Enforcing Constant Execution Times for Software

Peter Puschner
TU Vienna

EUROPEAN COOPERATION IN SCIENCE AND TECHNOLOGY

T-CREST
TIME-PREDICTABLE MULTI-CORE ARCHITECTURE FOR EMBEDDED SYSTEMS

Vienna, IFIP WG 1.9/2.15 Mtg.

July, 2014

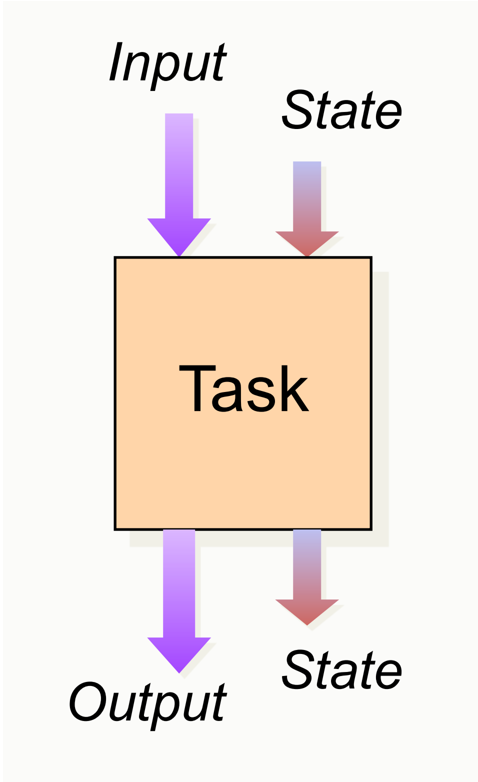
Contents

- Problem and possible solutions
- Code generation
- Properties

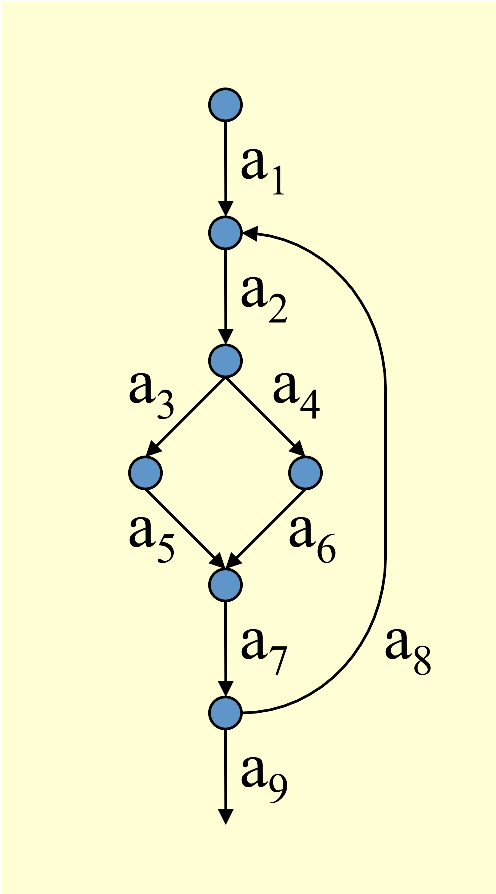
Simple Task

- Inputs available at start
- Outputs ready at the end
- No blocking inside
- No synchronization or communication inside
- Execution time variations only due to differences in
  - inputs
  - task state at start time

  (no external disturbances)



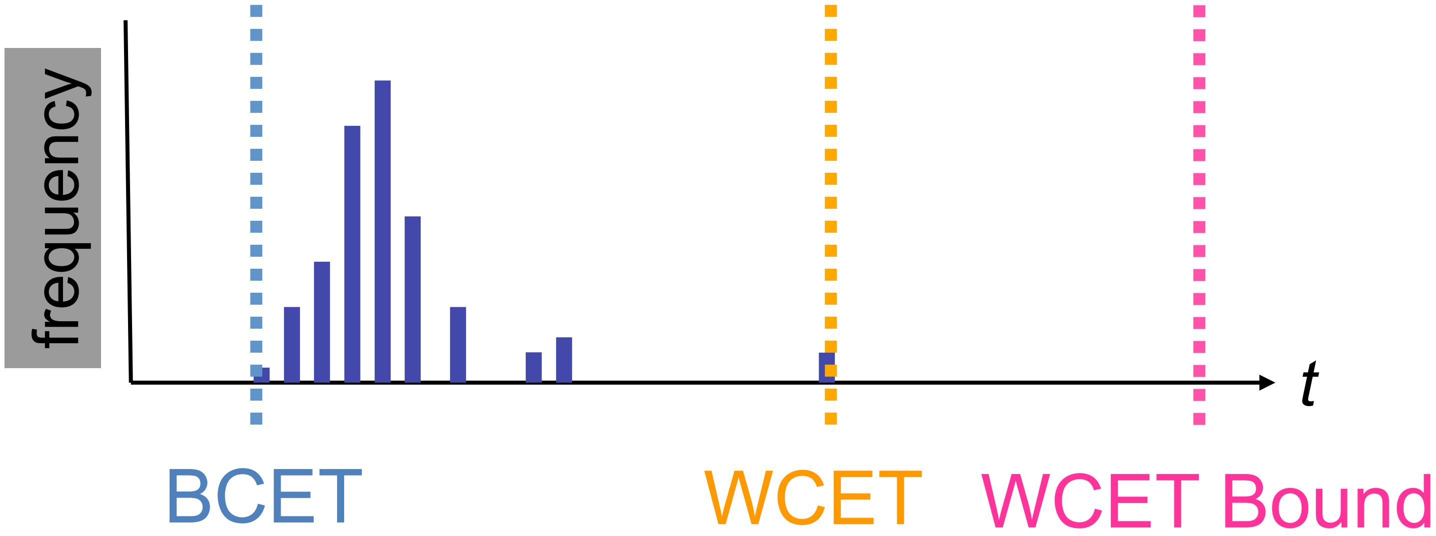
Task Execution Time



1. Sequence of actions (execution path)
2. Duration of each occurrence of an action on the path

Actual path and timing of an execution depends on task inputs (incl. state)

WCET Analysis



Many different execution times

- Non-trivial analysis of (in)feasible paths
- Complex modeling of task timing on hardware

Task Timing Goals

Prioritized goals:

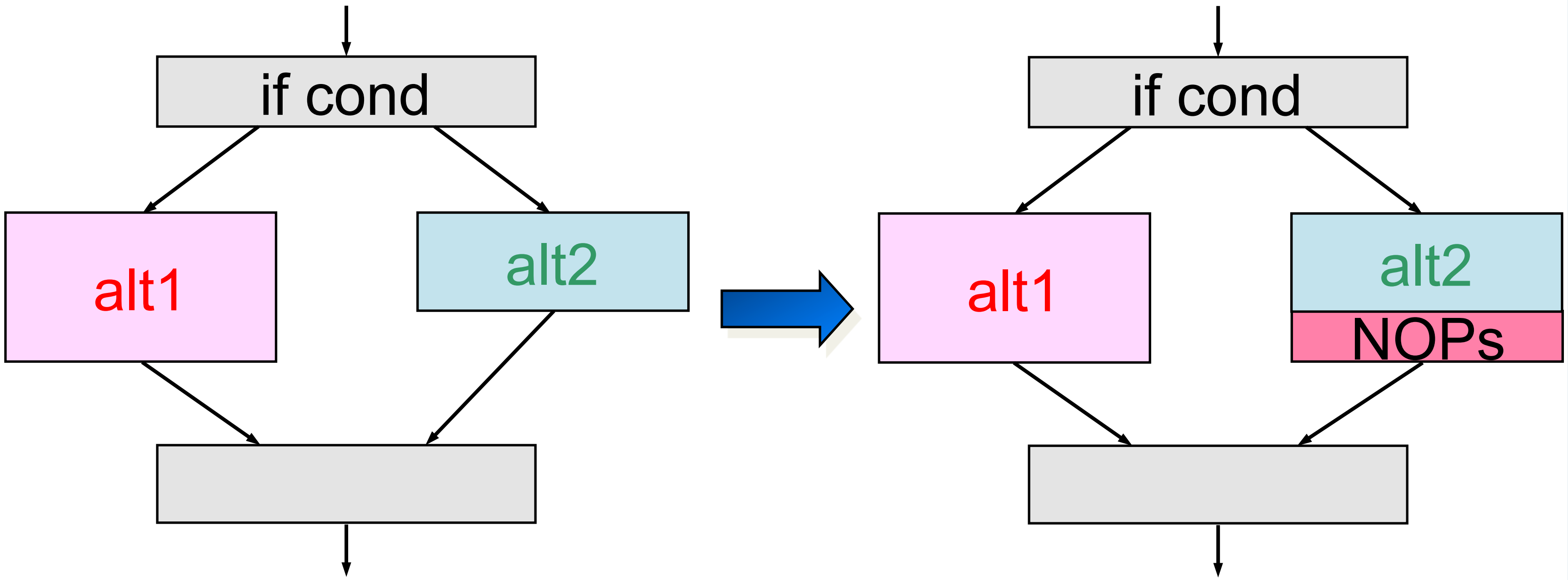
1. Temporal predictability/stability first
2. Performance second

⇨ Strategy: Get the overall timing constant:

- Instruction padding
- Delay termination until end of WCET-bound time budget
- Single-path code transformation

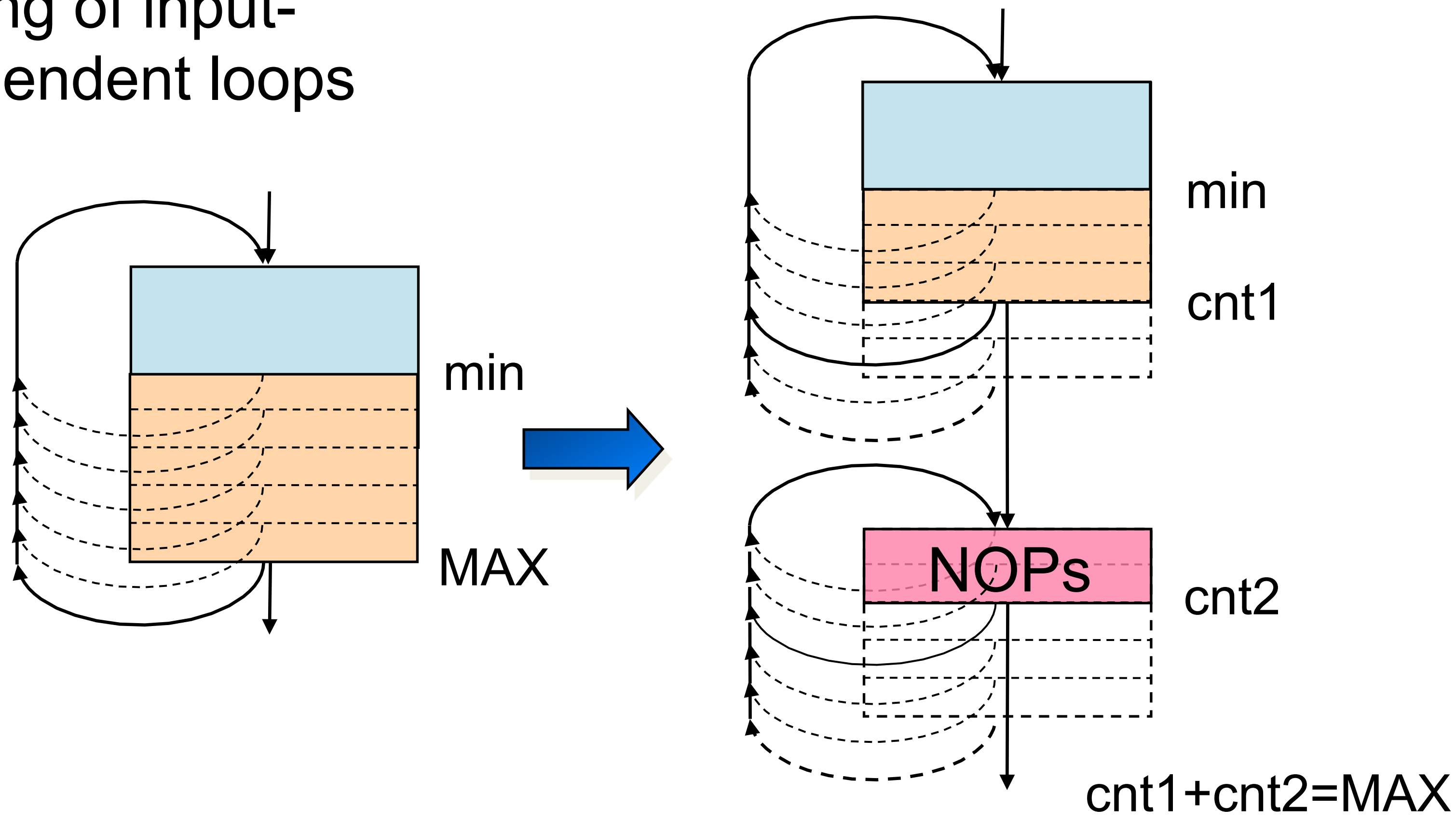
Instruction Padding

Idea: add NOPs to make execution times of alternatives with input-dependent conditions equal



Instruction Padding

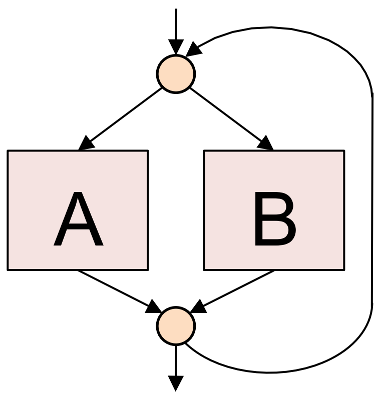
Padding of input-dependent loops



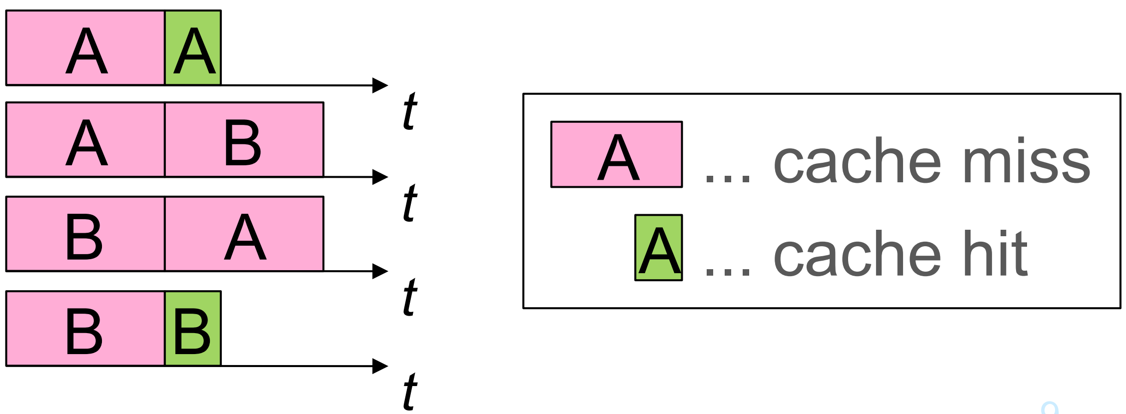
Instruction Padding Problem

Duration of actions depends on the execution history

→ we cannot remove execution-time variations from branching code



Caching: loop with instructions A and B, executing two iterations



Instruction Padding

Applicable to simple architectures: execution times of instructions are not state dependent

- WCET bound of transformed code ≈ original WCET bound
- Code-size increase

Constant Exec. Time Using a Delay

Strategy:

1. Def: task time budget = computed WCET bound
2. Insert delay(until end of time budget) at end of task

Problem: bad resource utilization due to

- Pessimism in path analysis (all architectures)
- Pessimism in hardware modelling (complex arch.)

⇨ Full flavour of WCET analysis problems ...

Time-Predictable Single-Path Code

Don‘t let the environment dictate

- Sequence of actions
- Durations of actions

Take control decisions offline!!!

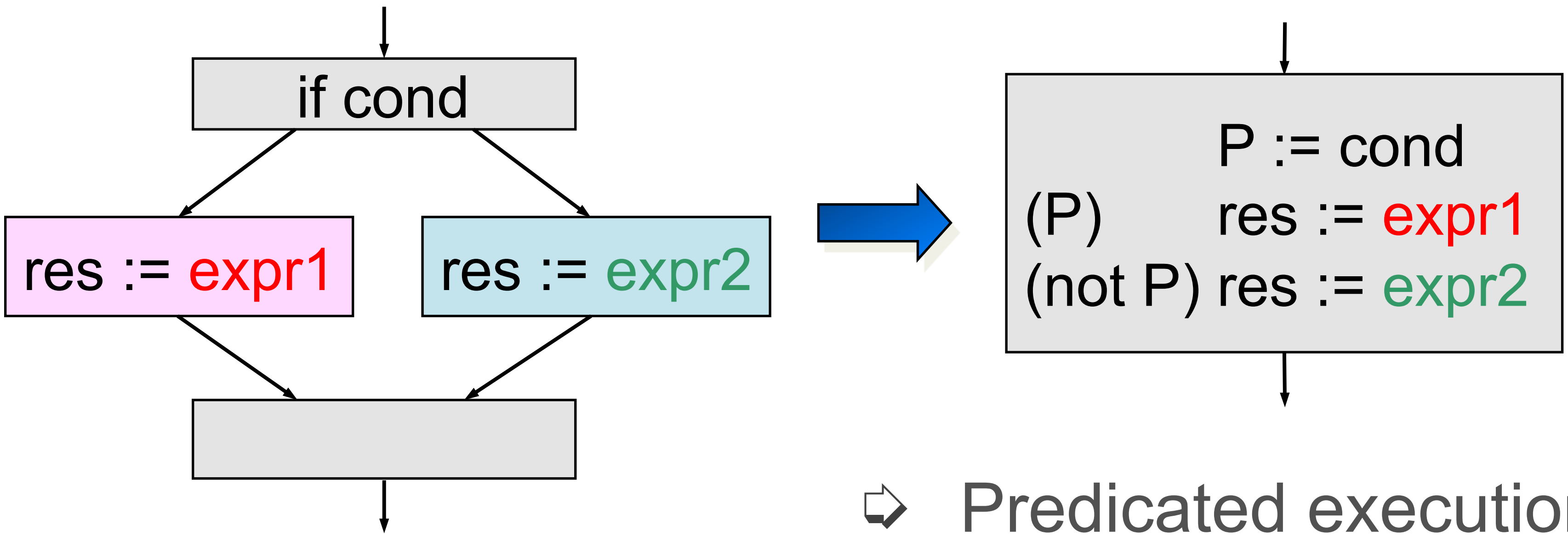
Control sequencing of all actions instead of being controlled by the environment (data, interrupts)

→ Single-path code:

- no input-data dependent branches
- predicated execution (poss. with speculation)
- control-flow orientation → data flow focus

Remove Data Dependent Control Flow

- Hardware with invariable timing
- Single-path conversion of code



➯ Predicated execution

Branching vs. Predicated Code

Code example: **if** rA < rB **then** swap(rA, rB);

Branching code

```
        cmplt  rA, rB
        bf     skip
        swp    rA, rB
skip:
```

Predicated code

```
        predlt Pi, rA, rB
(Pi)    swp    rA, rB
```

How to Generate Single-Path Code

Introduce the transformation in two steps:

1. Transformation model
   set of rules
   assumes full predication
2. Implementation details
   adaptation for platforms with partial predication

Single-Path Transformation Rules

Only constructs with input-data dependent control flow are transformed, the rest of the code remains unchanged → two steps:

1. data-flow analysis: mark variables and conditional constructs that are input dependent
   → result available through predicate *ID*(...)

2. actual transformation of input-data dependent constructs into predicated code

Single-Path Transformation Rules

Recursive transformation function based on syntax tree:

SP[[p]]$\sigma\delta$

p … code construct to be transformed into single path

σ … inherited precondition from previously transformed code constructs. The initial value of the inherited precondition is 'T' (logical true).

δ ... counter, used to generate variable names needed for the transformation. The initial value of δ is zero.

Single-Path Transformation Rules (1)

simple statement: S

SP[[S]]$\sigma\delta$

⇒

if $\sigma = T$:	S	// unconditional
if $\sigma = F$:		// no action
otherwise:	(σ) S	// predicated (guarded)

Single-Path Transformation Rules (2)

sequence: S = S1; S2

SP[[S1; S2]]σδ

⇒

guard$_\delta$:= σ;
SP[[S1]]⟨guard$_\delta$⟩⟨δ+1⟩ ;
SP[[S2]]⟨guard$_\delta$⟩⟨δ+1⟩

Single-Path Transformation Rules (3)

alternative: S = if *cond* then S1 else S2 endif

SP[[if *cond* then S1 else S2 endif]]$\sigma\delta$

if *ID*(cond): *guard*$_\delta$:= *cond*;
SP[[S1]]$\langle\sigma \wedge guard_\delta\rangle\langle\delta+1\rangle$;
SP[[S2]]$\langle\sigma \wedge \neg guard_\delta\rangle\langle\delta+1\rangle$

otherwise: if *cond* then SP[[S1]]$\sigma\delta$
else SP[[S2]]$\sigma\delta$
endif

Single-Path Transformation Rules (4)

loop: S = while *cond* max *N* times do S1 endwhile

$SP[\![$ while *cond* max *N* times do S1 endwhile $]\!]\sigma\delta$

if *ID*(cond):

end$_\delta$:= *F;* // loop-body-disable flag
for *count*$_\delta$:= 1 to *N* do // "hardwired loop"
  $SP[\![$ if ¬*cond* then *end*$_\delta$:= *T* endif $]\!]\sigma\langle\delta+1\rangle$;
  $SP[\![$ if ¬*end*$_\delta$ then S1 endif $]\!]\sigma\langle\delta+1\rangle$
endfor

Single-Path Transformation Rules (5)

loop: S = while *cond* max *N* times do S1 endwhile

SP[[while *cond* max *N* times do S1 endwhile]]$\sigma\delta$

⇒

if ¬*ID*(cond):

while *cond* max *N* times do
  SP[[S1]]$\sigma\delta$
endwhile

Single-Path Transformation Rules (6)

procedure call: S = proc(act-pars)

SP⟦ proc(act-pars) ⟧$\sigma\delta$

⇒

$$\begin{cases} \text{if } \sigma = T: & \text{proc(act-pars)} \\ \text{otherwise:} & \text{proc-sip}(\sigma, \text{act-pars}) \end{cases}$$

Single-Path Transformation Rules (7)

procedure definitions: proc p(form-pars) S end

SP⟦ proc p(form-pars) S end ⟧σδ

⇒

proc p-sip(*precond-par*, form-pars)
    SP⟦ S ⟧⟨*precond-par* ⟩⟨0⟩
end

HW-Support for Predicated Execution

Predicate registers

Instructions for manipulating predicates
(define, set, clear, load, store)

Predication support of processors

- Full predication
  execution of all instructions is controlled by a predicates
- Partial predication
  limited set of predicated instructions
  (e.g., conditional move, select, set, clear)

Implications of Partial Predication

Speculative code execution

- unconditional execution of non-predicated instructions
- the results are stored in temporary variables;
- subsequently, predicates determine which values of temporary variables are further used

```
                 src1 := expr1
                 src2 := expr2
(pred)           cmov dest, src1
(not pred)       cmov dest, src2
```

Cave: speculative instructions must not raise exceptions!
(e.g., div. by zero, referencing an invalid memory address)

Fully vs. Partially Predicated Code

Original code: **if** src2 $\neq$ 0 **then** dest := src1/ src2;

Fully predicated code:

Pred := (src2 $\neq$ 0)

(Pred) div dest, src1, src2

Fully vs. Partially Predicated Code (2)

Original code: **if** src2 ≠ 0 **then** dest := src1/ src2;

↓

Partially predicated code, first attempt:

```
                Pred := (src2 ≠ 0)
                div    tmp_dst, src1, src2
(Pred)          cmov   dest, tmp_dst
```

may raise an exception on division by zero

Fully vs. Partially Predicated Code (3)

Original code: **if** src2 ≠ 0 **then** dest := src1/ src2;

Partially predicated code:

if src2 equals 0, then replace it by a safe value (e.g., 1) to avoid division by zero

```
                Pred := (src2 ≠ 0)

                mov    tmp_src, src2
(not Pred)      cmov   tmp_src, $safe_val
                div    tmp_dst, src1, tmp_src
(Pred)          cmov   dest, tmp_dst
```

“Minimal” Predicated-Exec. Support

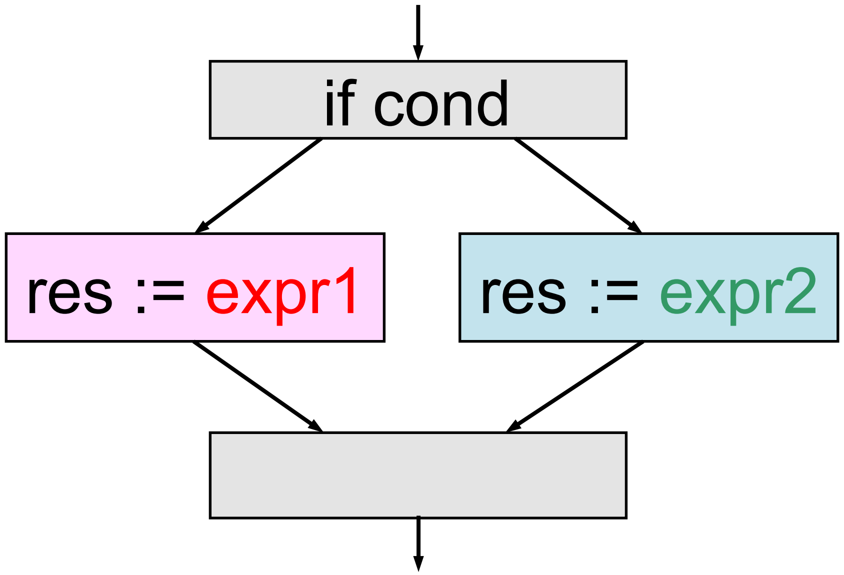
Conditional Move instruction:

mov*CC destination*, *source*

Semantics:

if *CC*
then *destination* := *source*
else no operation

If-conversion with conditional move



t1 := *expr1'*
t2 := *expr2'*
test *cond*
movT res, t1
movF res, t2

avoid side effects!

Emulation of conditional move

In architectures without predicate support, conditional moves can be emulated with bit-mask operations

Example: **if** (cond) x=y; **else** x=z;

```
t0 = 0 – cond;      // fat bool: 0..false, -1..true
t1 = ~t0;           // bitwise negation (fat bool)
t2 = t0 & y;
t3 = t1 & z;
x = t2 | t3;
```

assumption: the types of all values have the same size

Example

Bubble sort: input array a[SIZE]

```
for(i=SIZE-1; i>0; i--)
{
 for(j=1; j<=i; j++)
 {
   if (a[j-1] > a[j])
   {
      t = a[j];
      a[j] = a[j-1];
      a[j-1] = t;
   }
 }
}
```

```
for(i=SIZE-1; i>0; i--)
{
 for(j=1; j<=i; j++)
 {
   t1 = a[j-1];
   t2 = a[j];

   (t1>t2): t = a[j];
   (t1>t2): a[j] = a[j-1];
   (t1>t2): a[j-1] = t;
 }
}
```

Single-Path Properties

Every execution has the same instruction trace, i.e., the same sequence of references to instruction memory

Path analysis is trivial – there is only one path

Two executions starting from the same instruction-cache state have identical hit/miss sequences on accesses to instruction memory

Single-Path and Timing

Every execution uses the same sequence (and thus number) of instructions → good basis for obtaining invariable timing

variable, data-dependent instruction execution times cause execution-time jitter

starting from a different memory state may cause different access times to instruction and data memory, and thus variable execution times

Enforcing Invariable Timing

Don‘t let the environment dictate

- Sequence of actions
- Durations of actions

- ⇨ Always start from the same state of instruction cache, pipeline, branch prediction logic, etc.
- ⇨ Enforce invariable access times for data objects
- ⇨ Invariable durations of all processor operations
- ⇨ All interference must be predictable (preemptions)

Invariable Duration of Operations

Processor operations have to be implemented such that they execute in constant time, i.e., independent of operand values (e.g., shift, mul, div)

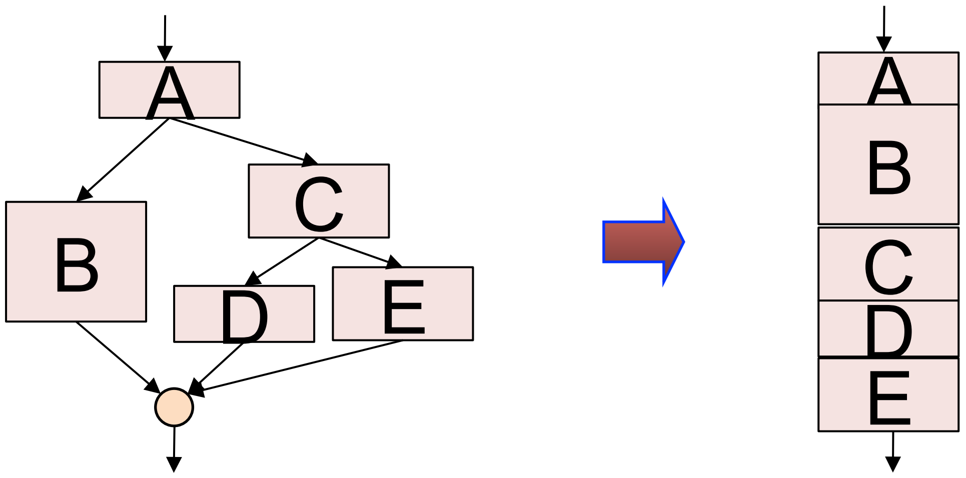
In particular, predicated instructions need to execute in constant time → if predicate is false: allow instruction to execute, but disallow changes of the processor state in the write-back stag

ARM7 experiment: use of strCC-strNCC pairs to obtain constant time despite variable strCC timing

Performance of Single-Path Code

Execution times of input-dependent alternatives sum up due to serialization

⇨ Execution times of single-path code are long if the control flow of its source is strongly input dependent



Performance of Single-Path Code (2)

CPUs with deep pipelines need a number of cycles to re-fill the pipeline after a (mis-predicted) branch

⇨ predicated execution can be cheaper than jumping

⇨ this is where modern compilers/processors use predicated execution to improve performance

Example: Speedup by if-conversion

if rA < rB **then** swap(rA, rB);

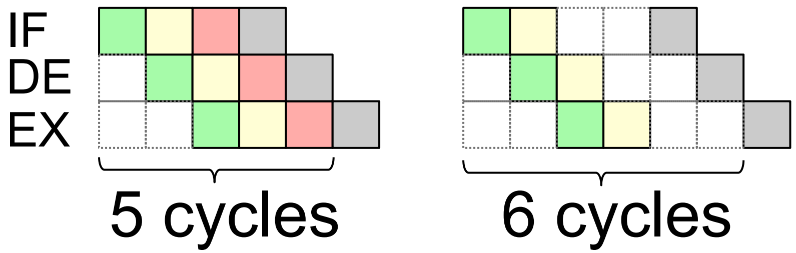
Branching code

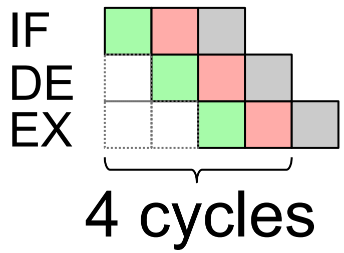
```
          cmplt  rA, rB
          bf     skip
          swp    rA, rB
skip:
```

Predicated code

```
          predlt  Pi, rA, rB
(Pi)      swp     rA, rB
```

Execution in three-stage pipeline





Avoiding Long Execution Times

- Input-invariant coding
  - ➩ avoid classical optimisation patterns that test inputs
  - ➩ do „the same“ for all inputs
  - ➩ programming style, libraries, etc.
- Mode-specific execution times
  - ➩ Make „hidden“ modes visible
  - ➩ Generate single-path code for each mode

Related Issues

- State disruption by dynamic scheduling
  - ➪ Static, table-driven scheduling
  - ➪ Scheduled preemption
  - ➪ Preemption points
- Benefit from path knowledge – we know the future!
  - ➪ Predictable memory hierarchy instead of cache

Summary

Completeness: every piece of code with boundable WCET can be transformed

Transformed code has a single path

WCET analysis is trivial and exact

We know the future

Inputs do not influence timing – execution times do not give clues about what‘s going on

Execution Times