WCET-Aware C Compiler: WCC

Jian-Jia Chen
(slides are based on Prof. Heiko Falk)
TU Dortmund,
Informatik 12

2015年 05 月 05 日
Worst-Case Execution Times (WCET)

Requirements

– **Safeness:** $\text{WCET} \leq \text{WCET}_{\text{EST}}$

– **Tightness:** $\text{WCET}_{\text{EST}} - \text{WECT} \to \text{minimal}$

**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
  \( \Rightarrow \) No guarantee that a real-time system \textit{always} terminates in time

Design Time
– How many iterations are required until step 7 successful?
  \( \Rightarrow \) Depends on in how far steps 2-3 lead to the effective acceleration of the generated code in the worst case
  \( \Rightarrow \) \textit{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 optimizations explicitly aiming at $\text{WCET}_{\text{EST}}$ minimization.
Adopting aiT in Compilation

– **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.
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

[http://1s12-www.cs.tu-dortmund.de/research/activities/wcc]
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!
Flow Facts (1)

**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
– 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;
```
Annotation of Flow Restrictions

A Triangular Loop

```c
Pragma( "marker outer" )
Statement A;

for ( i = 0; i < 10; i++ )
  for ( j = i; j < 10; j++ )
   Pragma( "marker inner" )
  Statement B;

Pragma( "flowrestriction 1*inner <= 55*outer" );
```

To read as follows
– The execution frequency of B is at most 55 times that of A
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 Automated 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
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]
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}

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

\textit{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 $a$

- 10 Cyc.
- 50 Cyc.
- 80 Cyc.
- 65 Cyc.

main

120 Cyc.
Instability of the WCEP (2)

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

WCET\text{\textsubscript{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
Instability of the WCEP (4)

- Novel WCEP: main, d, c
- Novel WCET_{EST}: 195

\( WCEP \text{ has changed due to an optimization!} \)
Challenges 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}

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

... and of its changes in the course of an optimization.
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

- No autonomous hardware
- SPM seamlessly integrated in processor’s address space
- Latencies of memory accesses constant
- \( WCET_{EST} \) extremely precise
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 $\text{WCET}_{\text{EST}}$
  - ...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:**

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

\[
w_E = c_E
\]

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

**Reducible) Loops:**

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

[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\textsubscript{EST} and code size, depending on decision variables

- Jump scenarios:
  a) Implicit
  b) Unconditional
  c) Conditional
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 \textit{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
    $w_{\text{entry}}^{\text{main}} \mapsto \min.$
Average $\text{WCET}_{\text{EST}}$ for 73 Benchmarks

- Steady $\text{WCET}_{\text{EST}}$ decreases for increasing SPM sizes
- $\text{WCET}_{\text{EST}}$ reductions from 7% – 40%

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