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

- Complier 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.)

3

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

5

Traceability of flow information Motivation

Example : loop unrolling

- Reduces loop overhead
- Increases instruction-level parallelism

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

Original code

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

Optimized code

6

(NB: shown at source code-level, performed at IR level)

Traceability of flow information Motivation

Example : loop unrolling

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

Original code

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

Optimized code

- Original flow information still safe
- Flow information is pessimistic

7

Traceability of flow information Motivation

Example : loop re-rolling

- Reduces code size (cache optimization)
- Exists in LLVM

```
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];
}
```

Original code

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

Optimized code

8

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

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

Optimized code

- Original flow information clearly unsafe

9

Traceability of flow information Motivation

```
int tab1[4096];
int tab2[4096];
int tab_sum[4096];
void sum() {
  int res = 0;
  for (int i=0;i<4096;i++) {
      tab_sum[i] = tab1[i]
     + tab2[i];
  }
}
```

```
400600 <_Z5sumv>:
400600:  …
400608:  movdqa 0x609080(%rax),%xmm0
40060f:  add    $0x10,%rax
400614:  paddd  0x605070(%rax),%xmm0
40061c:  movdqa %xmm0,0x601070(%rax)
400624:  cmp    $0x4000,%rax
40062a:  jne 400608
```

- **Error prone**

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

10

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

11

Traceability of flow information
Tracing what compiler does

- gcc -O2 -Q --help=optimizers toto.c
  - Outputs which optimizations are enabled at -O2

```
-ftree-switch-conversion      [disabled]
-ftree-tail-merge             [disabled]
-ftree-ter                    [disabled]
-ftree-vect-loop-version      [enabled]
-ftree-vectorize              [disabled]
-ftree-vrp                    [disabled]
-funit-at-a-time              [enabled]
-funroll-all-loops            [disabled]
…
```

  - No ordering information
  - Enabled does not mean applied

12

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)

13

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

14

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

15

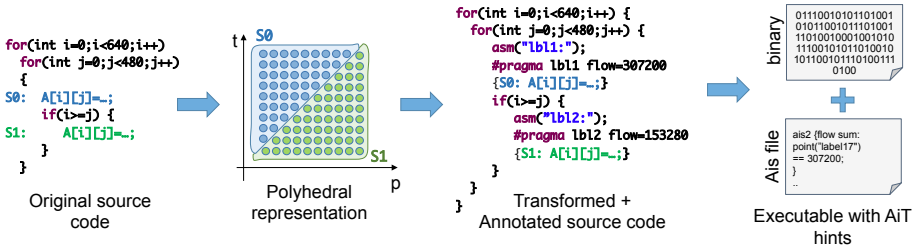
Workaround: flow annotations

- Remarks
  - Helps tightening WCETs
  - Need expertize 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)

16

Traceability of flow information
Tracing what the compiler does

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

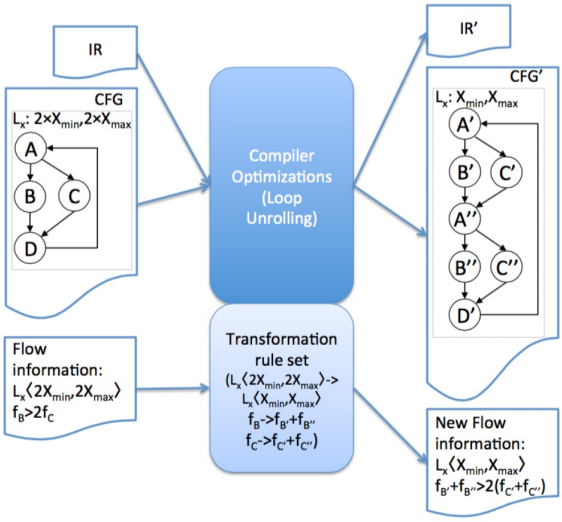


Traceability of flow information
Tracing what the compiler does

- 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



19

Traceability of flow information
Tracing what the compiler does

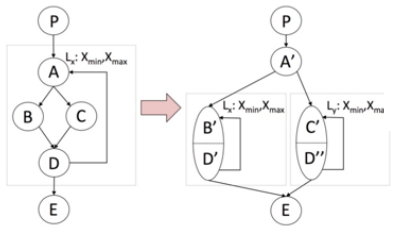
- Examples from Li's thesis: loop unswitch

```
for(i=0; i<n; i++){
  a[i]=b[i];
  if(x>0)
    a[i]=a[i]+1;
  else
    a[i]=a[i]*2;
}
```

(a) Original source code

```
if(x>0)
  for(i=0; i<n; i++){
    a[i]=b[i];
    a[i]=a[i]+1;
  }
else
  for(i=0; i<n; i++){
    a[i]=b[i];
    a[i]=a[i]*2;
  }
```

(b) Optimized code



(c) CFG of loop unswitch

$$L_X\langle X_{\min}, X_{\max}\rangle \rightarrow L_X\langle X_{\min}, X_{\max}\rangle, L_Y\langle X_{\min}, X_{\max}\rangle$$
$$f_B \rightarrow f_{B'D'}$$
$$f_C \rightarrow f_{C'D''}$$
$$f_D \rightarrow f_{B'D'} + f_{C'D''}$$

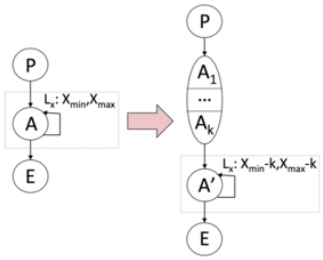
20

Traceability of flow information
Tracing what the compiler does

- Examples from Li's thesis: loop peeling

`for(i=0; i<n; i++)`
`  a[i]=a[i]+c;`
`for(i=1; i<n; i++)`
`  b[i]=b[i]*2;`
	`if(n>0)`
	`  a[0]=a[0]+c;`
	`for(i=1; i<n; i++)`
	`  a[i]=a[i]+c;`
	`for(i=1; i<n; i++)`
	`  b[i]=b[i]*2;`

(a) Original source code	(b) Optimized code



(c) CFG of loop peeling

$$L_X\langle X_{\min}, X_{\max}\rangle \rightarrow L_X\langle X_{\min}-k, X_{\max}-k\rangle$$
$$f_A \rightarrow k \times f_{A_1 \ldots A_k} + f_{A'}$$

21

Traceability of flow information
Tracing what the compiler does

- Examples from Li's thesis: experimental results

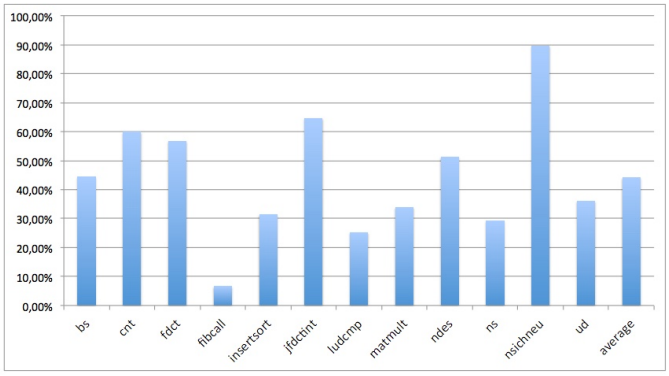


Figure 4.1 – Impact of optimizations (`-O3`) on WCET. The y-axis represents the WCET with optimizations, normalized with respect to the WCET without optimization (`-O0`)

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

23

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)

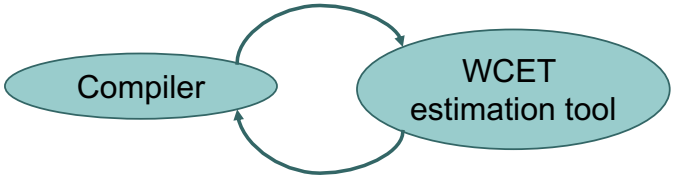
24

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



26

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, …

27

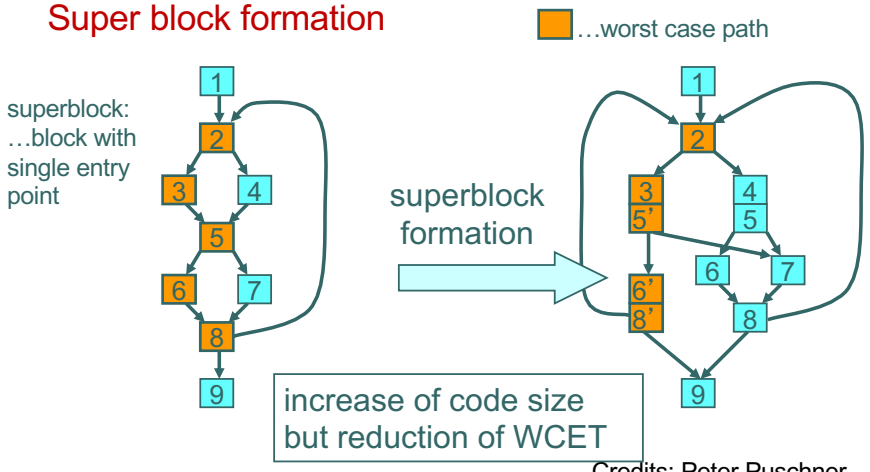
WCET-oriented optimizations

- Example: superblock formation (Zhao et al, 2001)
  - Architecture : SC100 (very simple embedde 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

28

WCET-oriented optimizations

Super block formation

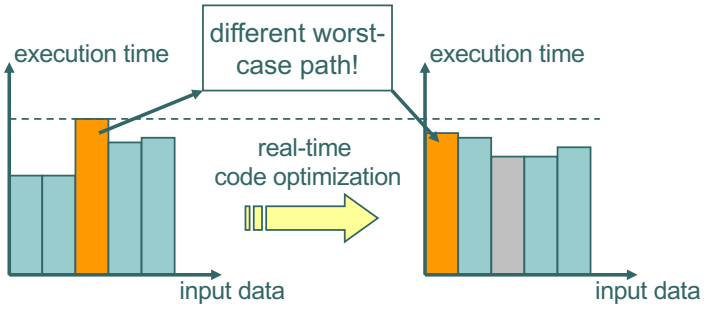


Credits: Peter Puschner

29

WCET-oriented optimizations

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

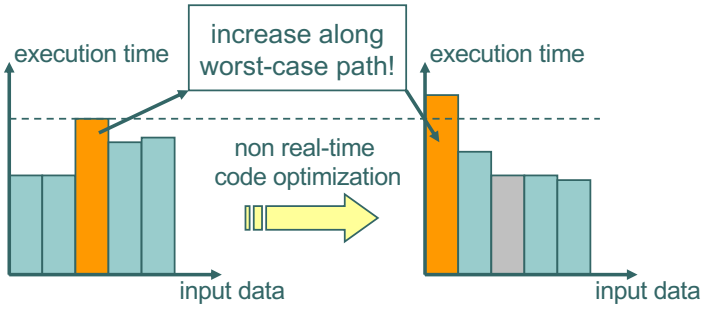


Credits: Peter Puschner

30

WCET-oriented optimizations

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



Credits: Peter Puschner

31

WCET-oriented optimizations

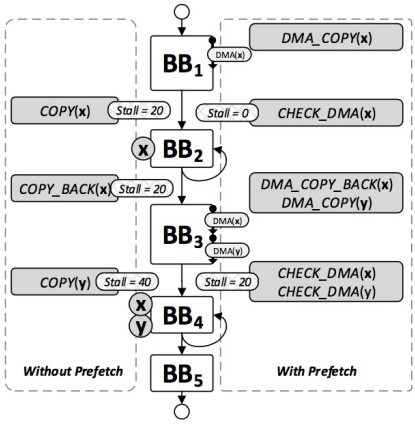
- Issues with WCEP-oriented optimizations: solutions
  - Iterative optimization : re-evaluation of WCET during the optimization process

32

WCET-oriented optimizations

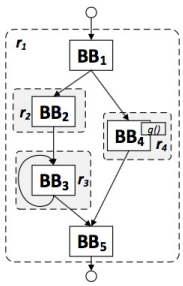
- Optimizations with no explicit used of WCEP
  - Example: WCET-directed prefetching in SPM-based systems

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

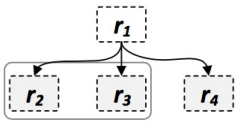


WCET-oriented optimizations

- Example: WCET-directed prefetching in SPM-based systems
  - Single-Entry single-Exit (SESE) regions
  - Analysis variable usage per region



(a) Program CFG



(b) Region tree

WCET-oriented optimizations

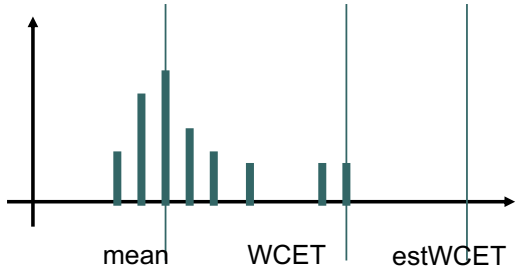
- Example: WCET-directed prefetching in SPM-based systems: allocation problem with ILP
  - Feasible solutions
    - For all regions, allocated data fit (including addresses) into scratchpad
  - Approach
    - Find feasible solutions
    - Run WCET analysis to estimate overlaps and determine the best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

35

Compilers and predictable programming

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

WCET analysis

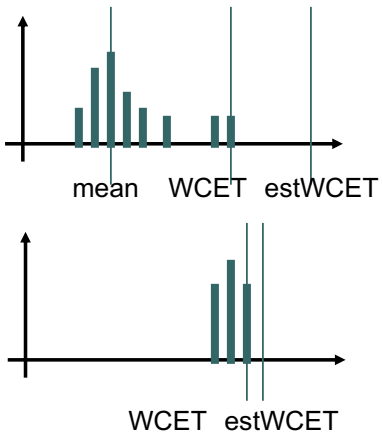


- Many different paths and execution times
- Non trivial analysis of infeasible paths
- Complex (and pessimistic) analysis of hardware timing (and WCET analysis in general)

37

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



38

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)

39

Traditional programming

```
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;
                }
        }
   }
}
```

40

Predictable programming

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

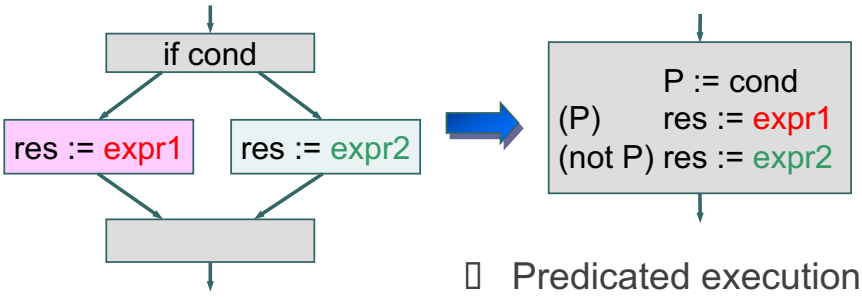
41

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

42

Single-path paradigm



- Predicated execution

Credits: Peter Puschner

43

Single-path paradigm: example

```
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;
        }
    }
}
```

44

Single-path paradigm

	Nb Paths	Min E.T.	WCET
Traditional	3628800	675	810
Single-path	1	972	972

- 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

45

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

46

Single-path transformation rules

- Recursive transformation function based on syntax tree:

  SP[[p]] σ δ
- With:
  - p: code construct to be transformed into single path
  - σ: inherited precondition from previously transformed code (initial value = T)
  - δ: counter, used to generate variable names (initial value = 0)

47

Single-path transformation rules (1)

- Simple statement S

  SP[[S]] σ δ
- Transformed into: (three cases depending on σ)
  - σ = T : S // unconditional
  - σ = F : // no action
  - else (σ) S // predicated

48

Single-path transformation rules (2)

- Sequence S = S1 ; S2

  SP[[S1 ; S2]] σ δ
- Transformed into:
  - $guard_\delta$:= σ ;
  - SP[[S1]] <$guard_\delta$> <δ+1> :
  - SP[[S2]] <$guard_\delta$> <δ+1> :

49

Single-path transformation rules (3)

- Alternative: S = if cond then S1 else S2 endif;

  SP[[if cond then S1 else S2]] σ δ
- Transformed into:
  - if ID(cond)

    $guard_\delta$:= cond ;

    SP[[S1]] < $\sigma \wedge guard_\delta$> <$\delta$+1> :

    SP[[S2]] < $\sigma \wedge \neg\, guard_\delta$> <$\delta$+1> :
  - Else

    if cond then SP[[S1]] σ δ :

    else SP[[S2]] σ δ :

    endif

50

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

51

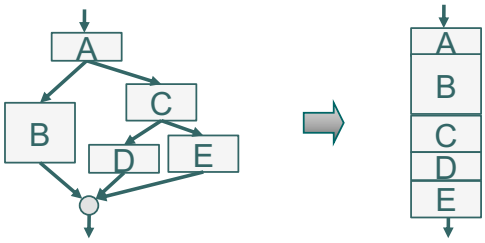
Single-path: enforcing invariable timing

- 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

52

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



Credits: Peter Puschner

53

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

54