FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

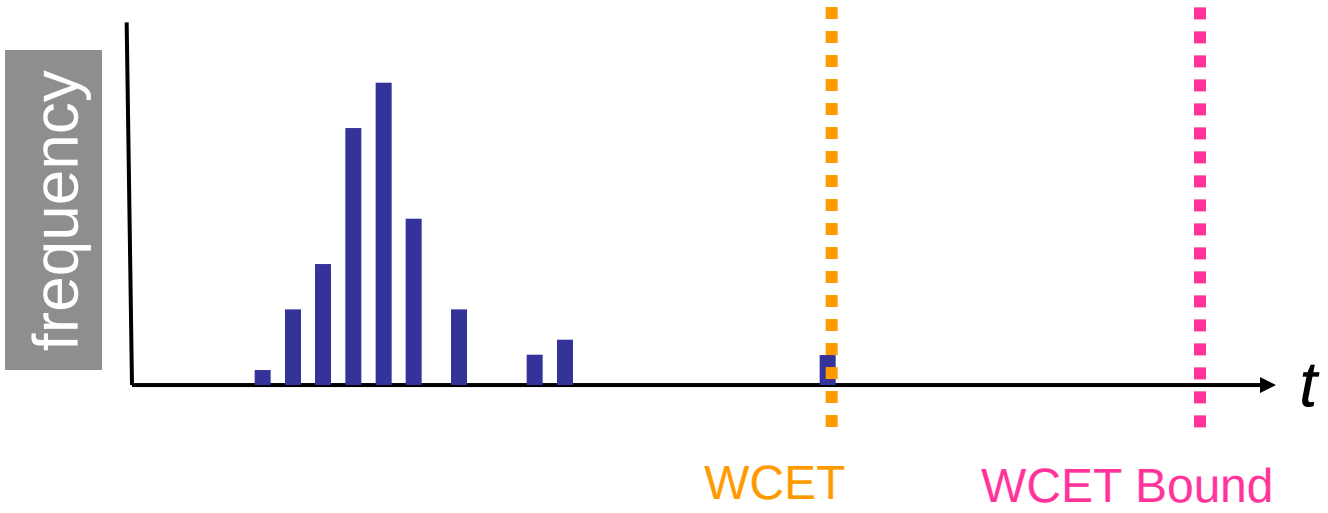
Designing a Time-Predictable Memory Hierarchy for Single-Path Code

Bekim Cilku, Peter Puschner

Outline

- Introduction
- Single-path code
- Transformation Rules
- Memory hierarchy for single-path code
- Prefetching
- Summary

Static Time Analysis



- Software is written to execute fast – different execution paths for different input data;
- Hardware features extend the analysis with state dependencies and mutual interferences;
- State-of-the-art WCET tools are using integrated approach considering all interferences;

Single-Path Code

- Converts all input-dependant alternatives of the code into pieces of sequential code;
- Properties of SP code:
  - Every execution has the same instruction trace, i.e., the same sequence of references to instruction memory;
  - Eliminates control-flow induced variations in execution time – forces the execution time to become constant;
  - Path analysis is trivial – there is only one path;

```
if(a)                beq     a,0,L1        pred_eq  p,a
  x=x+1              add     x,x,1         add      x,x,1 (p)
else                 jump    L2            add      y,y,1 (not p)
  y=y+1          L1:
                     add     y,y,1
                 L2:
```

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

The Single-Path Transformation

Construct S		Translated Construct $\mathcal{SP}[\![S]\!] \sigma\delta$
S	if $\sigma = T$	S
	otherwise (σ)	S
$S_1 ; S_2$		$\mathcal{SP}[\![S_1]\!] \sigma\delta ;$ $\mathcal{SP}[\![S_2]\!] \sigma\delta$
`if` *cond* `then` S_1 `else` S_2	if *ID(cond)*	$guard_\delta$ `:=` *cond* ; $\mathcal{SP}[\![S_1]\!] \langle \sigma \wedge guard_\delta \rangle \langle \delta + 1 \rangle ;$ $\mathcal{SP}[\![S_2]\!] \langle \sigma \wedge \neg guard_\delta \rangle \langle \delta + 1 \rangle$
	otherwise	`if` *cond* `then` $\mathcal{SP}[\![S_1]\!] \sigma\delta$ `else` $\mathcal{SP}[\![S_2]\!] \sigma\delta$

`while` *cond* `max` N `times` `do` S	if *ID(cond)*	end_δ `:= false`
		`for` $count_\delta$ `:= 1 to` N `do begin`
		$\mathcal{SP}[\![$ `if` $\neg$*cond* `then` end_δ `:= true` $]\!] \sigma \langle \delta + 1 \rangle ;$
		$\mathcal{SP}[\![$ `if` $\neg end_\delta$ `then` $S]\!] \sigma \langle \delta + 1 \rangle$
		`end`

	otherwise	`while` *cond* `do` $\mathcal{SP}[\![S]\!] \sigma\delta$

- σ… inherited precondition from previously transformed code constructs;

- δ... counter, used to generate variable names needed for the transformation;

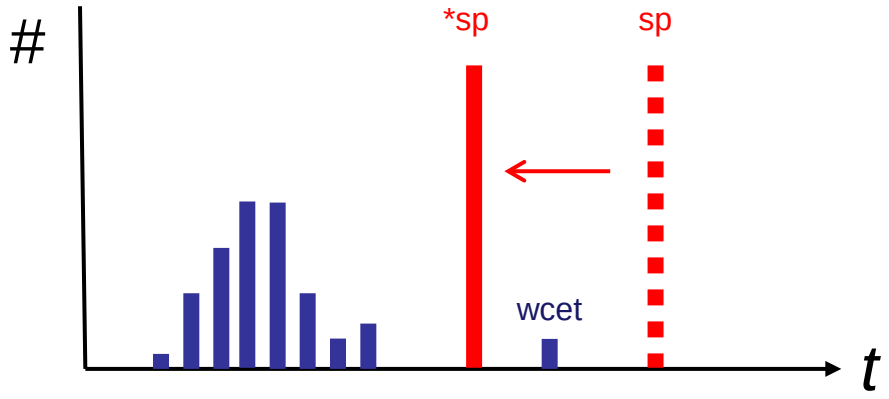
Compositionality of Single-Path Code

- Large programs can be decomposed into smaller segments – each can be analyzed in separation;

- Traditional segment analysis:
  - Wider interfaces between segments – a lot of information gets lost;
  - High complexity for calculating initial states at segment boundaries;
  - Highly pessimistic results;

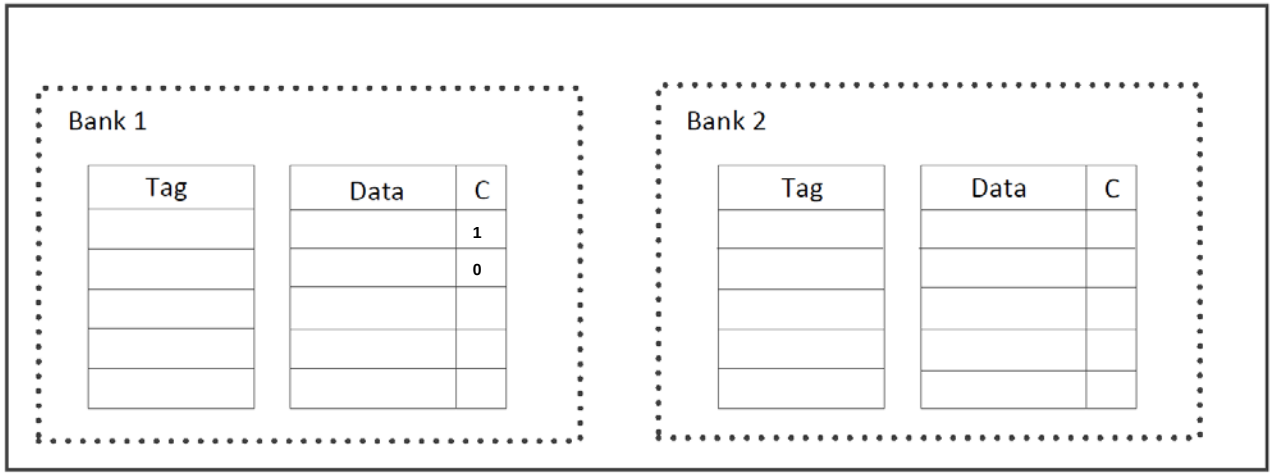
- Single-path segment analysis:
  - Narrow interface between segments - single trace of execution;
  - Easy calculation of initial state;
  - Accurate results;

Reducing Execution Time for Single-Path Code



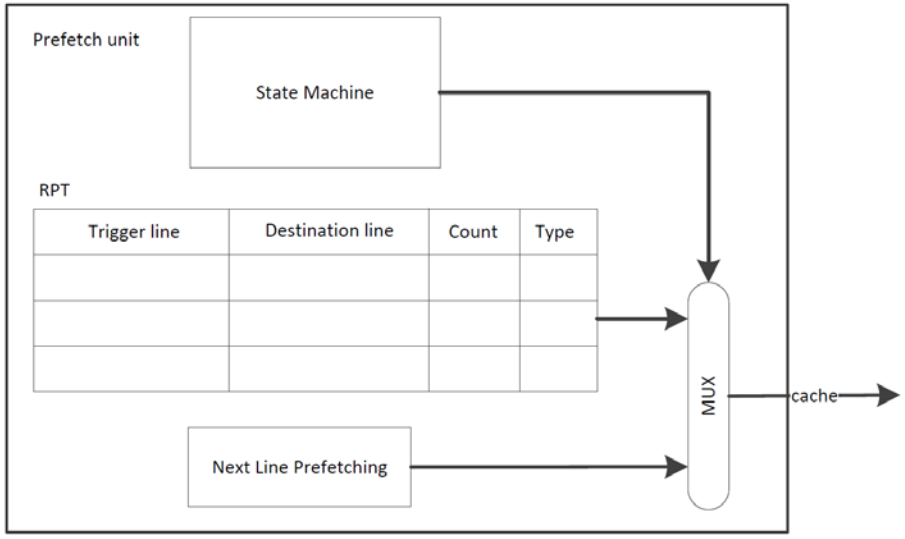
- The long latency of memory accesses is one of the key performance bottlenecks;
- Use “ knowledge of the future” properties of single-path code to control cache content and get a higher benefit from locality principle;
- Add prefetch unit to hide/reduce memory latency;

Cache Memory



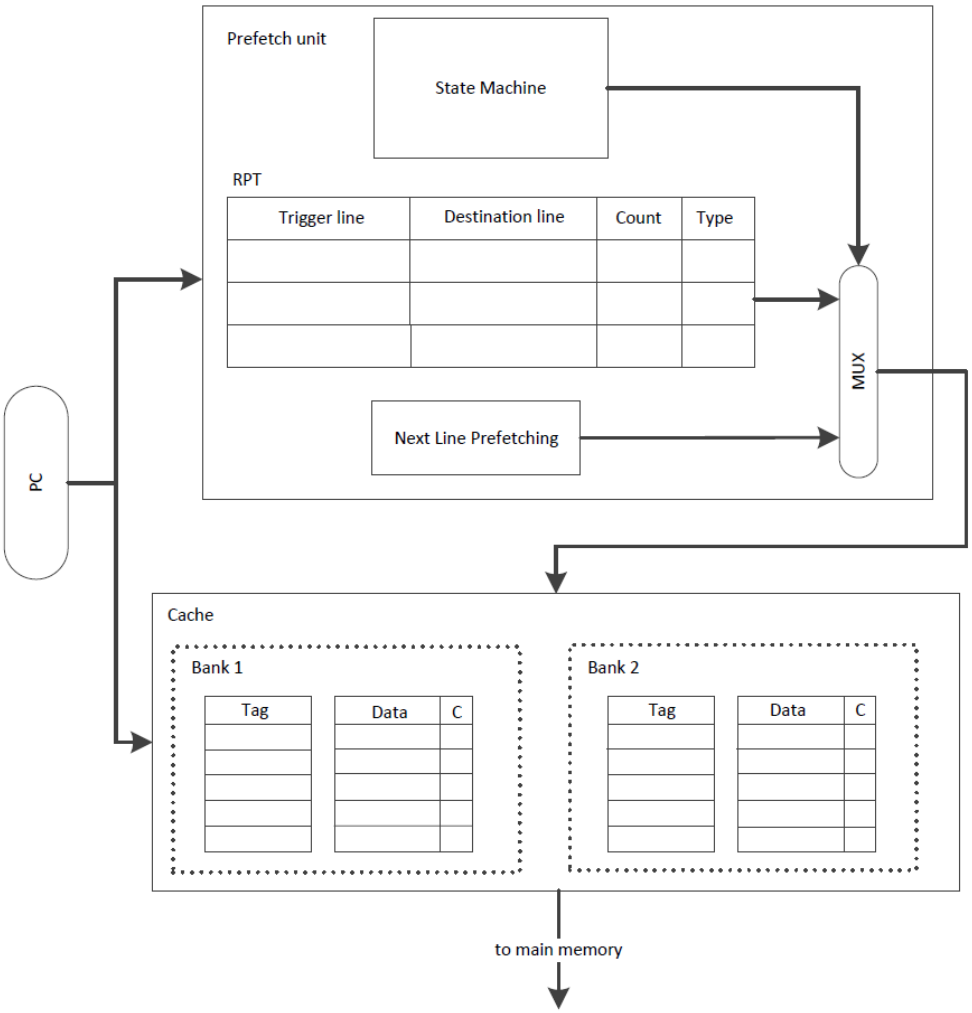
- Two banks allow to overlap the process of fetching with prefetching;
- C-bit prevents replication of requests issued between CPU and prefetcher;
- State of references generated from CPU:
  - No match with *Tag* columns;
  - *Tag* match, C=0;
  - *Tag* match, C=1;

Prefetching Algorithm



- Sequential and non-sequential prefetching:
  - Sequential: simple algorithm;
  - Non-sequential: needs input for target destination;
- RPT entries: *trigger line, destination line, count, type;*
- RPT output has precedence over NLP;

Architecture of the Memory



Summary

- Memory hierarchy that increases performance for single-path code;
- Higher efficiency of cache by better exploitation of the principle of locality;
- Hardware approach – no overhead by extra instructions;
- Dual-bank approach pipelines CPU and prefetch accesses;
- Cache pollution and useless memory traffic is almost zero;

Thank you