any path starting at $b_j$.
Scratchpad Capacity and Objective Function

- **Scratchpad capacity constraint:**
  \[
  \sum_{d_i \in \text{data objects}} (S_i \times y_i) \leq S_{spm}
  \]

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

- **Objective function:**
  - \( w_{\text{entry}}^F \) models WCET\(_{\text{EST}}\) of \( F \) if \( F \) is executed exactly once
  - Variable \( w_{\text{entry}}^{\text{main}} \) models WCET\(_{\text{EST}}\) of the entire program

\[ \phi w_{\text{entry}}^{\text{main}} \xrightarrow{} \min. \]

[F. Rotthowe, Scratchpad Allocation of Data for Worst-Case Execution Time Minimization (in German), Diploma Thesis, TU Dortmund, 2008]
Support of the ILP by WCC Infrastructure

Max. Iteration counts of loop $L$: $C^L_{max}$

**ANSI-C Sources & Flow Facts**

- High-Level ICD-C
- Loop Analyzer
- ILP-based WCET-aware SPM Code Allocation
- WCET-Optimized Assembly
- Linker Script

- Low-Level LLIR
- LLIR Code Selector
- aiT WCET Analysis
- Memory Hierarchy Specification

**WCET_{EST} of BB $b_j$ using main memory:** $C_j$

Size of data object $d_i$: $S_i$

- SPM Size $S_{spm} = 40$ kB
- SPM Access = 1 Cycle
- Flash Access = 6 Cycles
Data SPM – WCET_{EST} for petrinet

- Notable WCET_{EST} reductions already for SPMs of only a few bytes
- 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
Data SPM – WCET\textsubscript{EST} for \texttt{fsm}

- More steady WCET\textsubscript{EST} reductions for increasing SPM sizes
- WCET\textsubscript{EST} reductions by 21.4% for 256 bytes SPM
- X-Axis: Absolute SPM sizes
- Y-Axis: 100% = WCET\textsubscript{EST} when not using SPM at all
Data SPM – Average $\text{WCET}_{\text{EST}}$ for 14 Benchmarks

- Steady $\text{WCET}_{\text{EST}}$ decreases for increasing SPM sizes
- $\text{WCET}_{\text{EST}}$ reductions from 2.6% – 20.2%
- X-Axis: Absolute SPM sizes
- Y-Axis: 100% = $\text{WCET}_{\text{EST}}$ when not using SPM at all