OPC: Optimizing and Parallelizing Compilers

Isabelle PUAUT

Université de Rennes I / IRISA Rennes (Pacap)
2017-2018

Compilers and predictable programming

- Compilation and WCET analysis
  - Optimizations and WCET estimation
  - WCET-oriented compilation
- Predictable programming
Motivations for compiler support

- Compiler knows code representation
  - High level (source code)
  - Low level (generated instructions)
- The compiler optimizes code
  - Could optimize the worst-case path
  - Could generate time-predictable code (cache locking, scratchpad memories, single path code, etc.)
- The compiler can generate information relevant for WCET analysis
  - Flow information
  - Compiler optimizations (unrolling factor, etc.)

Compilers and predictable programming

- Compilation and WCET analysis
  - Optimizations and WCET estimation
  - WCET-oriented compilation
- Predictable programming
Traceability of flow information

Motivation

- Optimizations transform the code / data layout
  - Flow information obtained at the source code level may not be valid anymore
- Need for traceability of flow information
- Similar challenge as source-level debugging
  - Mapping of code locations
  - Mapping of variables

Example: loop unrolling
- Reduces loop overhead
- Increases instruction-level parallelism

Original code

```plaintext
for (i=0; i<2*n; i++)
  // MAXITER(100)
  { body(i); }
```

Optimized code

```plaintext
for (i=0; i<2*n; i+=2)
  // MAXITER?
  { body(i); body(i+1); }
```

(NB: shown at source code-level, performed at IR level)
Traceability of flow information

Motivation

Example: loop unrolling

Original code

```c
for (i=0; i<2*n; i++)
// MAXITER(100)
{
    body(i);
}
```

Optimized code

```c
for (i=0; i<2*n; i+=2)
// MAXITER = 50
{
    body(i);
    body(i+1);
}
```

Original flow information still **safe**

Flow information is **pessimistic**

---

Traceability of flow information

Motivation

Example: loop re-rolling

Original code

```c
for (int i = 0; i < 3200; i += 5) {
    a[i] += alpha * b[i];
    a[i + 1] += alpha * b[i + 1];
    a[i + 2] += alpha * b[i + 2];
    a[i + 3] += alpha * b[i + 3];
    a[i + 4] += alpha * b[i + 4];
}
```

Optimized code

```c
for (int i = 0; i < 3200; ++i) {
    a[i] += alpha * b[i];
}
```
Traceability of flow information

Motivation

Example: loop re-rolling

for (int i = 0; i < 3200; i += 5) {
    a[i] += alpha * b[i];
    a[i + 1] += alpha * b[i + 1];
    a[i + 2] += alpha * b[i + 2];
    a[i + 3] += alpha * b[i + 3];
    a[i + 4] += alpha * b[i + 4];
} // Maxiter 640

Original code
- Original flow information clearly unsafe

for (int i = 0; i < 3200; ++i) {
    a[i] += alpha * b[i];
} // Maxiter 3200

Optimized code

Vectorized code (MMX) – padd on 128 bits
rax incremented by 16 at every iteration
Ints on 4 bytes: Iteration count = 1024

Error prone
Traceability of flow information
Tracing what the compiler does

- Matching between source code and binary code
  - Debug information
  - Not guaranteed exact (best effort)
- Graph matching to match CFG between source code and binary code
  - May work in simple cases
  - Match may be found with incorrect conclusions
    - Example: re-rolling
      - Structure of CFG identical,
      - Flow information has changed, and is unsafe

Traceability of flow information
Tracing what compiler does

- gcc -O2 -Q --help=optimizers toto.c
  - Outputs which optimizations are enabled at -O2
    -ftr-ex-switch-conversion [disabled]
    -ftr-ex-tail-merge [disabled]
    -ftr-ex-ter [disabled]
    -ftr-ex-vec-loop-version [disabled]
    -ftr-ex-vectorize [disabled]
    -ftr-ex-vrpt [disabled]
    -ftr-ex-data [disabled]
    -ftr-ex-all-loops [disabled]
- No ordering information
- Enabled does not mean applied
Traceability of flow information
Tracing what compiler does

- gcc -O2 -fdump-tree-all toto.c
  - Transformed code after each transformation (gimple, gcc IR)
  - Gcc specific
  - Hard to find easy to exploit information (e.g. unrolling factor for loops)

Workaround: flow annotations

- Example of aiT (non exhaustive)
  - Targets for indirect jumps or calls
    - instruction 0xc0f8 calls "disable";
    - instruction 0x9024 calls 0x4a, "go", "munch";
    - instruction 0x6709 branches to 0xc0f8, 0x9024 ;
  - Relations between numbers of executions
    - flow sum 3 ("f") + 9 (0x8100) <= 100;
  - Loop bounds
    - loop "prime" max 10;
  - Recursion bounds
    - recursion "fac" max
Workaround: flow annotations

- Example of aiT (non exhaustive)
  - Possible values of variables
    - condition 0xc0f8 is always true;
    - snippet 0x8100 is never executed;
  - Boundaries of memory accesses
    - instruction 0xc0f8 accesses 0x10000 .. 0x20000;

- Notes
  - Identifiers of loops / blocks / functions
    - Addresses in binary
    - Symbols in binary symbol table
  - Error if an annotation contradicts automatic flow fact estimation, else trusts the user

Workaround: flow annotations

- Remarks
  - Helps tightening WCETs
  - Need expertise from the user
  - Annotations may be erroneous
  - Manually annotating the code is labor-intensive
  - Source of errors: addresses may change after re-compilations
    - Use symbols already in symbol table
    - Insert asm labels in source code
  - The user has to know optimizations (if any)
  - Not all annotations are artificial (*operation modes – ranges of initial values*)
Traceability of flow information
Tracing what the compiler does

- Particular case of polyhedral transformations
  - Polyhedral theory help obtaining safe and precise flow information

```
for(int i=0;i<640;i++) {
  for(int j=0;j<480;j++) {
    asm("lbl1:");
    #pragma lbl1 flow=307200
    {
      S0: A[i][j]=…;
    }
    if(i>=j) {
      asm("lbl2:");
      #pragma lbl2 flow=153280
      {
        S1: A[i][j]=…;
      }
    }
  }
}
```

- Augmenting the compiler with traceability of flow information: research on the topic
  - Kirner’s thesis: in gcc 2.7.2 (released in 1995)
  - Li’s thesis: in LVMV 3.5 (released in 2014)
    - Flow information for each basic block / loop
    - For every optimization
      - Formulas to calculate the new flow information
      - In case not possible (too complex, new optimization), discard flow information
Traceability of flow information
Tracing what the compiler does

- Li’s thesis

Examples from Li’s thesis: loop unswitch

\[
\begin{array}{c|c}
\text{Original source code} & \text{Optimized code} \\
\hline
\text{for } i=0; i<\alpha; i++ \\
\text{\quad } a[i]=b[i+1]; & \text{if } (z=0) \\
\text{\quad } \text{if } (x=0) & \text{\quad } \text{else} \\
\text{\quad } a[i]=a[i]+1; & \text{\quad } a[i]=a[i]+2; \\
\text{\quad } \text{else} \\
\end{array}
\]

\[L_X < X_{\min}, X_{\max}> \rightarrow L_X < X_{\min}, X_{\max}, L_Y < X_{\min}, X_{\max} >
\]

\[f_B \rightarrow f_B + f_C
\]

\[f_C \rightarrow f_C + f_D
\]

\[f_D \rightarrow f_B + f_C
\]

(c) CFG of loop unswitch
Traceability of flow information
Tracing what the compiler does

Examples from Li’s thesis: loop peeling

Examples from Li’s thesis: experimental results

Figure 4.1 – Impact of optimizations (-03) on WCET. The y-axis represents the WCET with optimizations, normalized with respect to the WCET without optimization (-00)
Traceability of flow information
Tracing what the compiler does

- Remarks from Li’s thesis
  - Labor-intensive:
    - Many tricky details (e.g. number of iterations not multiple of unrolling factor)
    - Has to be applied to all optimizations
    - But has to be done only once per compiler
  - Has to be checked / re-done at every compiler version
    - Painful in case of change of compiler structure
  - Some optimizations are hidden in the back-end
  - Never integrated in production compilers

Traceability of flow information
Compiler wish list

- Provide information on code structure
  - Source code level or binary code level
- Provide flow information
  - Loop bounds (min/max)
  - Bounds of recursion depth
  - Targets of calculated jumps / calls
- Provide mapping of source code annotations to object code
  - Formulas to calculate new bounds at binary level, or
  - Invalidation of annotations provided at source code level (ex: code or loop removed)
Compilers and predictable programming

- Compilation and WCET analysis
  - Optimizations and WCET estimation
  - WCET-oriented compilation
- Predictable programming

WCET-oriented optimizations

- Objective
  - Optimize code to reduce WCET estimate
- Class of techniques:
  - Optimizations along worst-case path (WCEP)
  - Separate optimization problem
WCET-oriented optimizations

- Optimizations along worst-case path
  1. Estimate WCET and use frequency information to estimate (one) longest execution path (WCEP)
  2. Optimize along WCEP
- Examples of WCEP-oriented optimizations:
  - Superblock formation
  - Scratchpad memory management
  - Cache locking
  - Etc, …

Example: superblock formation (Zhao et al, 2001)

- Architecture: SC100 (very simple embedded architecture)
  - 5-stages pipeline
  - delay when a conditional branch is taken
  - delay when the target of a branch is misaligned
- Principle
  - Identify worst-case path
  - Re-organize code layout to eliminate branch delays: superblock formation
WCET-oriented optimizations

Super block formation

superblock: ...block with single entry point

...worst case path

increase of code size but reduction of WCET

Issues with WCET-oriented optimizations
- Optimizations may change the worst-case path
WCET-oriented optimizations

- Issues with WCEP-oriented optimizations
  - Optimizations may change the worst-case path

```
Issues with WCEP-oriented optimizations:
  - Solutions
    - Iterative optimization: re-evaluation of WCET during the optimization process
```

Credits: Peter Puschner
WCET-oriented optimizations

- Optimizations with no explicit used of WCEP
  - Example: WCET-directed prefetching in SPM-based systems

- SPM can fit x or y
- Static solution will choose x or y for all execution
- Solution:
  - Dynamic
  - Overlap transfer and calculation (asynchronous prefetching)
WCET-oriented optimizations

- Example: WCET-directed prefetching in SPM-based systems: allocation problem with ILP
  - Feasible solutions
    - For all regions, allocated data fit (including addresses) into scratchpad
  - Approach
    - Find feasible solutions
    - Run WCET analysis to estimate overlaps and determine the best allocation
    - Exploration of feasible solutions: genetic algorithm
  - No issue if stability of worst-case path, but still some exploration (genetic algorithm)

Compilers and predictable programming

- Compilation and WCET analysis
  - Optimizations and WCET estimation
  - WCET-oriented compilation
- Predictable programming
WCET analysis

- Many different paths and execution times
- Non trivial analysis of infeasible paths
- Complex (and pessimistic) analysis of hardware timing (and WCET analysis in general)

Predictable programming goals

- Traditional programming
  - Good average-case performance
- Hard real-time programming
  - WCET must be computable
  - WCET estimate must be as low as possible
Traditional programming

- Goal: good average-case performance
- Strategy
  - Test input data to favor frequent cases
- Negative effect on WCET estimate
  - Costs for identifying frequent situations (inputs)
  - Complexity is unevenly distributed over input data sets
  - Savings of frequent situations may paid back in rare situations
  - Branching costs
    - Direct and indirect (merging of system states for low-level analysis)

```c
void bubble (int a[]) {
  for (i=N-1;i>0;i--) {
    for (j=1;j<i;j++) {
      if (a[j-1]>a[j]) { /* Swap */
        i = a[j]; a[j]=a[j-1]; a[j-1]=t;
      }
    }
  }
}
```
Predictable programming

- Objectives
  - Enable performance predictability
  - Enable simple WCET estimation (thus less pessimistic)
  - Raw performance is only a secondary objective

Example of predictable programming: single-path paradigm

- Produce code free from input-data dependent control decisions
- Use predication (data-dependencies) instead of control dependencies
- Hardware support: predicated instructions
  - Unconditional fetch of instruction
  - If predicate is true, normal execution, else no modification of the processor state
- Control flow orientation → data flow focus
Single-path paradigm

```
void bubble (int a[]) {
    for (i=N-1;i>0;i--) {
        for (j=1;j<=i;j++) {
            s = a[j-1];
            t = a[j];
            s<=t: a[j-1] = s;
            s<=t: a[j] = t;
            s>t: a[j]=t;
            s>t: a[j]=s;
        }
    }
}
```
**Single-path paradigm**

<table>
<thead>
<tr>
<th></th>
<th>Nb Paths</th>
<th>Min E.T.</th>
<th>WCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional</td>
<td>3628800</td>
<td>675</td>
<td>810</td>
</tr>
<tr>
<td>Single-path</td>
<td>1</td>
<td>972</td>
<td>972</td>
</tr>
</tbody>
</table>

- **Benefits**
  - Path analysis is trivial (one single path)
  - WCET estimation is simplified
    - Two executions starting from the same cache state have identical hit/miss sequences in icaches

**Single-path transformation rules**

- Only constructs with input-dependent control flow are transformed (the rest of the code remains unchanged)
- Two steps:
  - Data-flow analysis: mark variables and conditional constructs that are input dependent (predicate ID(var))
  - Actual transformation of input-dependent constructs into predicated code
Single-path transformation rules

- Recursive transformation function based on syntax tree:
  \[ \text{SP}[[p \mid] \sigma \delta] \]
- With:
  - \( p \): code construct to be transformed into single path
  - \( \sigma \): inherited precondition from previously transformed code (initial value = T)
  - \( \delta \): counter, used to generate variable names (initial value = 0)

Single-path transformation rules (1)

- Simple statement \( S \)
  \[ \text{SP}[[S \mid] \sigma \delta] \]
- Transformed into: (three cases depending on \( \sigma \))
  - \( \sigma = \text{T} \) : \( S \)  // unconditional
  - \( \sigma = \text{F} \) :  // no action
  - else \( (\sigma) S \)  // predicated
**Single-path transformation rules (2)**

- Sequence $S = S_1 ; S_2$
  
  $\text{SP}[[ S_1 ; S_2 ]] \sigma \delta$

- Transformed into:
  
  - $\text{guard}_\delta := \sigma ;$
  - $\text{SP}[[ S_1 ]] <\text{guard}_\sigma> <\delta + 1> :$
  - $\text{SP}[[ S_2 ]] <\text{guard}_\sigma> <\delta + 1> :$

**Single-path transformation rules (3)**

- Alternative: $S = \text{if} \, \text{cond} \, \text{then} \, S_1 \, \text{else} \, S_2 \, \text{endif};$
  
  $\text{SP}[[ \text{if} \, \text{cond} \, \text{then} \, S_1 \, \text{else} \, S_2 ]] \sigma \delta$

- Transformed into:
  
  - if $ID(\text{cond})$
    
    - $\text{guard}_\delta := \text{cond} ;$
    
    - $\text{SP}[[ S_1 ]] <\sigma \land \text{guard}_\sigma> <\delta + 1> :$
    
    - $\text{SP}[[ S_2 ]] <\sigma \land \top \text{guard}_\sigma> <\delta + 1> :$
  
  - Else
    
    - if cond then $\text{SP}[[ S_1 ]] \sigma \delta :$
    
    - else $\text{SP}[[ S_2 ]] \sigma \delta :$

endif
Single-path and timing

- Same instruction sequence fetched: good basis for invariable timing
- However:
  - Data-dependent execution times cause execution-time jitter
  - Starting from a cache state may cause
    - Different access times to instruction and data, and then variable execution times

Single-path: enforcing invariable timing

- Enforce invariable access times for data objects
- Enforce invariable execution times of instrs regardless of operand values (e.g. mul)
- Enforce invariable timing of predicated instructions
  - If predicate is false, execute but does not commit

- Note: requires hardware modifications
Performance of single path code

- Execution times of input-dependent alternatives sum up due to serialization
  - Execution times are long if the code is strongly input-dependent

Single-path paradigm

- Limitations
  - Programmer or compiler-directed: modification of programming habits/toolchain
  - WCET may be higher than with traditional programming
  - Needs support for predication
  - Invariable timing needs hardware modifications

- Can be used to reduce the number of paths, even if not reduced to one