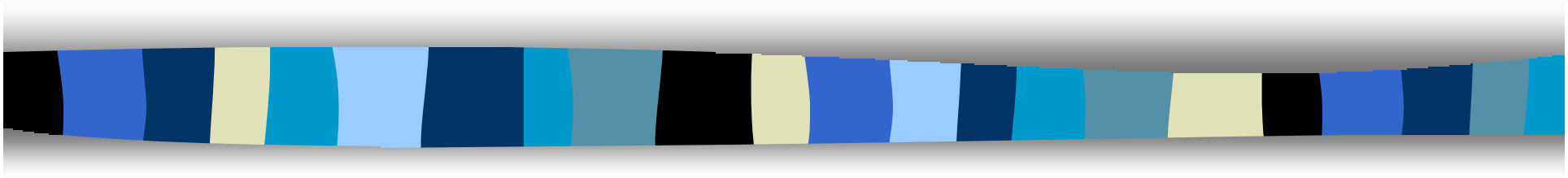
Software Performance Estimation by Static Analysis



Presented by : Robert Nilsson

References are available at the hw/sw co-design course web page

Outline

- Introduction and problem description
- Basic WCET estimation techniques
- Path Clustering and architecture classification
  - SFP / MFP - analysis
- Instruction cache modeling
  - direct memory mapped caches
- Pipelined architectures ?

Introduction

- Real-time systems need to guarantee timeliness
- Scheduling analysis assume known wcet of tasks
  - to predict worst case response times
  - also bcet (best-case execution times) relevant in some systems
- Useful during hw/sw codesign decision to map functions to resources

Introduction

- Worst case execution time analysis must be conservative
  - correctness (safety) vs. resource usage issue
- The execution time of a program depends on
  - program path
    - data dependent
  - computer architecture properties
    - e.g. caches, pipelines, etc

Problems

- Program path analysis is in general an undecidable problem
  - E.g. unbounded loops, recursive function calls
  - Even if such constructs are prohibited the number of program paths grows aggressively with
    - nested loops
    - branches in loops
- Difficult to model complex microarchitectures
  - E.g. caches, branch prediction, superscalar processors
  - Too expensive to neglect the impact of hardware properties
    - very pessimistic estimations → large resource waste

Program path analysis techniques

- Program code is classified into *basic blocks*
  - *a program segment which is only entered at the first statement and only left at the last statement*
- The basic blocks represents nodes in the program flow graph
- program paths are determined by traversing the flow graph from start block to end block
  - ***feasible paths*** can occur during execution
  - ***false paths*** can never occur during execution
- We need clever ways to classify paths !

Program path classification techniques

- loop annotation required for all loops
  - puts a on bound the number of program paths
- manually identification of classes of “false paths”
- implicit path enumeration
  - linear equations are used to specify constraints between blocks execution
    - e.g. $x1 \leq x2$ means block x1 is executed at most as often as x2

→ integer linear programming optimization problem

- e.g. simplex algorithm

Basic block timing analysis

- The actual execution times for a basic block is acquired trough :
  - **Instruction timing addition (ITA)**
    - execution time of instructions are added
      - number of processor cycles for each instruction
  - **Path segment simulation (PSS)**
    - a cycle true processor model is used to simulate the execution of the basic block (or a path segment)

Architectural properties and time analysis

- ***Data dependent instruction execution times***
  - hard for simulators to guarantee accurate timing
  - ITA could be used (using worst-case)
- ***Pipelined architectures***
  - PSS must assume worst-case behavior on block boundaries
  - ITA cannot anticipate pipeline hazards
- ***Superscalar architectures***
  - ITA is completely inappropriate if it does not model instruction scheduling

Architectural properties and time analysis, cnt.

- ***Program (instruction) caches***
  - PSS can be exact for program caches since the cache is simulated
    - worst case must be assumed at block boundaries
    - Better estimations if the blocks are big
  - ITA does not cover caches
    - need a method to model cache-hits and cache misses
- ***Data caches***
  - PSS is precise if the same data variables always are accessed in a block
  - Same properties as program caches

Execution time models

- Simplest is **sum-of-basic-blocks** model
  - each basic block has an associated constant execution time
- In many architectures this is not realistic
  - data dependent instruction execution times
  - overlapped basic block execution
  - ...

  → Different sequences of basic blocks (***path segments)*** gives different execution times
    - **sequences-of-basic-blocks** model needed

Program partitioning

- Analysis of **sequences-of-basic-blocks** model require exhaustive path analysis in the worst case

  →we're in trouble !

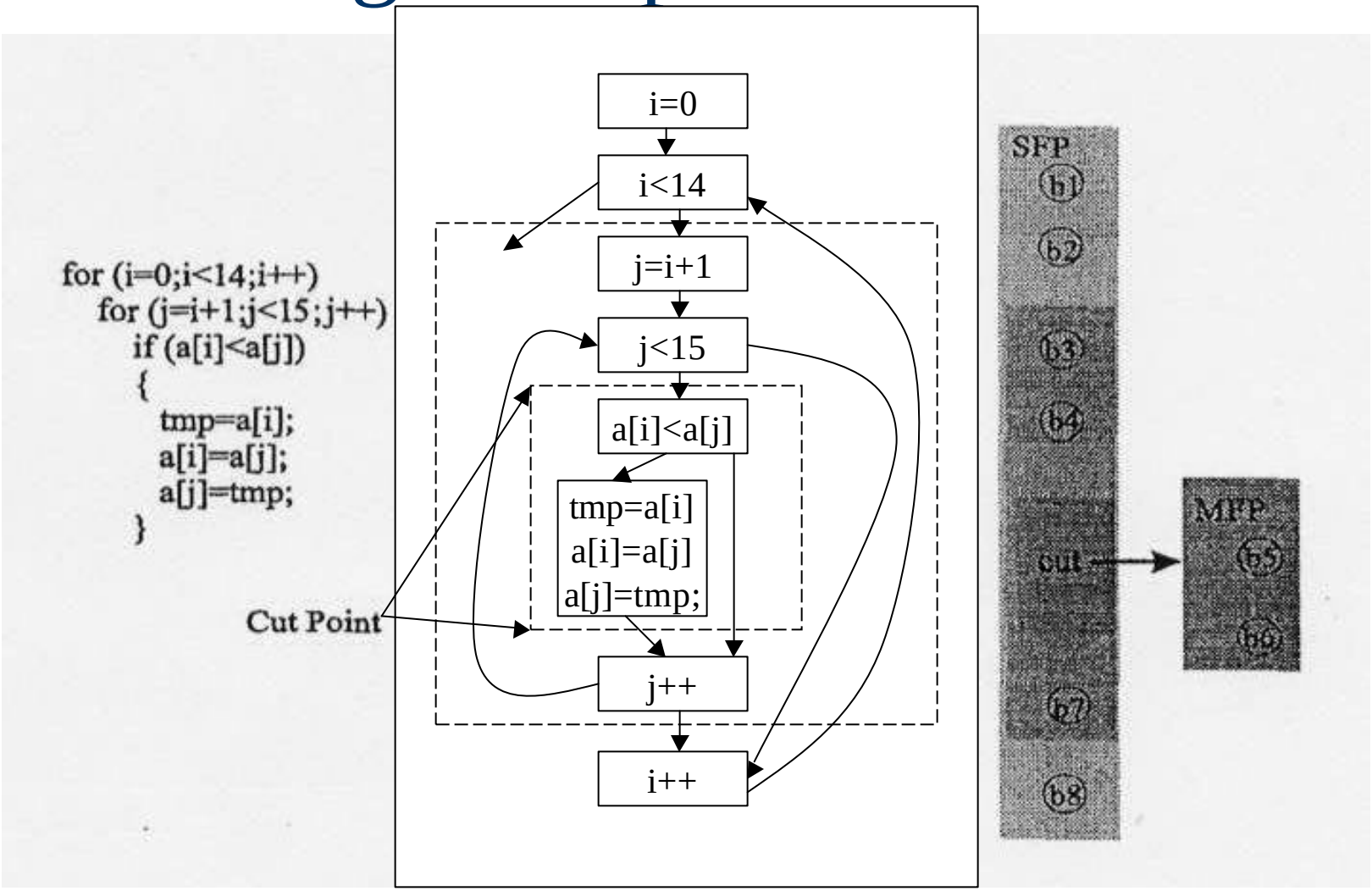
- Analysis is simplified if the program path is independent of input data
  - always get the same sequence of basic blocks
  - common in e.g. signaling processing applications
  - ***single feasible path (SFP)*** *property*

- However, most practical programs have at least some parts where the program path depends on input variables
  - ***multiple feasible paths (MFP)*** *property*

Symbolic hybrid timing analysis method

- R. Ernst, W.Ye (1997)
- Aim to partition the program in parts with the SFP and MFP properties
- "*Hierarchal flow graph clustering*" can be used for the partitioning
  - control constructs are hierarchical nodes
  - basic blocks are leaf nodes
    - Basic blocks are SFP by definition
    - A hierarchical node is SFP if
      - it only contain SFP nodes
      - its associated condition is independent of input data

Hierarchical flow graph clustering example

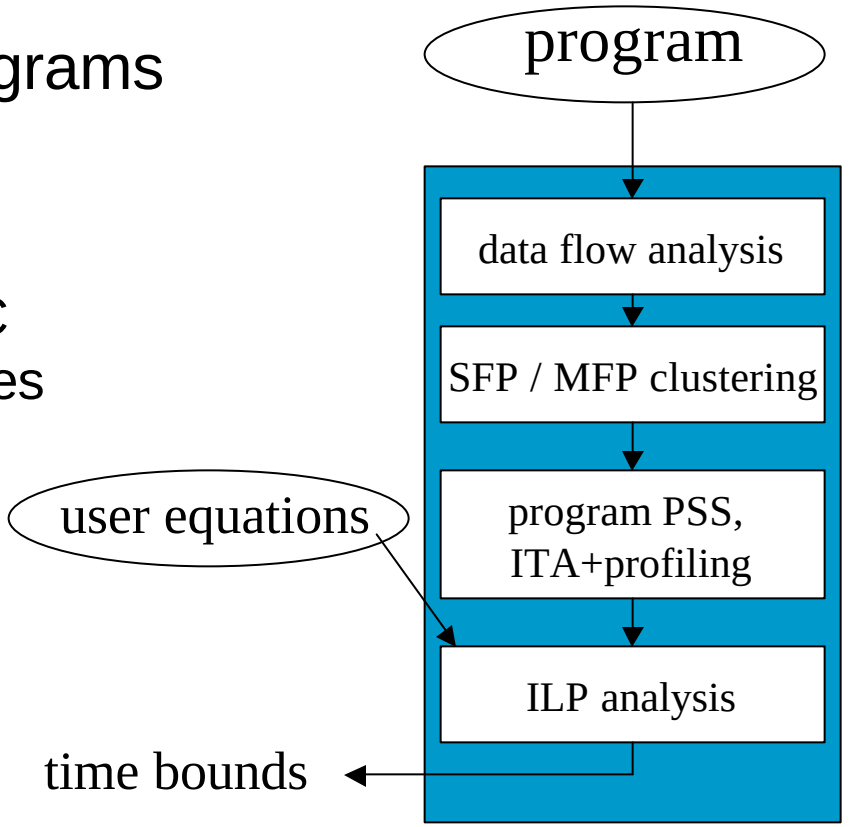


Global timing analysis

- For each SFP block, the execution time is determined.
  - using PSS or ITA as appropriate for the architecture
  - assuming worst-case behaviors at “cut-points”
- The resulting execution times for the SFP blocks are summarized
- For the remaining MFP parts, worst-case behavior is acquired
  - using for example a Integer Linear Programming approach
  - a pessimistic constant cost for each basic block in the reduced flow graphs

Experiments

- A tool was implemented for supporting the SYMTA approach
- Set of example programs was used
- Two architectures
  - superscalar SPARC with 4-stage pipelines
  - Intel 8051



Experimental results

Programs	Total nodes	Nodes in SFP		Nodes in MFP		Source lines
3D-image	94	85	90%	9	10%	164
diesel	65	65	100%	0	0%	160
fft	78	78	100%	0	0%	145
bsort	14	8	57%	6	43%	25
smooth	48	39	81%	9	19%	86
blue	80	53	66%	27	34%	127
check-data	18	0	0%	18	100%	44
whetstone	122	122	100%	0	0%	251
line	101	19	19%	82	81%	250
key3	100	100	100%	0	0%	151

Table 1: Experimental results for the Clustering

Experimental results

Programs	Measured bounds(cycles)		Analyzed bounds(cycles)		Analysis time* (sec)
	BCET**	WCET**	BCET	WCET	
SPARC					
3D-image	34908	37848	33874	38037	0.79
diesel	62944	62994	61445	63333	0.84
fft	1498817	1499176	1494650	1499290	135.69
bsort	4423	8938	4423	8938	0.34
smooth	3635651	4846511	3570227	4881135	304.90
blue	3564938	316865761	3345041	346541760	4325.23
check-data	80	431	65	435	0.23
whetstone	2928459	3369459	2880230	3378098	298.19
line	514	1619	381	2035	0.39
8051					
fft	26421460	26421460	26419338	26488288	0.23
bsort	9347	15045	7804	18167	0.12
smooth	9737378	9737516	9737469	9737522	0.23
key3	1218229	1223314	1164883	1265227	0.39
check-data	68	559	63	588	0.17

** The example programs have been analyzed on the SPARC 10 workstation.*
*** WCET: The worst case execution time; BCET: The best case execution time.*

Table 2: Experimental results of the example programs in SYMTA

Instruction Cache modeling

- Y-T.S Li,S. Malik. A Wolfe (1993)
- A method for getting tighter time estimations of programs running on architectures with instruction caches
- Cache memories are difficult to model and impose a lot of pessimism if neglected
  - direct mapped-instruction caches

Program path analysis as ILP

- Program path analysis can be transformed into a ILP problem using
  - program structural constraints
    - derived from program control flow graph (CFG)
  - mprogram functionality constraints
    - provided by the user
    - e.g. specifies loop bounds
  - *Without cache modeling a constant instruction execution time is assumed, hence;*

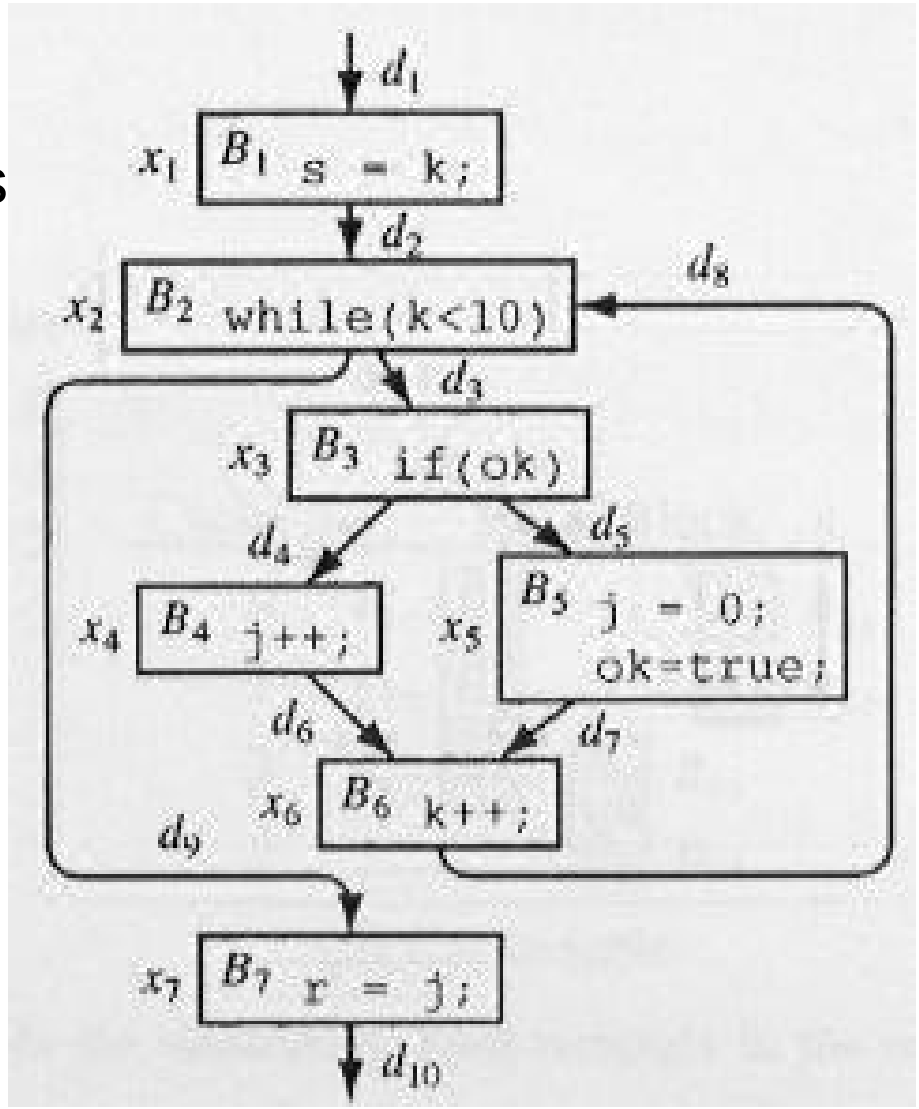
$$Execution\_time = \sum_{i=1}^{N} c_i x_i$$

c = execution time for basic block

x = execution count for basic block

Program path analysis example

- structural constraints
  - d1 = 1
  - x1 = d1 = d2
  - x2 = d2 + d8 = d3 +d9
  - x3 = d3 = d4+d5
  - x4 = d4 = d6
  - x5 = d5 = d7
  - x6 = d6 + d7 = d8
  - x7 =d9 = d10



Program path analysis example

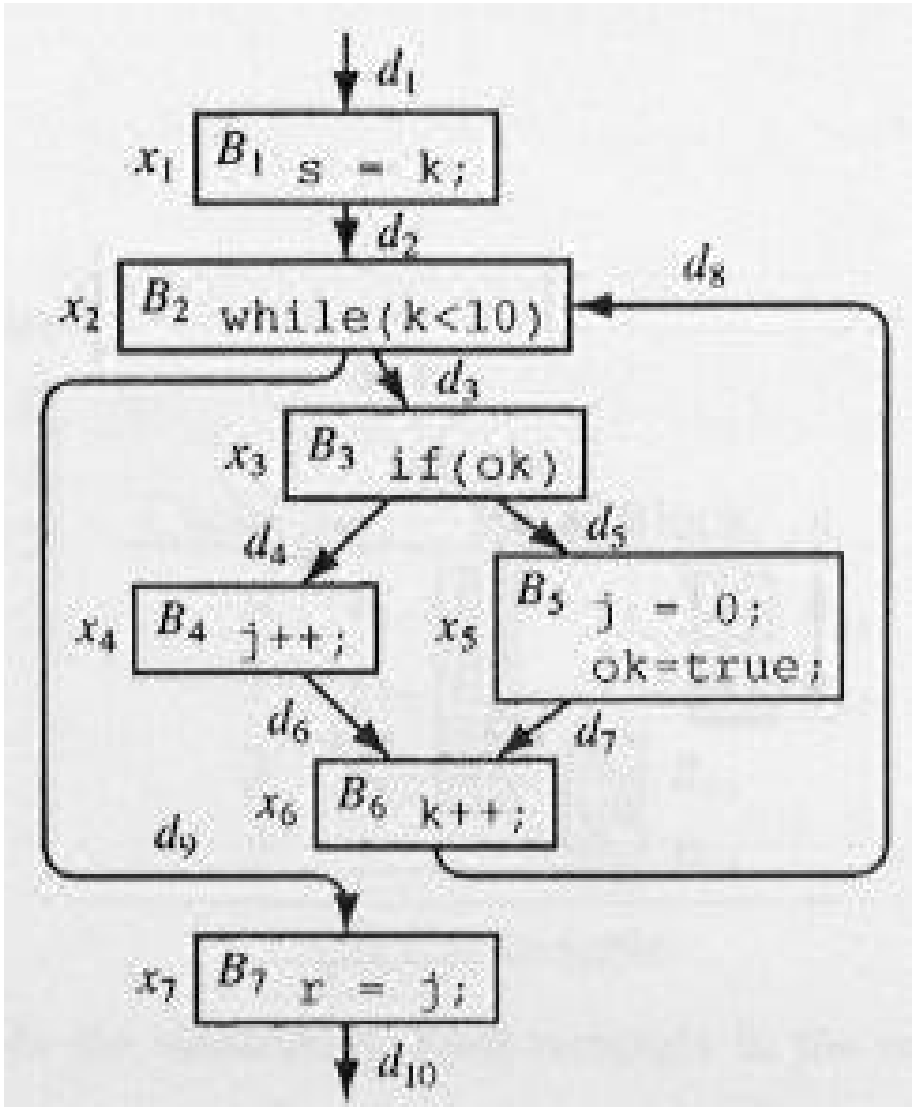
- functional constraints

  0 x1 ≤ x3 ≤ 10 x1

  x5 ≤ 1 x1

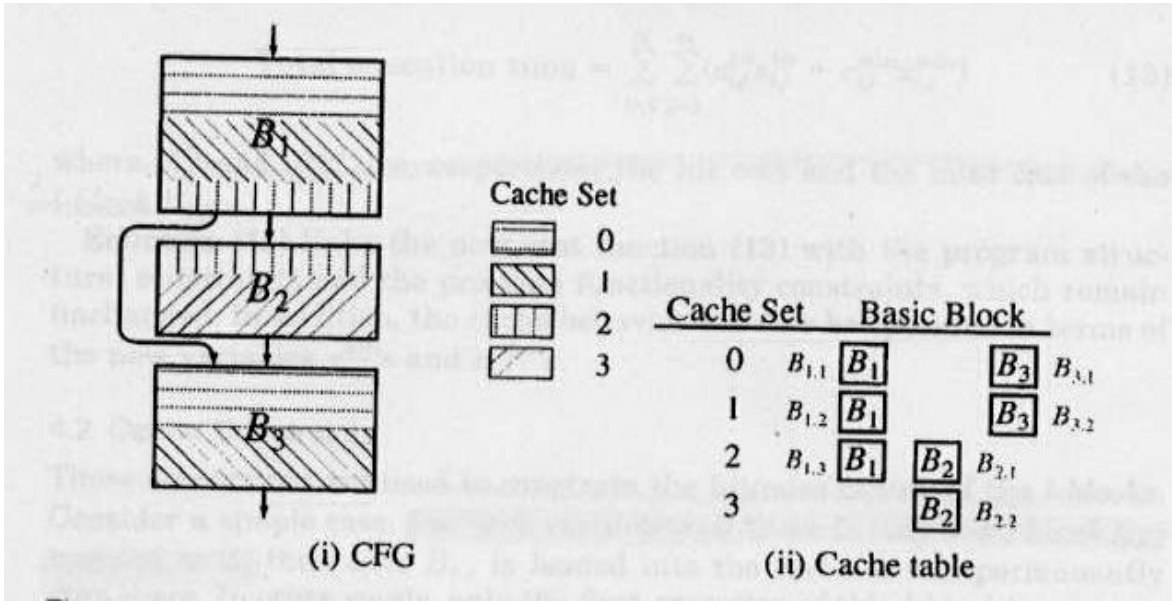
  ...

- All these constraints are passed to the ILP-solver



Direct mapped Instruction caches

- the code in basic blocks are divided into a number of line-blocks
- the line blocks are assigned to cache lines
  - cache sets (cache lines) represent physical cache memory



Adding cache analysis to the ILP model

- Now execution times of basic blocks differ if the line-blocks are in the cache or not

$$Execution\_time = \sum_{i=1}^{N} \sum_{j=1}^{n_i} \left(c_{i,j}^{hit} x_{i,j}^{hit} + c_{i,j}^{miss} x_{i,j}^{miss} \right)$$

- the execution count of a basic block becomes

$$x_i = x_{i,j}^{hit} + x_{i,j}^{miss}$$

$j = 1,2 \ldots n_i$

i = all basic blocks

j = all line blocks in block i

x^{hit} = number of cache hits

x^{miss} = number of cache misses

c^{hit} = execution time for cache hit

c^{miss} = execution time of cache miss

Cache constraints

- There are three possible types of cache assignments that can occur
  - Only one line-block assigned to a cache line
    - when a miss occurs, the line-block will be loaded and no more cache misses will occur

$$x_{k.l}^{miss} \leq 1$$

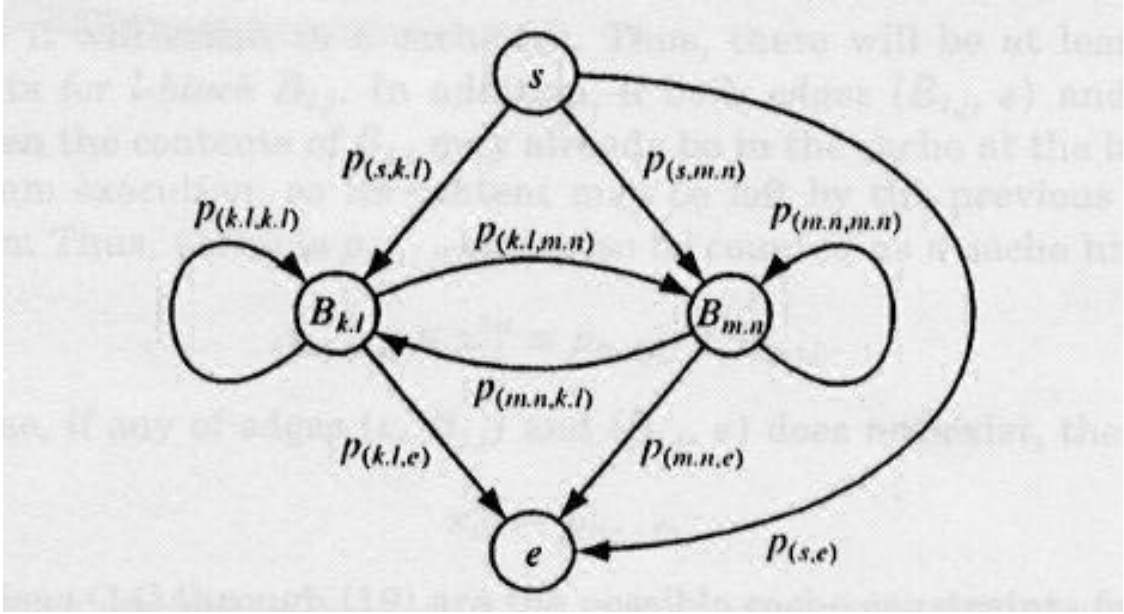
  - Two or more *nonconflicting* line-blocks are assigned to the same cache line
    - when a miss occurs in either block, the line-blocks will be loaded and no more cache misses will occur

$$x_{1.3}^{miss} + x_{2.1}^{miss} \leq 1$$

  - a cache line contains two or more conflicting line-blocks

Cache conflict graphs

- s and e nodes represents the start and the end of the program respectively
- B – nodes represent conflicting line-blocks
- Edges represent possible program flow between blocks
  - acquired from program cfg
- p(node1, node2) is a counter associated with each edge

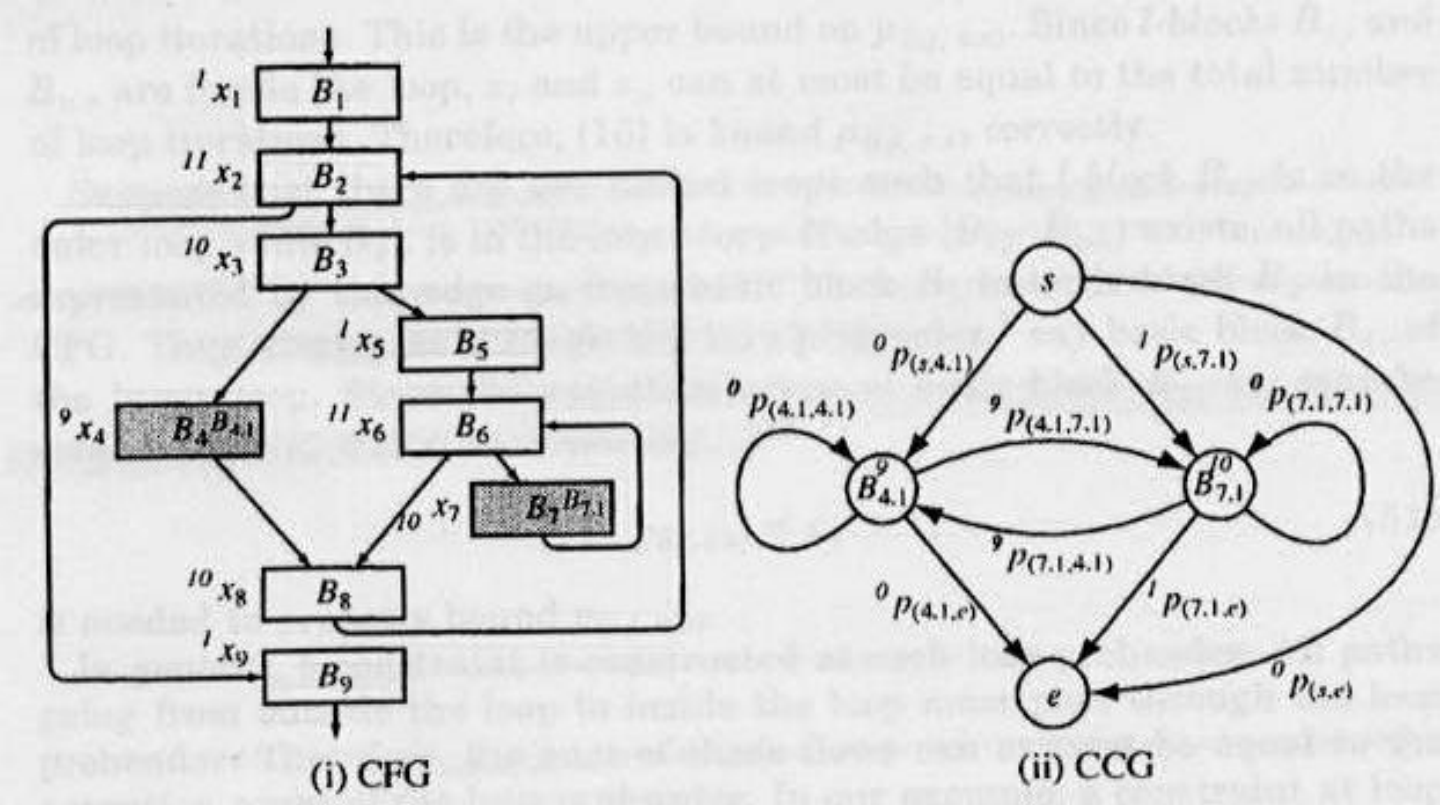


Constraints on cache conflict graphs

- The counters (p) are bound to the structural and functional constraints trough the x variables
  - the execution count of a line-block must be equal to the execution count of the basic block
  - the control flow to a line-block node must be equal to the flow from the line-block node

$$x_i = \sum_{u.v} p(u.v, i.j) = \sum_{u.v} p(i.j, u.v)$$

Loops and cache constraints



$$p(s,7.1) + p(4.1) \leq x_5$$

Experiments

- Intel QT960
  - 32 x 16 bytes direct-mapped instruction cache
- Execution times are assigned statically to line-blocks hits and misses
- tool, called “cinderella” was implemented
  - generate CFG
  - generates CCG
  - output structural constraints
  - ask user for loop bounds and other optimization constraints
  - used in conjunction with public domain ILP-solver
- Evaluation programs chosen so that worst-case input data was available for measurement and comparison

Experiments results

Table II. Estimated WCETs of Benchmark Programs. Estimated WCETs and Measured WCETs In Units of Clock Cycles

Program	Measured WCET	Estimated WCET	Ratio
check data	4.30×10^2	4.91×10^2	1.14
circle	1.45×10^4	1.54×10^4	1.06
des	2.44×10^5	3.70×10^5	1.52
dhry	5.76×10^5	7.57×10^5	1.31
djpeg	3.56×10^7	7.04×10^7	1.98
fdct	9.05×10^3	9.11×10^3	1.01
fft	2.20×10^6	2.63×10^6	1.20
line	4.84×10^3	6.09×10^3	1.26
matcnt	2.20×10^6	5.46×10^6	2.48
matcnt2	1.86×10^6	2.11×10^6	1.13
piksrt	1.71×10^3	1.74×10^3	1.02
sort	9.99×10^6	27.8×10^6	2.78
sort2	6.75×10^6	7.09×10^6	1.05
stats	1.16×10^6	2.21×10^6	1.91
stats2	1.06×10^6	1.24×10^6	1.17
whetstone	6.94×10^6	10.5×10^6	1.51

WCET analysis beyond direct mapped instruction caches

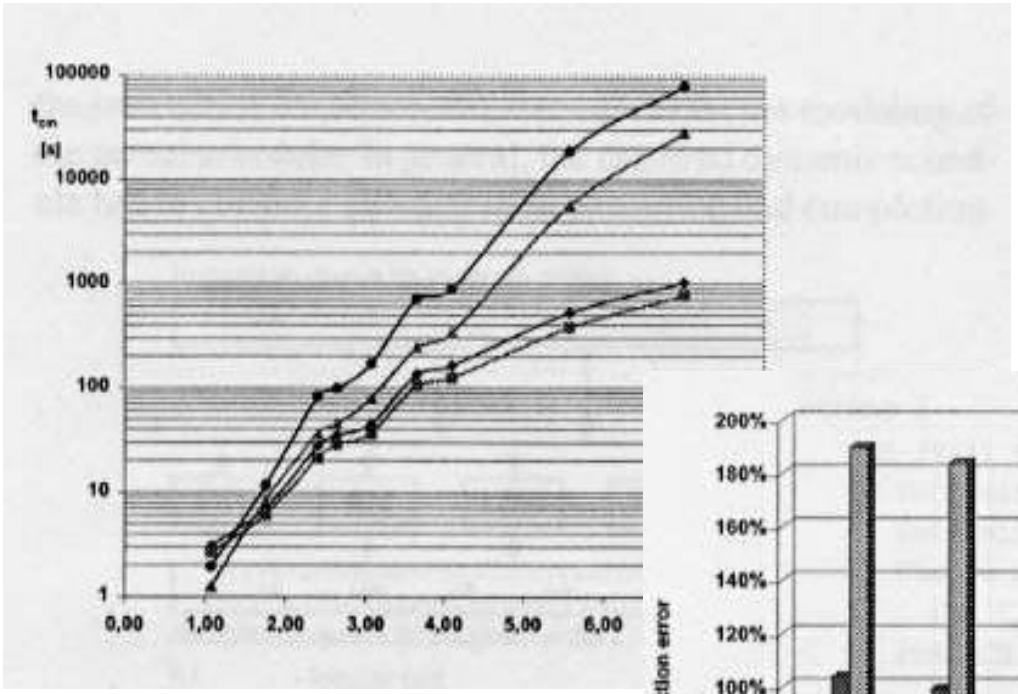
- There are several variables which influence the complexity of cache analysis
  - number of competing line-blocks (m)
  - cache associatity level (n)
  - cache replacement method

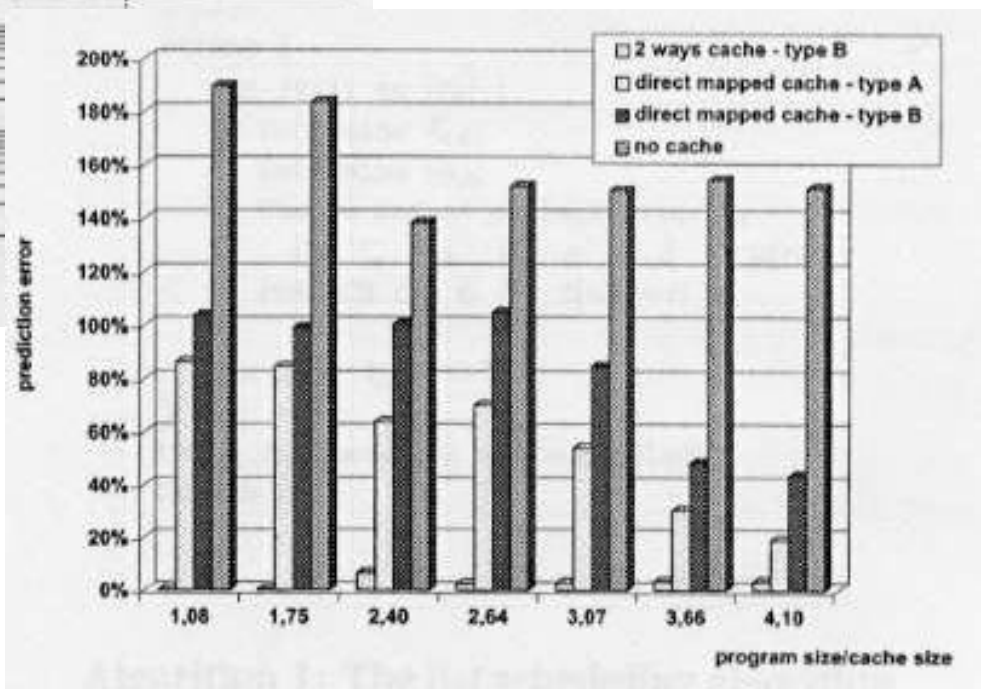
- For LRU (least recently used), the complexity grows as

$$\sum_{i=0}^{n} \frac{m!}{(m-i)!}$$

- By using a more detailed level of cache modeling better estimations can be acquired, but the problem become intractable if programs are large

Some measurements





Analysis of multi-issue Pipelines

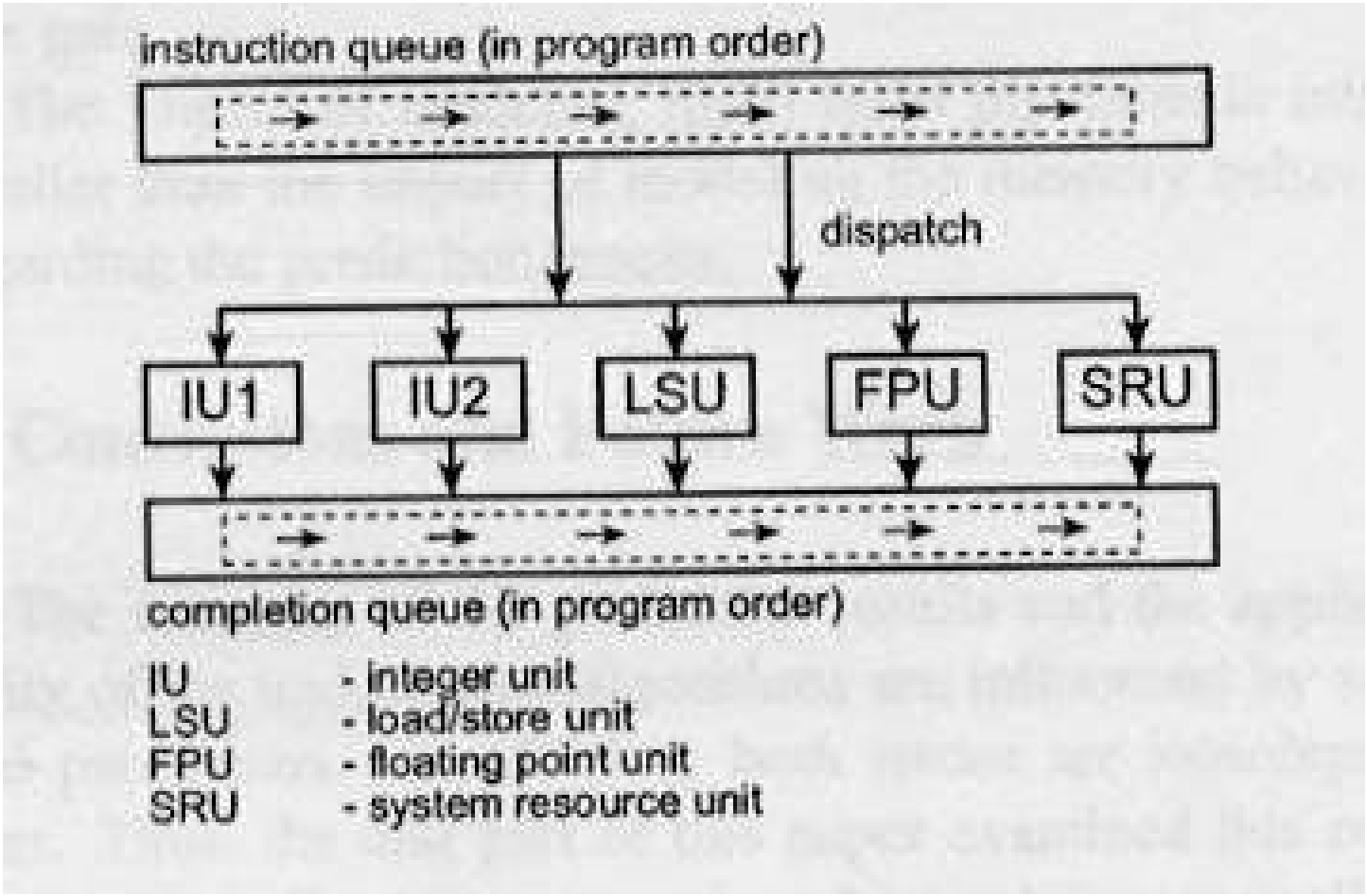
- Previously we have assumed that no parallelism occur in the processor
- If we can model the gain from instruction level parallelism we can get a tighter WCET bound
- When introducing pipelines in our model, the execution time depends on instruction scheduling
  - the code provides data-dependencies and order between instructions
  - the “assignment” of instruction types to resource types needs to be explicitly given

Modeling instruction level schedules

- A processor is said to have a set of resource types
  For each resource types there are a set of instances of this resource type
  - e.g 2 floating point units

- The program consist of a queue of instructions (in basic block or line-block)
  - The priority of the instruction is set according to the order of the program and data-dependecies
  - if two instruction have no data-depencencies they can have the same priority

Pipelined architecture

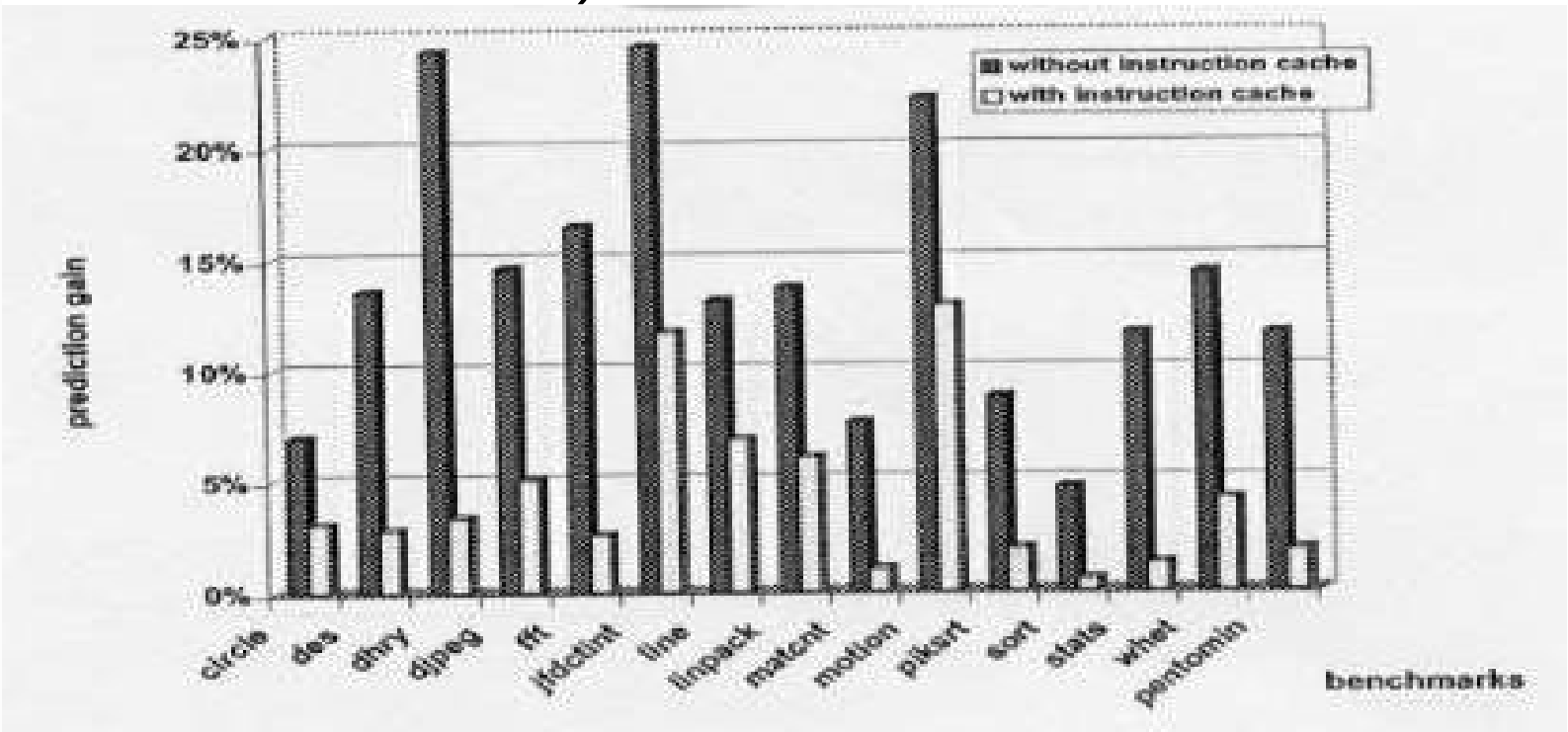


Simplified scheduling algorithm

- For each time unit
  - for each resource type
    - Determine set of instructions which can be executed at this time-unit without violating data dependencies
    - Determine which resource instances that can execute instructions of this type at this time
    - assign highest priority instructions to free resources
  - Continue until all instructions are scheduled

Prediction gain

- The schedule gives the number of time units that are required for executing the block (or the line-block if caches are used)



Summary

- Performance estimations neglecting processor architectures are to pessimistic
- Avoid explicit enumeration of execution paths
  - Program partitioning
  - implicit path enumeration → ILP problem
- Cache modeling improve tightness considerable
  - but for advanced cache models the complexity of analysis explode
- Pipeline schedule modeling
  - Improves bound slightly
  - Knowledge about instruction scheduling policy required