the WCET-aware C Compiler WCC

Heiko Falk
Embedded Systems/Real-Time Systems
Ulm University, Germany

Jan C. Kleinsorge
Computer Science 12
TU Dortmund, Germany
Outline

- Introduction and Motivation
- Structure of the WCET-aware C Compiler WCC
- Systematic WCET Minimization by Compiler Optimizations
- Discussion
- Coffee Break
- WCC’s WCET-aware Optimizations
- Conclusions and Outlook
- Discussion
Outline

- Introduction and Motivation
  - Safety-critical real-time systems and Worst-Case Execution Times
  - Current state of the art in industrial design of hard RT systems
  - Current state of the art in compiler construction
  - Static WCET analysis
- Structure of the WCET-aware C Compiler WCC
- Systematic WCET Minimization by Compiler Optimizations
  - Discussion
- WCC’s WCET-aware Optimizations
- Conclusions and Outlook
  - Discussion
Safety-Critical Real-Time Systems

Real-Time Systems
- Computation of a function $f(x)$ must complete within given time bounds
- Too late computation of $f(x) \approx$ incorrect computation of $f(x)$

Hard/Safety-Critical Real-Time Systems
- Late computation of $f(x) \rightarrow$ 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 (WCET)

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

*WCET in general not computable*

[Borrowed from Reinhard Wilhelm]
Design of 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 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 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 the compiler
  - ACET optimizations potentially increase WCETs – Compilers often invoked without any optimizations for real-time systems
Motivation

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 (WCET_{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 WCET$_{EST}$ of each individual basic block.
Workforce of the WCET Analyzer aiT (4)

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

– Introduction and Motivation
– Structure of the WCET-aware C Compiler WCC
  – Integration of a WCET timing model into the compiler
  – Specification of memory hierarchies
  – Annotation and transformation of Flow Facts for WCET analysis
  – Automatic polyhedral loop bound analysis
– Systematic WCET Minimization by Compiler Optimizations
– Discussion

– WCC’s WCET-aware Optimizations
– Conclusions and Outlook
– Discussion
Integration of WCET into WCC Compiler (1)

- Supported processors: Infineon TriCore TC1796 and TC1797; ARM7
- Re-implementation of a WCET timing model in compiler makes no sense
- Instead: 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:
- $\text{WCET}_{\text{EST}}$ 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

Automotive Engine Control System

DemoCar
- AUTOSAR-compliant lightweight Engine Control (BOSCH)
- 54,000 lines of code, 7 Tasks (3 event-triggered, 4 time-triggered), 27 Runnables
WCC Tool Demonstration using DemoCar

Integration WCC ↔ aiT
- Runnable IgnitionSWCSync: Activated 4x per crankshaft rotation highly timing critical
- Code of runnable allocated to L2 flash memory
- Demo: Code generation by WCC, including fully automated WCET analysis using aiT

CPU Runtimes
- ≈ 40 CPU seconds for code generation, optimization (-O3) and WCET analysis

☞ WCET-aware code generation and optimization feasible in practice!
Memory Hierarchies and Execution Times

Memories and Execution Times
– Overall system performance largely dominated by memory subsystem
– Large speed gap between slow memories and fast processors
– Execution time of software depends on characteristics
  of underlying memory hierarchy and
  of memory accesses performed by a program

WCET estimates also heavily depend on memories!
Memory Hierarchies of Embedded Processors

Example: Infineon TriCore TC1796
- Embedded automotive processor with complex memory hierarchy

- Separate busses / memories for code and data
- L1 Scratchpad for Data
- L1 Scratchpad and Cache for Code
- L2 Data SRAM
- L2 Flash for Code and Data, cacheable or not
Consequences for WCET-aware Compiler

Compiler and Memory Hierarchies
- Compiler is responsible to pass memory-related information to WCET analyzer
- WCET estimates must adhere to a program’s actual memory layout

The compiler...
- ... decides on a program’s memory layout, and not the linker!
- ... should be able to optimize a program’s memory layout in order to minimize WCETs (cf. memory-related optimizations later)
- ... needs detailed knowledge about memories
Specification of Memory Hierarchies

Plain-text interface per memory region:
- base address, length
- access attributes
- access times
- assembly-level sections

# Data SRAM (DMU)
[DMU–SRAM]
origin = 0xc0000000
length = 0x10000 #64k
attributes = RWA
cycles = 6
sections = .data.sram
Flow Facts (1)

Static WCET Analysis
- Estimates the longest possible execution path
- Such paths can contain cycles stemming from loops and/or recursions
  - How many times are such cycles iterated in the worst case?
  - Traversals of cycles have to be upper-bounded for WCET analysis!

Flow Facts...
- ... are (user-provided) meta-information containing such upper bounds
- ... have to be supported by a WCET-aware compiler
Flow Facts (2)

aiT’s Flow Fact Support

– Via separate annotation file (cf. slide 15)
– Contains loop bounds and recursion depths at machine code level, i.e. based on physical memory addresses
  ❆ Extremely low-level, tedious to generate Flow Facts
  ❆ Cumbersome since Flow Facts must be (manually) updated if a program’s memory layout changes

WCC’s Flow Fact Support

– High-level, directly within ANSI-C source codes
– No relation to machine code level required for programmer
– Automatic Flow Fact update during code transformations
Annotation of Loop Bounds

Simple Loops with Constant Bounds

```c
 Pragma( "loopbound min 100 max 100" )
 for ( i = 1; i <= 100; i++ )
     Array[ i ] = i * fact * KNOWN_VALUE;
```

A Data-dependent Loop
- E.g. if loop depends on a function’s parameter `maxIter` whose possible min/max values are known:

```c
 Pragma( "loopbound min 50 max 100" )
 for ( i = 1; i <= maxIter; i++ )
     Array[ i ] = i * fact * KNOWN_VALUE;
```
Loop Bounds vs. Flow Restrictions

Loop Bounds
– Applicable for well-structured loops
  – for-, while-do- and do-while-loops
  – with single-entry property
  – with well-defined termination criterion
– Not applicable for loops with multiple entries or without explicit termination criterion or including goto’s

Flow Restrictions
– Applicable for irregular code structures, incl. recursion
– Allow to formulate linear relations between execution frequency of one C statement and that of other statements
Annotation of Flow Restrictions (1)

A Triangular Loop

```c
#pragma( "marker outer" )

Statement A;
```

```c
for ( i = 0; i < 10; i++ )
  for ( j = i; j < 10; j++ )
#pragma( "marker inner" )

Statement B;
```

```c
#pragma( "flowrestriction 1*inner <= 55*outer" );
```

To read as follows

- The execution frequency of B is at most 55 times that of A
Recursion

```c
int fib(int i)
{
    if ((i == 0) || (i == 1))
        return 1;
    return (fib(i - 1) + fib(i - 2));
}
```

```c
int main()
{
    int In = fib(7);
    _Pragma("marker rec");
    _Pragma("flowrestriction \1*fib <= 41*rec");
    return In;
}
```
Flow Fact Translation

Semantic Gap
- Flow Facts are specified at ANSI-C level, but used during static WCET analysis at machine code level
- Flow Facts have to be propagated through all compiler stages lowering the intermediate code’s abstraction level

Flow Fact Managers
- ANSI-C to ICD-C: Extracts pragmas from source code, attaches them to WCC’s high-level IR
- ICD-C to LLIR: Translates high-level Flow Facts to machine code level Flow Facts during code selection
- LLIR to aiT: Produces aiT-specific Flow Facts for WCET analysis
Flow Fact Update (1)

What about Optimizations?
– Optimizations potentially drastically change execution frequencies of loops or functions

Flow Facts have to be aware of such code changes and must be updated accordingly!

Example: Loop Unrolling

```c
__Pragma( "loopbound min 100 max 100" )
for ( i = 1; i <= 100; i++ )
    Array[ i ] = i * fact * KNOWN_VALUE;
```
Flow Fact Update (2)

What about Optimizations?

- Optimizations potentially drastically change execution frequencies of loops or functions

Flow Facts have to be aware of such code changes and must be updated accordingly!

Example: Loop Unrolling

```c
#pragma( "loopbound min 100 max 100" )
for ( i = 1; i <= 100; i += 2 ) {
    Array[ i ] = i * fact * KNOWN_VALUE;
    Array[ i+1 ] = (i+1) * fact * KNOWN_VALUE;
}
```
Flow Fact-aware Optimizations

WCC’s Optimizations

- Are aware of Flow Facts and automatically update Flow Facts whenever transformations modify a piece of code’s execution frequencies
- Update of Flow Facts is tightly integrated into the optimizations themselves

Optimizations apply basic Flow Fact operators:
- creation, copying, deletion of loop bounds and flow restrictions
- displacement of min/max interval of loop bounds
- replacement of a flow restriction by an equivalent one
- replacement of a flow restriction by a nonequivalent but safe one
WCC’s Flow Fact Mechanisms

Flow Fact-Awareness:
- Transversal concept in the entire compiler
- Starting in the front-end
- Touching all IRs and code selection
- Covering all optimizations
- Ending in the back-end during WCET analysis
Automatic Loop Bound Analysis

Disadvantages of Manual Flow Fact Annotation
– Time-consuming procedure for developer
– “Stupid” and error-prone work
– Risk of wrong annotations and thus unsafe WCET estimates

WCC’s Loop Analyzer
– Operates on high-level IR ICD-C, i.e. on ANSI-C code
– Automatically extracts loop bound Flow Facts required for WCET analysis for broad classes of loops
– Combines interprocedural program slicing, abstract interpretation and polytope models for fast and precise analysis
Program Slicing and Abstract Interpretation

**Program Slicing**
- Finds program statements relevant for a particular computation
- Here: slicing criterion \( <q, V> \) for a loop exit condition \( q \) and all variables \( V \) used or defined by \( q \)
- Result of slicing: all statements that influence a given loop exit condition \( q \)

**Abstract Interpretation (AI)**
- Technique for sound approximation of a program’s semantics
- Abstract states (here: min/max intervals for loop iteration counts) reduce complexity of program analysis, since general loop analysis is undecidable
Polyhedral Loop Bound Analysis

Disadvantage of Abstract Interpretation
- Fixed-point algorithm computing abstract states for CFG nodes
- Slow due to iterative approach, in particular for loops with many iterations

Polyhedra
- $N$-dimensional geometrical objects, defined by linear (in)equations
- Used to represent loop nests and affine condition expressions
- Non-iterative algorithms to efficiently compute size of polyhedra, i.e. number of loop iterations, are well-known
- If sliced loop can be represented as polyhedron, do polyhedral analysis; otherwise, use abstract interpretation as fallback
WCC’s Loop Analyzer

- 99% of 707 loops analyzable
- Exact loop iteration counts for 96% of the loops
- Analysis times between few CPU seconds and max. 4 minutes
- Automatic generation of loop bound Flow Facts

[P. Lokuciejewski, A Fast and Precise Static Loop Analysis..., CGO, 2009]
WCC – The Complete Picture

- Integration of static WCET analysis into compiler
- Transversal support of mandatory Flow Facts
- Loop Analyzer for automatic Flow Fact generation
- WCET timing data available at all IRs

[H. Falk et.al., A compiler framework for the reduction of WCETs, Springer RTS, 2010]
Outline

– Introduction and Motivation
– Structure of the WCET-aware C Compiler WCC
– Systematic WCET Minimization by Compiler Optimizations
  – The Worst-Case Execution Path (WCEP)
  – Challenges imposed by the WCEP
– Discussion

– WCC’s WCET-aware Optimizations
– Conclusions and Outlook
– Discussion
Challenges during WCET\textsubscript{EST} Minimization

The \textit{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}

\begin{itemize}
  \item Optimizations minimizing WCET\textsubscript{EST} require detailed knowledge of the WCEP...
\end{itemize}

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

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

10 Cyc. → main

50 Cyc. → a

80 Cyc. → b

65 Cyc. → c

120 Cyc. → d

© H. Falk | 2012-03-31
Instability of the WCEP (2)

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

\[ WCET\textsubscript{EST} = 205 \text{ Cyc.} \]
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.

10 Cyc. main

50 Cyc. a

40 × Cyc. b

120 Cyc. d

65 Cyc. c
Instability of the WCEP (4)

- Novel WCEP: `main, d, c`
- Novel WCET\textsubscript{EST}: 195

\textit{WCEP has changed due to an optimization!}
Challenges during $\text{WCET}_{\text{EST}}$ Minimization

The Worst-Case Execution Path (WCEP)
- 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 $\text{WCET}_{\text{EST}}$

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

$\Rightarrow$ ... and of its changes in the course of an optimization.
Outline

- Introduction and Motivation
- Structure of the WCET-aware C Compiler WCC
- Systematic WCET Minimization by Compiler Optimizations
  - Discussion

- Coffee Break

- WCC’s WCET-aware Optimizations
- Conclusions and Outlook
  - Discussion
Reconciling Compilers & Timing Analysis for Safety-Critical Real-Time Systems
–
WCET-aware program optimizations

Heiko Falk
Embedded Systems/Real-Time Systems
Ulm University, Germany

Jan C. Kleinsorge
Computer Science 12
TU Dortmund, Germany
Outline

- WCET-aware Optimizations and Code Quality
- Graph Coloring Register Allocation
- Scratchpad Memory Allocation
- Cache-aware Memory Content Selection
- Cache Partitioning for Multi-task Real-time Systems
- Combination of Scratchpad Allocation, Memory Content Selection and Cache Partitioning
WCET-aware Optimizations and Code Quality

WCET as objective function
- Actual speed-up but also by enhancing analyzability
- Side-effects of changes on timing hard to anticipate
- Issuing just a single instruction can lead to uncertainty regarding:
  - Location, alignment, access pattern (cache), schedule (pipeline), branch prediction, etc.

Code quality for WCET-aware optimizations
- Avoid dynamic dispatch, excessive inflation and layout changes without being clear about its effects
- In short: maintain predictability first
WCET-aware Optimizations and Systems (1)

Semantics:

- Computation
  - Expressions
- Layout
  - Insn (virt.)
- Pipeline
  - Insn (phys.)
- System
  - “BLOB”

Open parameters:

- + Dependencies
- + Order, registers
- + Location: accesses to busses, memories
- + Ideally: just program input

Uncertainty
WCET-aware Optimizations and Systems (2)

Practical heuristics to pick the right level of abstraction:

- Still on WCEP?
- Will decision change WCEP?
- Are side-effects possible and (in how far) are they bounded?
- What's the overall impact on the system?
- How often do we need to re-evaluate intermediate solutions?

Uncertainty that cannot be tackled at any level:

- Speculative execution, cache hierarchies (and replacement policies), timing anomalies in general, general I/O, etc.
- However: Trend towards (many) simpler cores in fact improves situation as far as per-task predictability is concerned
Outline

- WCET-aware Optimizations and Code Quality
- Graph Coloring Register Allocation
- Scratchpad Memory Allocation
- Cache-aware Memory Content Selection
- Cache Partitioning for Multi-task Real-time Systems
- Combination of Scratchpad Allocation, Memory Content Selection and Cache Partitioning
Workflow of Graph Coloring RA

1. **Initialization:** Build Interference Graph $G = (V, E)$ with $V = \{\text{virtual registers}\} \cup \{K \text{ physical processor registers}\}$, $e = (v, w) \in E \iff v$ and $w$ may never share the same PHREG, (i. e. $v$ and $w$ interfere)

2. **Simplification:** Remove all nodes $v \in V$ with degree $< K$

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

4. **Repeat** steps 2 and 3 until $G = \emptyset$

5. **Coloring:** Successively re-insert nodes $v$ into $G$ in reverse order; if there is a free color $k_v$, color $v$; else, mark $v$ as actual spill

Problem of Standard Graph Coloring

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

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 selection,
- … the node with highest degree in the interference graph,
- … a node with high degree, with many DEFs/USEs, in some inner loop – maybe depending on profiling data.

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

- Derived from classic Chaitin graph coloring
- Registers allocation as a problem from the „tip“ of the memory hierarchy
- Besides runtime overhead, spill-code affects:
  - Instruction count, schedule, memory layout, cache access and pattern, etc.
- WCET-aware optimization must take into account:
  - Where to store data (actual allocation decision)?
  - **But also:** Where to store (spill) instruction (relative to WCEP)?

**The catch:**
- …relies on WCET data provided by WCET analysis using aiT
- …can’t obtain WCET data since code contains **virtual** registers
WCET-aware RA: of Chickens and Eggs

Pessimistic register allocation:

- Start by marking all VREGs as actual spill (each VREG is spilled. Now code is fully analyzable)
- Perform WCET analysis, get WCEP
- Allocate VREGs of basic block $b$ with *most worst-case spill code executions* to PHREGs using standard GC on original program
- Re-evaluate novel WCEP
- Stop and allocate rest if no more VREGS on WCEP
Results – Worst-Case Execution Times

100% = WCET_{EST} using Standard Graph Coloring (highest degree)
**Results – Average-Case Execution Times**

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

Summary

- Standard graph coloring unaware of worst-case properties
- May thus lead to uncontrolled spill code generation along WCEP
- WCET-aware register allocation: combination of standard graph coloring with WCET-aware spill heuristic
- Average WCET reductions over 46 benchmarks: 31.2%

Caveats

- “Bad” spills not revocable, might unbalance pipeline load
- Experiments with highly accurate ILP-based WCET-aware register allocation

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

- WCET-aware Optimizations and Code Quality
- Graph Coloring Register Allocation
- **Scratchpad Memory Allocation**
- Cache-aware Memory Content Selection
- Cache Partitioning for Multi-task Real-time Systems
- Combination of Scratchpad Allocation, Memory Content Selection and Cache Partitioning
Caches vs. Scratchpad Memories \((SPM)\)

**Caches:**
- Processor
  - L1-Cache
    - Main Memory

**Scratchpads:**
- Processor
  - SPM
    - Main Memory

- Hardware-controlled
- Cache contents difficult to predict statically
- Latencies of memory accesses highly variable
- \(WCET_{EST}\) often imprecise
- Caches often deactivated in hard real-time systems

- 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
Scratchpad Allocation: Variants and Caveats

Characteristics of code and data allocation:

- Data relocation and mutation „naturally“ supported by architecture
- Code relocation usually requires modification of instructions
  - Locality annihilated
  - (Potentially already optimized) Runtime properties destroyed

Static and dynamic scratchpad optimization:

- Static: Precompute global and static relocation, maintain order (therefore locations implicit)
- Dynamic: Precompute dynamic exchange of SPM contents
  - Perspective of static analysis: self-modifying code
  - Static dispatch (overlaying targets)
  - Memory allocation is hard
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: no need for backtracking techniques

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

- Binary decision variables per basic block (BB):

\[ x_i = \begin{cases} 
1 & \text{if basic block } b_i \text{ is assigned to } \text{mem}_{spm} \\
0 & \text{if basic block } b_i \text{ is assigned to } \text{mem}_{main} 
\end{cases} \]

- Costs of basic block \( b_i \):

\[ c_i = C_{main}^i \times (1 - x_i) + C_{spm}^i \times x_i \]

\( c_i \) models the \( \text{WCET}_{\text{EST}} \) of \( b_i \) if it is allocated to main memory or SPM, respectively.
Intraprocedural Control Flow

- Modeling of a function’s control flow:

Acyclic sub-graphs:

\[
\begin{align*}
  w_A & \geq w_B + c_A \\
  w_A & \geq w_C + c_A \\
  w_B & \geq w_D + c_B \\
  w_C & \geq w_D + c_C \\
  w_D & \geq w_E + c_D \\
  w_E & = c_E
\end{align*}
\]

\( w_A = \text{WCET of any path starting at A} \)

(Reducible) Loops:

- Treat body of inner-most loop \( L \) like acyclic sub-graph
- Fold loop \( L \)
- Costs of \( L \):
  \[
  c_L = w_B \times C_{max}^L
  \]
- Continue with next innermost loop

\[\text{V. Suhendra et al., WCET Centric Data Allocation to Scratchpad Memory, RTSS 2005}\]
Cross-Memory Jumps

- Allocation of consecutive BBs:
  - Allocation of consecutive BBs in the CFG to different memories requires adaption/insertion of dedicated jumping code
  - Cross-memory jumps are costly
  - Jumping code: variable overhead in terms of WCET_{EST} and code size, depending on decision variables

- Jump scenarios:

  a) Implicit  
  b) Unconditional  
  c) Conditional
Penalties for Cross-Memory Jumps

- Penalties for jump scenarios ($\otimes = \text{Boolean XOR}$):
  - **Penalty for Implicit jumps**: $jP_{impl}^i$
    
    $jP_{impl}^i = (x_i \otimes x_j) \times P_{high}$
    
    Add high penalty if BBs $i$ and $j$ are placed in different memories.

- **Penalty for Unconditional jumps**: $jP_{uncond}^i$
  - 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:

- **Conditional jumps**: Obvious combination of $jP_{impl}^i$ and $jP_{uncond}^i$
Jump Penalties & Interprocedural Control Flow

- **Jump penalties for basic block** $b_i$:
  
  $$jp_i = \begin{cases} 
  jjp^i_{impl} & \text{if Jump Scenario of } b_i \text{ is implicit} \\
  jjp^i_{uncond} & \text{if Jump Scenario of } b_i \text{ is unconditional} \\
  jjp^i_{cond} & \text{if Jump Scenario of } b_i \text{ is conditional} \\
  0 & \text{else}
  \end{cases}$$

- **Modeling of the global control flow:**
  - Variable $w^F_{entry}$ models cost of WCEP starting at $b^F_{entry}$
  - If $F'$ calls $F$, $w^F_{entry}$ must be added to WCET_{EST} of $F'$
Call Penalties

- Call penalties for basic block $b_i$:
  \[
  cp_i = \begin{cases} 
  w_{entry}^F + (x_i \otimes x_{entry}^F) \cdot P_{high} & \text{if } b_i \text{ calls } F \\
  + (1 - (x_i \otimes x_{entry}^F)) \cdot P_{low} & \text{else} \\
  0 & \text{else}
  \end{cases}
  \]

  If $b_i$ calls $F$, add $\text{WCET}_{\text{EST}}$ of $F$ to call penalty. Furthermore, add $P_{high}$ if $b_i$ contains cross-memory call, $P_{low}$ otherwise.

- Final control flow constraints per basic block $b_i$:
  \[
  \forall (b_i, b_{\text{succ}}) : w_i \geq w_{\text{succ}} + c_i
  \]

  Add jump and call penalties to variable $w_i$ modeling $\text{WCET}_{\text{EST}}$ of any path starting at $b_i$
Scratchpad Capacity

- Size of BB $b_i$ depends on actual jumping code for $b_i$:
  - Size $s_i$ of jumping code for $b_i$:
    - # bytes for jumping code, depending on jump/call scenario
  - Total size of basic block $b_i$:
    - Size $S_i$ of $b_i$ without any jumping code plus
    - Size $s_i$ of $b_i$’s jumping code

\[
\sum_{b_i} (S_i \times x_i + s_i \times x_i) \leq S_{spm}
\]

Objective Function

- $\text{WCET}_{\text{EST}}$ of entire program:
  - Variable $w_{\text{entry}}^\text{main}$ models $\text{WCET}_{\text{EST}}$ of entire program
  - $w_{\text{entry}}^\text{main} \xrightarrow{\sim} \text{min.}$
Average WCET<sub>EST</sub> for 73 Benchmarks

- Steady WCET<sub>EST</sub> decreases for increasing SPM sizes
- WCET<sub>EST</sub> reductions from 7% – 40%
Summary & Caveats

Summary

- Current state of the art:
  - Neglects varying jumping code in basic blocks
  - Select one element of power set of basic blocks
- Our approach:
  - Models changing WCEPs
  - Uses jump scenarios to cope with varying jumping code

Caveats

- Implicit control-flow model requires well-structured code
- No component-wise compilation

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

- WCET-aware Optimizations and Code Quality
- Graph Coloring Register Allocation
- Scratchpad Memory Allocation
- Cache-aware Memory Content Selection
- Cache Partitioning for Multi-task Real-time Systems
- Combination of Scratchpad Allocation, Memory Content Selection and Cache Partitioning
Cache-aware Memory Content Selection

- Compilers good at dealing with registers (register/stack)
- WCC good at SPM-allocation (spm/main memory)
- Aspects of cache-aware optimizations:
  - Generally unresolved problem due to system-wide influence of local decisions and generally unknown cache parameters
  - Only generalized attempts on data - like loop transformations - to improve average access pattern on data
- For predictability and idle optimization potential in code:
  - Divide program in cached or uncached parts
  - *Software-controlled memory content selection* to adapt to actual access pattern
**Cache-aware Memory Content Selection**

Example for unprofitable memory layout:
- Mutual eviction of functions
- Could lead to a highly increased $WCET_{EST}$ due to thrashing

```c
void foo1() {
    for(i=0; i<100; i++) {
        foo2();
        foo3();
        ...
    }
}
```
Cache-aware Memory Content Selection

Basic idea:

- Step-wise allocation of functions to cached memory areas
- Select functions whose $WCET_{EST}$ benefits most from cached execution

atism-profitable functions w. r. t. a program’s $WCET_{EST}$ must not evict profitable ones from cache
- Hill-climbing approach with a “profit”-metric:

$$profit(F) = \frac{WCET_{uncached}(F) - WCET_{cached}(F)}{size(F)}$$
Memory Content Selection Algorithm (1)

LLIR mcs( LLIR P, Cache cache ):

    // Precompute profit
    profit = computeFunctionProfit( P )

    // Fill cache exactly once
    for_each( sort( F in P, profit ) ):
        allocate( F, cache )
        if cache.full:
            break

    // Perform WCET-aware cache-allocation
    ...

Memory Content Selection Algorithm (2)

// “Overcharge” cache memory unless $\text{WCET}_{\text{EST}}$ degrades
wcet = computeWCET( P )
profit = computeFunctionProfit( P )

// As before: most profitable function first
for_each( sort( F in P, profit ) ):
    allocate( F, cache )
    tmp = computeWCET( P )

// Only keep improvements
if ( wcet < tmp ):
    deallocate( F, cache )
else
    wcet = tmp
    profit = computeFunctionProfit( P )
Results compared to unoptimized cache usage

![Graph showing relative WCET results compared to unoptimized cache usage for various benchmarks. The graph includes bars for 5%, 10%, and 20% cache usage, with some benchmarks such as 'statemate' and 'Average' showing a 20% improvement in relative WCET compared to unoptimized cache usage.](image-url)
Summary & Caveats

Conclusion

- Iterative approach ensures optimizing along a possibly switching WCEP.
- Profitable functions not evicted from cache by unprofitable ones w.r.t. their WCET_{EST}.
- Achieves WCET_{EST} reductions of up to 20%.

Caveats

- Greedy approach (upside: direct, simple).
- Functions as allocation units might be too coarse.

[S. Plazar, P. Lokuciejewski and P. Marwedel, WCET-driven Cache-aware Memory Content Selection, ISORC 2010]
Outline

- WCET-aware Optimizations and Code Quality
- Graph Coloring Register Allocation
- Scratchpad Memory Allocation
- Cache-aware Memory Content Selection
- **Cache Partitioning for Multi-task Real-time Systems**
- Combination of Scratchpad Allocation, Memory Content Selection and Cache Partitioning
Software-based Cache Partitioning

General thoughts on presented optimization strategies:

- Until now, greedy relocation successful strategy to get around intra-task cache conflicts due to tight coupling with static WCET analysis
- Fails in multi-task environments: Analysis unaware of potential preemptions
- Safety can only be achieved by guaranteeing no collisions
- Granularity: instructions (possibly splitting basic blocks)

Intuition:

- Divide the cache into partitions of optimal size
- Assign one task per partition to prevent mutual eviction
Software-based Cache Partitioning

- Exploit the cache addressing logic (index bits)
- Distribute memory blocks of tasks over address space
- Ensure mapping to particular cache lines
  - Effectively inverts the logical mapping direction
WCET-aware Cache Partitioning

Greedy approach
- Partition size depends on task’s code size
- Example: 4 tasks with the same code size

Better
- ILP-model to select individual partition size per task
- Take number of activations into account

[F. Müller, Compiler Support for Software-Based Cache Partitioning, 1995]
ILP Formulation

\[ T : \text{set of periodically scheduled tasks } t_1 \ldots t_m \]

\[ P : \text{set of partition sizes } p_0, p_1 \ldots p_j \]

\[ x_{ij} = \begin{cases} 1, & \text{if } t_i \text{ assigned to } p_j \\ 0, & \text{else} \end{cases} \]

\[ WCET_{ij} : WCET_{EST} \text{ if } t_i \text{ assigned to } p_j \]

\[ I : \text{hyperperiod of } T \text{ (lcm of periods)} \]

\[ c_i : \text{activations of } t_i \text{ in } I \]
ILP Formulation

Each task must have a partition assigned:

\[ \forall i = 1..m : \sum_{j=1}^{n} x_{ij} = 1 \]

Keep track of the cache size:

\[ \sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} \cdot p_j \leq S \]
ILP Formulation

Partition-specific WCET per task:

\[ WCET(t_i) = \sum_{j=1}^{n} x_{ij} \times WCET_{ij} \]

Objective function to minimize:

\[ \sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} \times c_i \times WCET_{ij} \rightarrow \min \]
Results: MRTC benchmarks

Average of 100 sets of randomly selected tasks:
- 5 tasks: ~6kB
- 10 tasks: ~12kB
- 15 tasks: ~19kB
Summary & Caveats

Conclusion

- Optimal partition sizes w.r.t. the overall system WCET
- Partitioning introduces predictability for preemptive schedules
- Average WCET reduction of 12% (5 tasks) up to 19% (15 tasks) compared to greedy approach

Caveats

- “Zero-collision” policy can be too conservative depending on the actual cache logic and scheduling policy
- Pre-computation of partitions time consuming
- Locality in address space (basic block splits, instruction corrections)

Outline

- WCET-aware Optimizations and Code Quality
- Graph Coloring Register Allocation
- Scratchpad Memory Allocation
- Cache-aware Memory Content Selection
- Cache Partitioning for Multi-task Real-time Systems
- Combination of Scratchpad Allocation, Memory Content Selection and Cache Partitioning
An Experiment: Combined Approach

Can SPM allocation, memory content selection and cache partitioning be combined?

- Intention is to fully exploit memory hierarchy
- All three severely alter the memory layout due to relocation and partitioning
- Order of application critical for good results

Example:
MCS prior to SPM
Combined Approach (1)

Reasoning about the order of application

- SPM allocation (SPMA) should be performed prior to Memory Content Selection (MCS) and Cache Partitioning (CP)
- CP prior to MCS:
  - Similar to previous example: cache potentially under-utilized
- MCS prior to CP:
  - CP only considers objects designated to be cached by MCS
  - Likely that the greedy MCS decision was inappropriate given the potential exploited by a fine-grained partitioning
- Computing MCS solution per partition in precomputation of CP
- Apply CP ILP to determine optimal combination
Application in order:

- Effects of SPM, CP invoking MCS in preprocessing

Remains uncached (MCS)

Not affected by CP/MCS (SPMA)
Evaluation

Gains in \( WCET_{\text{EST}} \):
- crc, fft1, gsm_decode, trellis

![Diagram showing gains in WCET, gain compared to unoptimized code vs. SPM size and cache size.]

- **Gain compared to unoptimized code**
  - 0% gain from 0% to 100% SPM size
- **SPM size (%)**
- **Cache size (%)**
- **Gains in WCET_{\text{EST}}**:
  - 92% gain at 92% SPM size
  - 73% gain at 25% SPM size
  - 5% gain at 1% SPM size
Conclusion

Remarks

- **WCET-aware compilation:**
  - Compilers usually are unaware of timing
  - Optimistic optimization strategies: no clearly defined objective
  - “Maybe faster but could be worse” doesn’t quite cut it for hard real-time applications (profile-guided optimization no match)
  - Fine-grained optimization decisions span from *well-directed exploitation* over *conflict reduction* to full *conflict freedom*

Challenges

- Multi-tasking: component-wise compilation, interaction, OS
- Multi-core: inter-core communication
- Tailor (fully) predictable but still highly configurable systems