H/S Codesign: Performance Analysis of Software

The University of Texas at Austin

Performance Analysis

Princeton: Sharad Malik, Wayne Wolf, Andrew Wolfe
(Cinderella Project)

- **Introduction**
- **Microarchitectural Modeling**
  - **Cache**
  - **CPU Pipelining**
- **Experimental Results**

Worst Case Analysis

- **Worst case execution time (WCET) of a process running on a given processor**
  - **In real time systems: designer must verify if the WCET satisfies the timing deadlines**
    - **Many real- time OSs rely on this information for process scheduling**
  - **In embedded system design: the WCET is often required to decide on Hw/Sw partitioning**

Worst Case Analysis

- **Determining the actual WCET of a program**

  ⇨ ***Impractical*: requires simulating all possible combinations of *input data values* and *initial system states***
- **Alternative: obtaining an estimate of the actual WCET by a static analysis of the program**
  - **Estimated WCET: must be tight and conservative (informative upper bound)**

Problem Formulation

➡ **Determining the the estimated WCET of a given program on a given hardware system, assuming *uninterrupted execution***

- **hardware system: includes the microprocessors and the memory systems**

- **Problem has two components**
  - **Program path analysis**
  - **Microarchitectural modeling**

Estimating WCET

- **The problem of finding a program's estimated WCET is in general undecidable**
- **Conditions for this problem to be decidable**
  - **absence of recursive function calls**
  - **bounded loops**

Program Path Analysis

- In order to tighten the estimated WCET, it is necessary to remove logically unfeasible program paths
  - path information provided by programmer
  - scope of path information provided by designer may have direct impact on the tightness of the estimated WCET
- Statically feasible paths and programmer annotations
  - loop bounds;
  - interactions among different program statements (e.g., mutual exclusion of two statements)

Estimating WCET

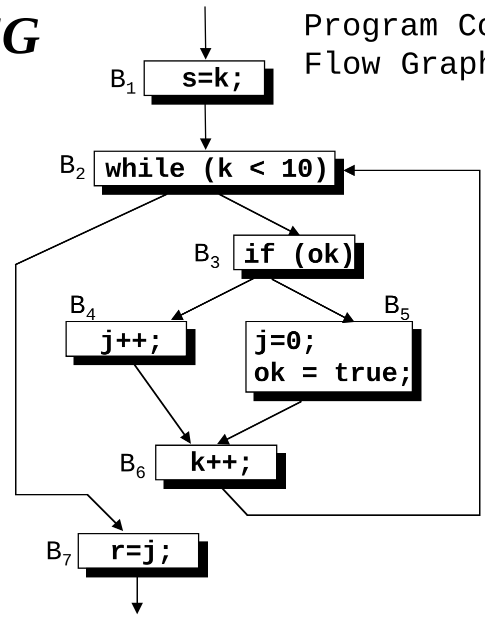
- Approach: the problem of solving the estimated WCET is converted into a set of Integer linear programming (ILP) problems
- Discussion assumes (for now) a simple (pessimistic) microarchitectural model
  - execution time of an instruction is constant
    - » every instruction fetch results in a cache miss

Program CFG

Program Control Flow Graph (CFG)

A basic block is a maximum contiguous sequence of instructions, with a single entry point at its start and a single exit point at its end

```
/*k >= 0*/
s=k;
while (k < 10){
  if (ok)
    j++;
  else {
    j=0;
    ok = true;
  }
 k++;
}
r=j;
```



Estimating WCET: ILP Formulation

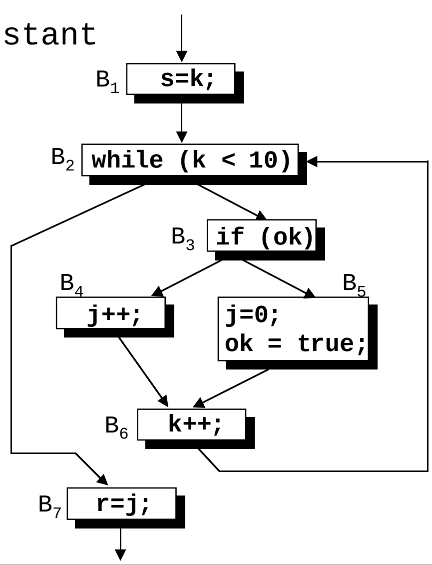
Assumption: each instruction takes constant time to execute

$$\text{Total execution time} = \sum_{i}^{N} c_i x_i$$

N = number of BBs

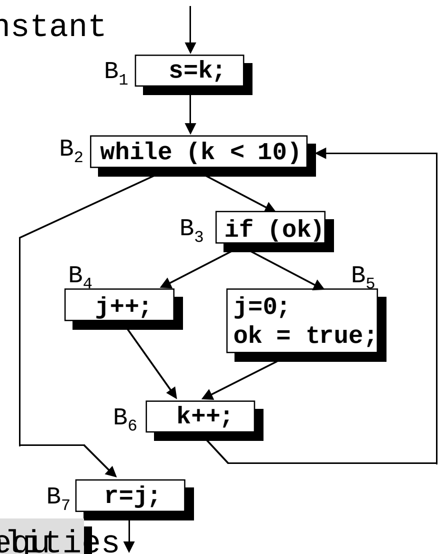
c_i = execution time of basic block B_i

x_i = execution count of basic block B_i



Estimating WCET: ILP Formulation

Assumption: each instruction takes constant time to execute



$$\text{Total execution time} = \sum_{i}^{N} c_i x_i$$

N = number of BBs
c_i = execution time of basic block B_i
x_i = execution count of basic block B_i

Constrained by
- ☛ Program structure
- ☛ Values of program variables

constraints represented as linear equalities

Execution Count of BBs

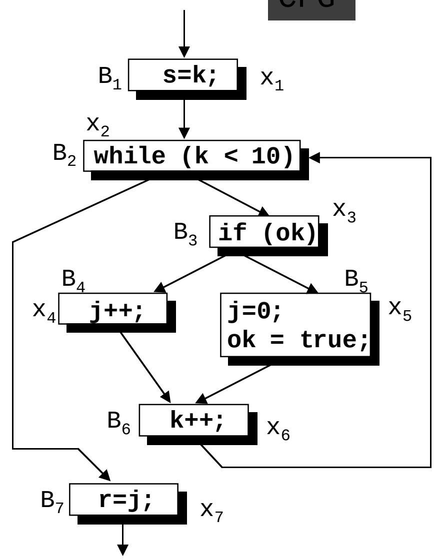
Two types of constraints on x_i:

- **Structural constraints**
  - **derived from the program's control flow graph (CFG)**
- **Program functionality constraints**
  - **provided by the user to specify loop bounds, and other path information**

Structural Constraints

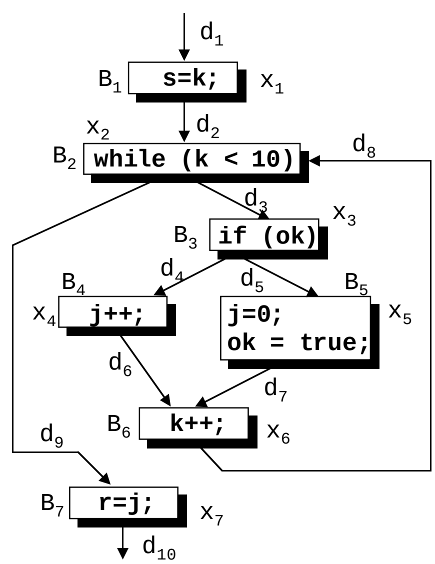
CFG

- Each node B_i defines an execution count variable x_i



Structural Constraints

- Each node B_i defines an execution count variable x_i
- Each edge defines a variable that counts the number of times the control flow passes through that edge

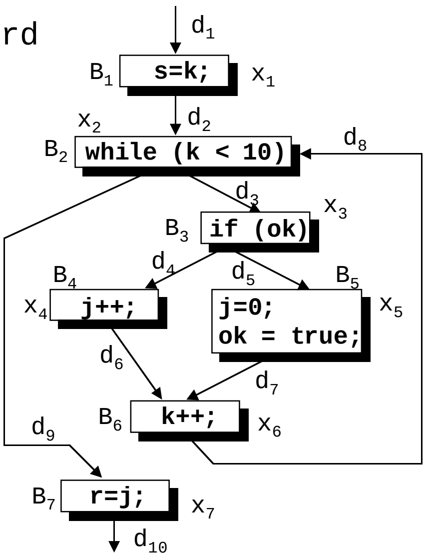


Structural Constraints

Analysis of CFG similar to standard network flow problem.

Execution count of node B_i is equal to:

- Number of times the control enters the node (inflow)
- Number of times the control exits the node (outflow)

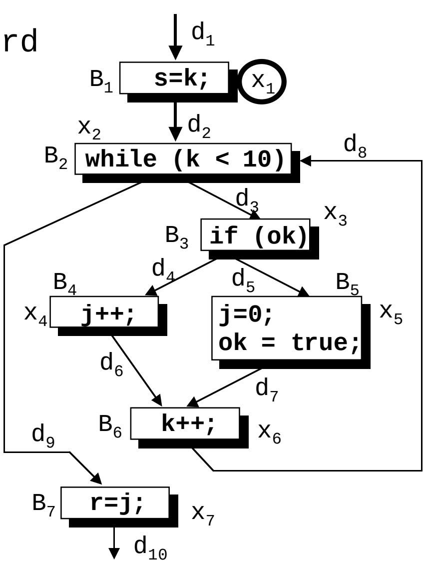


Structural Constraints

Analysis of CFG similar to standard network flow problem.

Execution count of node B_i is equal to:

- Number of times the control enters the node (inflow)
- Number of times the control exits the node (outflow)



$d_1 = 1$ - specifies that code fragment is to be executed once

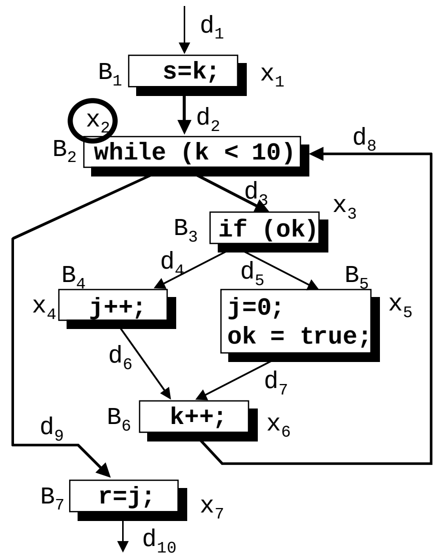
$\boxed{x_1 = d_1 = d_2}$

Structural Constraints

$d_1 = 1$

$x_1 = d_1 = d_2$

$\boxed{x_2 = d_2 + d_8 = d_3 + d_9}$



Structural Constraints

$d_1 = 1$

$x_1 = d_1 = d_2$

$x_2 = d_2 + d_8 = d_3 + d_9$

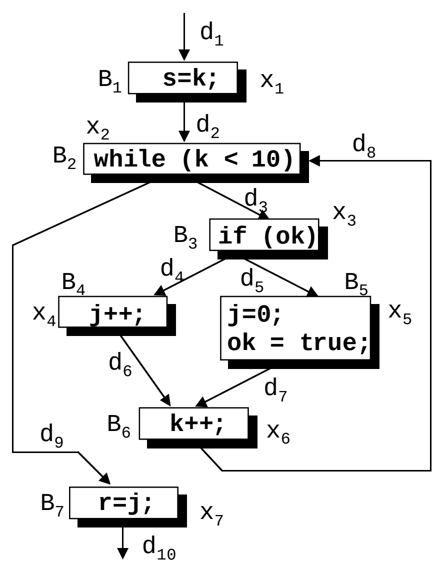
$x_3 = d_3 = d_4 + d_5$

$x_4 = d_4 = d_6$

$x_5 = d_5 = d_7$

$x_6 = d_6 + d_7 = d_8$

$x_7 = d_9 = d_{10}$



Constraints Execution Count of BBs

- **Structural constraints**
  - **derived from the program's control flow graph (CFG)**
- **Program functionality constraints**
  - **provided by the user to specify loop bounds, and other path information**

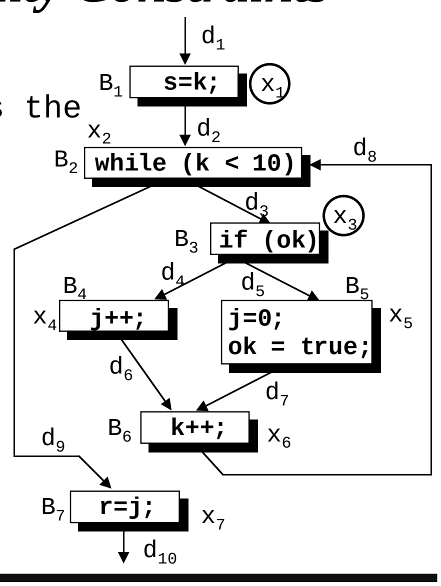
Program Functionality Constraints

Loop Bounds

Since k is positive before it enters the loop, the loop body will be executed between 0 and 10 times each time the loop is entered

$\boxed{0x_1 \leq x_3 \leq 10x_1}$

```
/*k >= 0*/
s=k;
while (k < 10){
  if (ok)
    j++;
  else {
    j=0;
    ok = true;
  }
 k++;
}
r=j;
```



Margarida Jacome - UT Austin 99 - 19

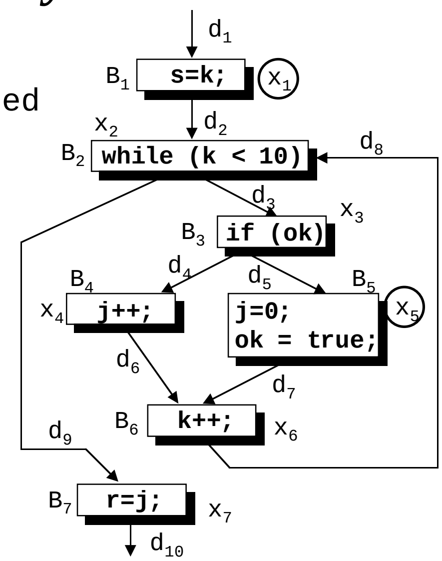
Program Functionality Constraints

- Other path information

Ex.1: If else statement (B5) can be executed at most once inside the loop

$\boxed{x_5 \le 1}\ x_3$

```
/*k >= 0*/
s=k;
while (k < 10){
  if (ok)
    j++;
  else {
    j=0;
    ok = true;
  }
 k++;
}
r=j;
```



Program Functionality Constraints

- Other path information

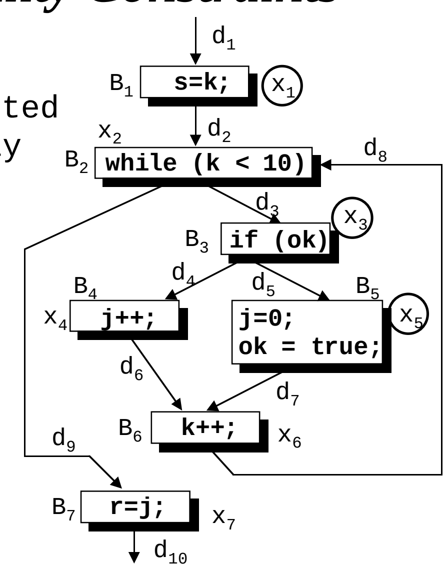
Ex.2: If the else statement (B5) is executed the loop will be executed exactly 5 times

$\boxed{(x_5 = 0) \mid (x_5 \ge 1\ \&\ x_3 = 5\, x_1)}$

Non linear constraint

disjunction of linear constraint sets

(at least one constraint must be satisfied)



Solving the Constraints

- **Because of the AND and OR operators, the program functionality constraints may, in general, be a disjunction of conjunctive constraint sets**
- **To solve the estimated WCET, each set of the functionality constraints is combined with the set of structural constraints**
  - **the combined set is passed to the ILP solver with (1) to be maximized**

$$\text{Total execution time} = \sum_{i}^{N} c_i x_i \qquad (1)$$

ILP solver returns max value of expression and the BB counts

Solving the Constraints

- **Procedure is repeated for every functionality constraint set**
- **The maximum over all these running times is the estimated WCET**

$$\text{Total execution time} = \sum_{i}^{N} c_i x_i$$

Time Complexity

- **Total time required to solve the estimated WCET depends on the number of functionality constraint sets and the time to solve each constraint set...**
- **Number of functionality constraint sets doubles for each functionality constraint with a disjunction operator**

Microarchitectural Modeling

Objective: model the CPU pipeline and the cache memory systems and determine the execution time (c_i's) of the BBs

Assumption: a cache miss stalls the entire CPU pipeline

Allows to divide the execution of a BB into two parts:

- ☛ Time for cache penalty (program cache)
- ☛ Memory-invariant ideal execution time in the CPU pipeline

Directed-Mapped Instruction Cache Analysis

- **Previous cost function (total execution time) is modified**
- **Linear constraints representing the cache memory behavior (cache constraints) are added**

Modified Cost Function

- **Without cache, the factors that affects total execution time are the execution counts of BBs and their execution times**
- **With cache, the execution time of an instruction may vary, depending on whether it results in a cache hit or a cache miss**
  - **for each instruction, subdivide original execution counts into counts of the *numbers of cache hits and cache hits***
  - **consider the cache hit and cache miss execution time of each instruction**

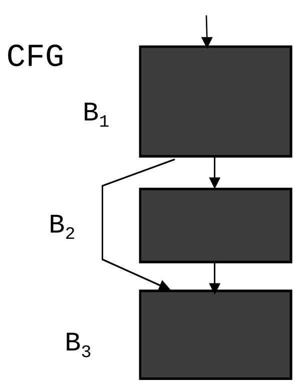
Modified Cost Function

- Instructions inside a BB
  - have the same execution count
  - may have different cache miss and cache hit counts
    - » they may map to different cache lines, each of which may have different cache activity

Different grouping of instructions is utilized for cache analysis ⇒ a new type of atomic structure

Line-Block (l-block)

- Defines a contiguous sequence of instructions within the same BB that are *mapped to the same line in the instruction cache*
  - the instructions within an *l-block* will always be *executed the same number of times* and have the same cache hit and cache miss counts

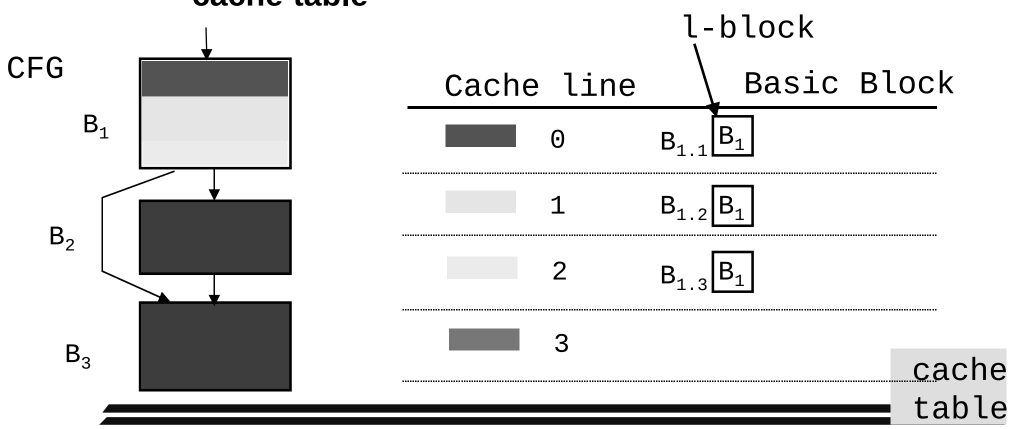


Example:

- ☛ CFG with 3 BBs
- ☛ Instruction cache with 4 lines

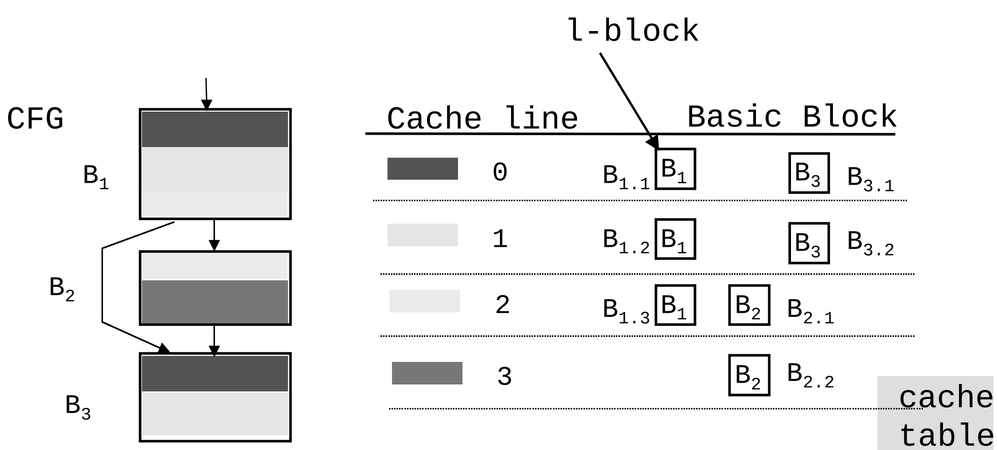
Identifying l-blocks

- For each basic block, find all the cache lines that instructions within it map to
  - Add an entry on these cache lines in the cache table



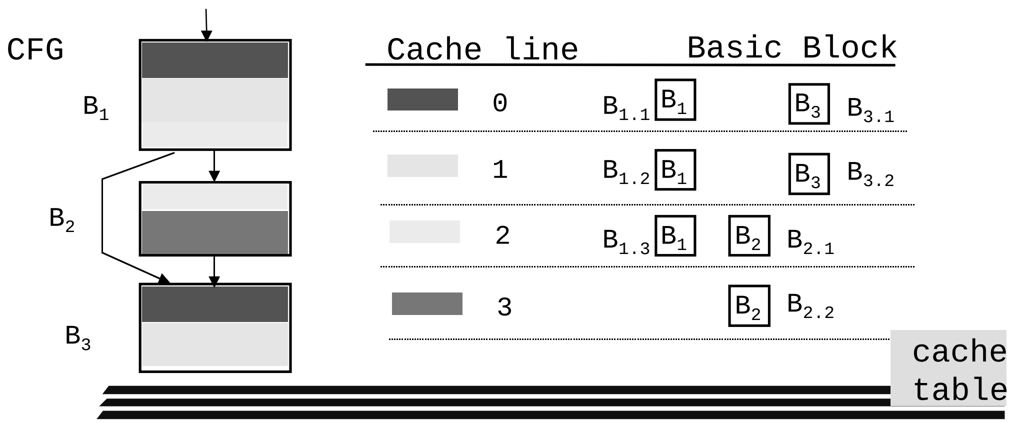
Line-Block (l-block)

- Basic block B_1 is partitioned into three *l-blocks*: $B_{1.1}$, $B_{1.2}$ and $B_{1.3}$



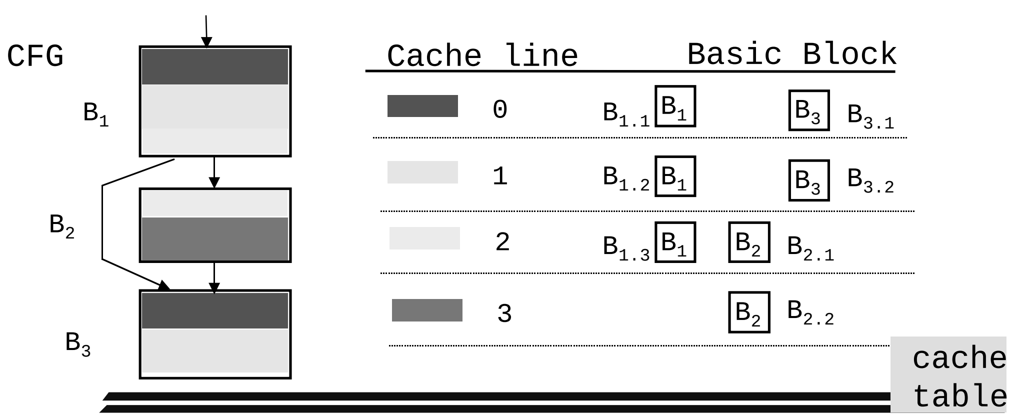
Line-Blocks (l-blocks)

- Two basic blocks are said to conflict with each other if the execution of one l-block will displace the other l-block in the instructions cache
  - for this to happen, they must map to the same cache line
  - Example: L-block $B_{1.1}$ conflicts with $B_{3.1}$



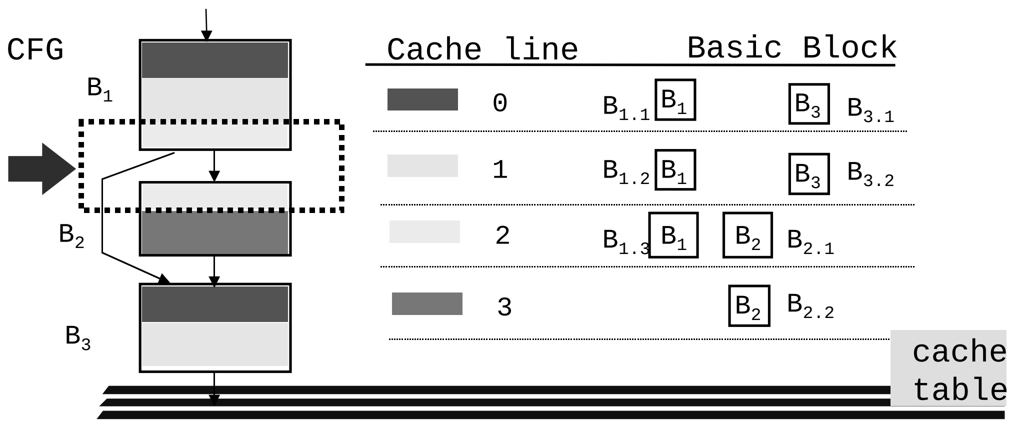
Line-Blocks (l-blocsk)

- L-Block $B_{2.2}$ does not conflict with any other blocks (once it is in cache, it will never be displaced)



Line-Block (l-block)

- L-blocks $B_{1.3}$ and $B_{2.1}$ occupy only a partial cache line
  - A cache miss during the execution of either l-block will cause the system to load the instructions of both l-blocks into cache. Later executions will be cache hits.



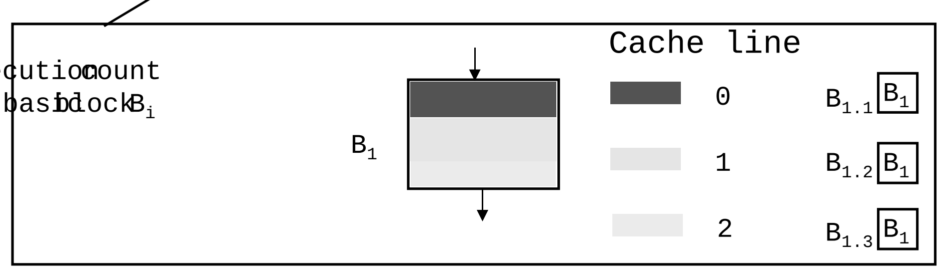
Execution Count of L-Blocks

- Execution count of l-block $B_{i.j}$ is denoted as $x_{i.j}$
- Cache hit count is denoted $x_{i.j}^{hit}$
- Cache miss count is denoted $x_{i.j}^{miss}$

(number of l-blocks of basic block B_i)

$$x_i = x_{i.j} = x_{i.j}^{hit} + x_{i.j}^{miss}, \quad 1 \le j \le n_i$$

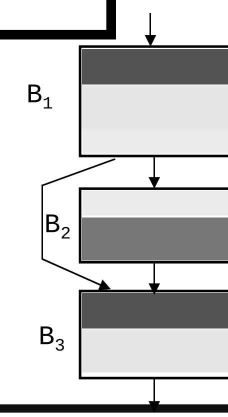
Execution count of basic block B_i



Total Execution Time

$$\text{Total execution time} = \sum_{i}^{N} \sum_{j}^{n_i} (c_{i.j}^{hit} x_{i.j}^{hit} + c_{i.j}^{miss} x_{i.j}^{miss})$$

- N = number of BBs
- n_i = number of l-blocks of BB B_i
- x_{ij}^{hit} = cache hit count of l-block $B_{i.j}$
- x_{ij}^{miss} = cache miss count of l-block $B_{i.j}$
- c_{ij}^{hit} = hit cost of l-block $B_{i.j}$
- c_{ij}^{miss} = miss cost of l-block $B_{i.j}$



(discussed later - CPU pipeline)

Constraints

$$\text{Total execution time} = \sum_{i}^{N} \sum_{j}^{n_i} (c_{i.j}^{hit} x_{i.j}^{hit} + c_{i.j}^{miss} x_{i.j}^{miss})$$

- **No changes to previous structural and program functionality constraints**

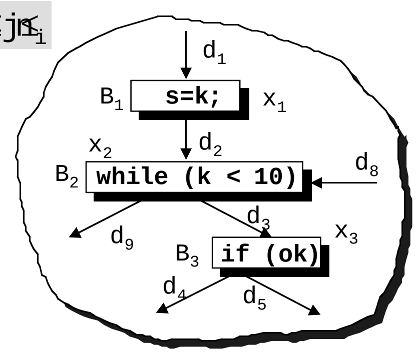
$$x_i = x_{i.j}^{hit} + x_{i.j}^{miss}, \quad 1 \leq j \leq n_i$$

Structural

$d_1 = 1$
$x_1 = d_1 = d_2$
$x_2 = d_2 + d_8 = d_3 + d_9$
$x_3 = d_3 = d_4 + d_5$

Functional

$(x_5 = 0) \mid (x_5 \geq 1 \;\&\; x_3 = 5 x_1)$



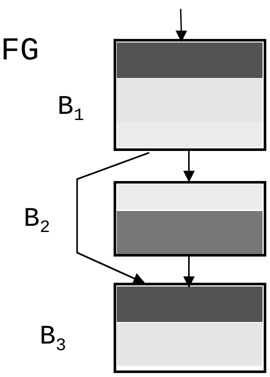
Margarida Jacome - UT Austin 99 - 19

Cache Constraints

Total execution time = $\sum_{i}^{N} \sum_{j}^{n_i} (c_{i.j}^{hit} x_{i.j}^{hit} + c_{i.j}^{miss} x_{i.j}^{miss})$

- **Additional "cache constraints" required...**

CFG



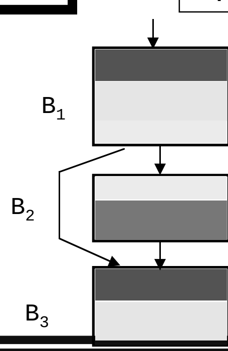
Cache line	Basic Block		
0	$B_{1.1}$ $\boxed{B_1}$		$\boxed{B_3}$ $B_{3.1}$
1	$B_{1.2}$ $\boxed{B_1}$		$\boxed{B_3}$ $B_{3.2}$
2	$B_{1.3}$ $\boxed{B_1}$	$\boxed{B_2}$ $B_{2.1}$	
3		$\boxed{B_2}$ $B_{2.2}$	

Cache Constraints

For each cache line

- If there is only one block $B_{k.l}$ that maps to it
  the first execution of the block may cause a cache miss, and all subsequent executions will result in cache hits

$\boxed{x_{k.l}^{miss} \le 1}$ and $\boxed{x_i = x_{i.j}^{hit} + x_{i.j}^{miss}}$



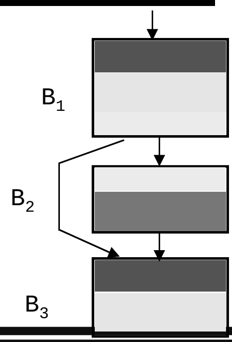
Cache line	Basic Block		
0	$B_{1.1}$ $\boxed{B_1}$		$\boxed{B_3}$ $B_{3.1}$
1	$B_{1.2}$ $\boxed{B_1}$		$\boxed{B_3}$ $B_{3.2}$
2	$B_{1.3}$ $\boxed{B_1}$	$\boxed{B_2}$ $B_{2.1}$	
3		$\boxed{B_2}$ $B_{2.2}$	

Cache Constraints

For each cache line

- If two or more non-conflicting l-blocks (e.g. $B_{1.3}$ and $B_{2.1}$) map to the line, execution of one of them will load the entire cache line content.

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

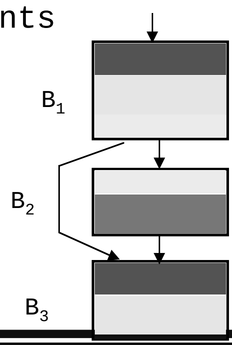


Cache line	Basic Block			
0	$B_{1.1}$	$\boxed{B_1}$	$\boxed{B_3}$	$B_{3.1}$
1	$B_{1.2}$	$\boxed{B_1}$	$\boxed{B_3}$	$B_{3.2}$
2	$B_{1.3}$	$\boxed{B_1}$	$\boxed{B_2}$	$B_{2.1}$
3			$\boxed{B_2}$	$B_{2.2}$

Cache Constraints

For each cache line

- If two or more conflicting l-blocks (e.g. $B_{1.3}$ and $B_{2.1}$) map to it, their execution will be affected by the sequence in which these l-blocks are executed
  - The execution of other l-blocks does not affect any counts



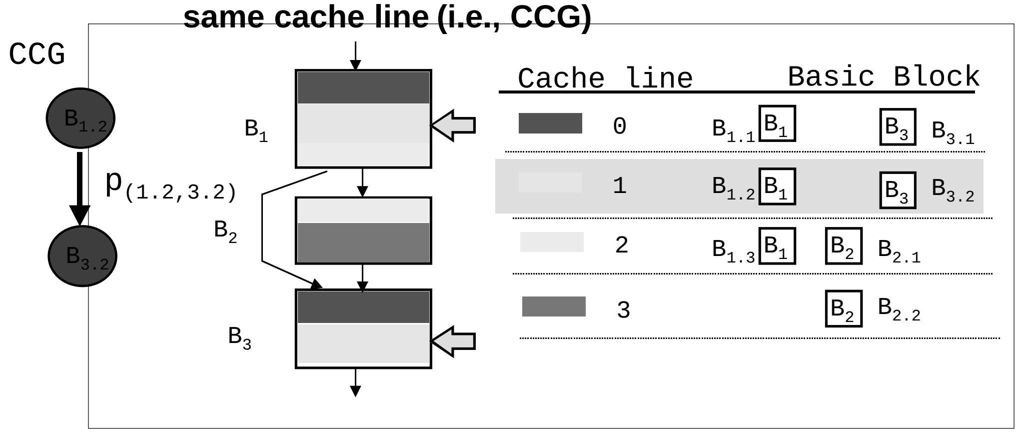
Cache line	Basic Block			
0	$B_{1.1}$	$\boxed{B_1}$	$\boxed{B_3}$	$B_{3.1}$
1	$B_{1.2}$	$\boxed{B_1}$	$\boxed{B_3}$	$B_{3.2}$
2	$B_{1.3}$	$\boxed{B_1}$	$\boxed{B_2}$	$B_{2.1}$
3			$\boxed{B_2}$	$B_{2.2}$

Cache Conflict Graph

- A cache conflict graph is constructed for every cache line that contains two or more conflicting l-blocks
- Nodes:
  - a start node *s* and an end node *e*
    - » representing the start and the end of the program
  - a node $B_{k.l}$ for every l-block that maps to that cache line

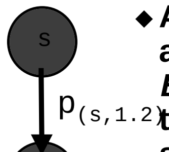
Cache Conflict Graph

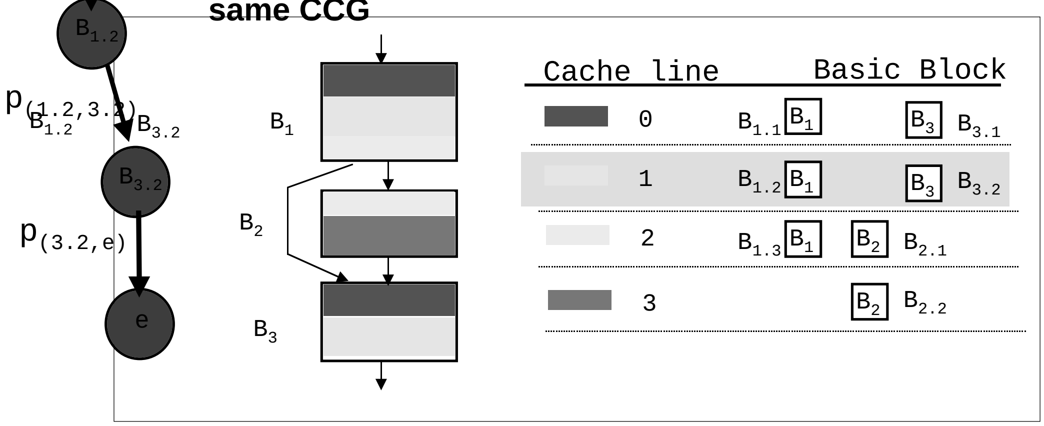
- Edges
  - A directed edge from node $B_{k.l}$ to node $B_{m.n}$ is added if there is a path in the CFG from BB B_k to BB B_m without passing through the BB of any other l-blocks of the same cache line (i.e., CCG)



Cache Conflict Graph

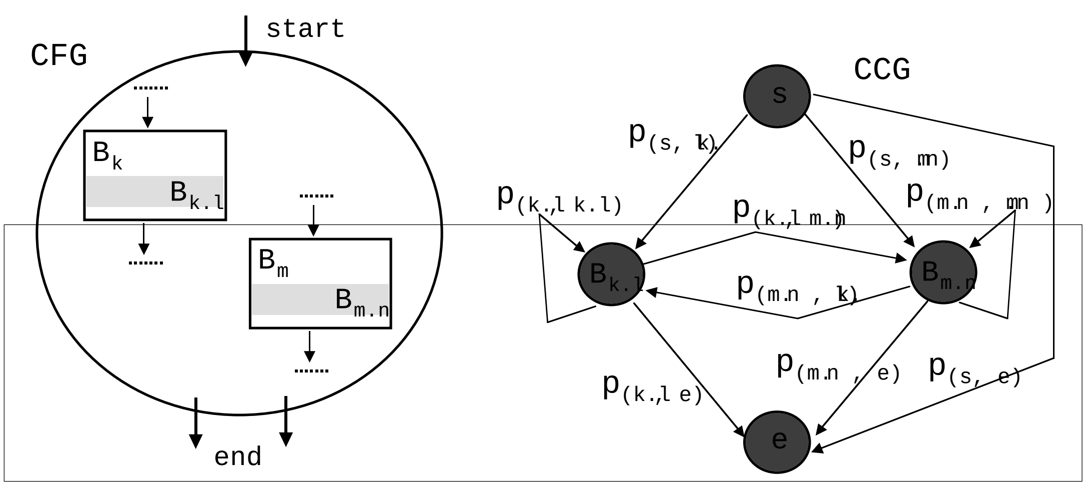
- **Edges**
  - **A directed edge from node $B_{k.l}$ to node e is added if there is a path in the CFG from BB B_k to the end of the CFG without passing through the BB of any other l-blocks of the same CCG**





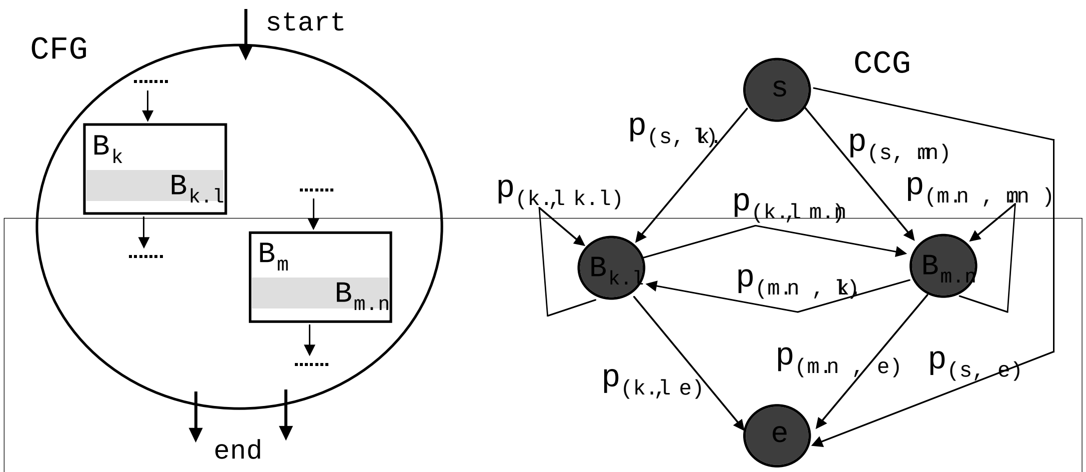
Cache Conflict Graph

- Possible CCG for two conflicting l-blocks



Cache Conflict Graph

- For each edge, a variable p is assigned, representing the number of times that the control flow passes through that edge (CCG has the structure of a network flow graph)



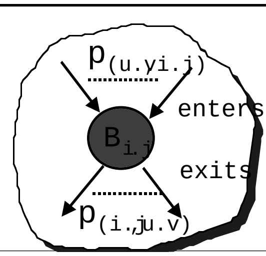
Cache Conflict Graph

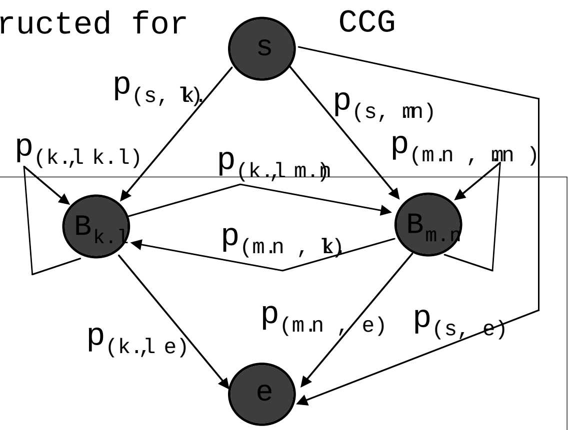
- For each edge, a variable p is assigned, representing the number of times that the control flow passes through that edge (CCG has the structure of a network flow graph)

Two constraints are constructed for each node $B_{i.j}$:

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

execution count of B_i





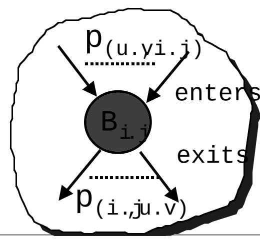
Cache Conflict Graph

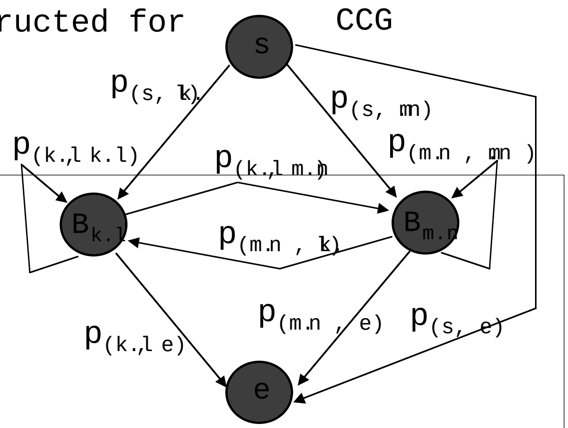
- Constraints to link the p-variables to the program structural and program functionality constraints (via the x-variables)

Two constraints are constructed for each node $B_{i.j}$:

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

execution count of B_i



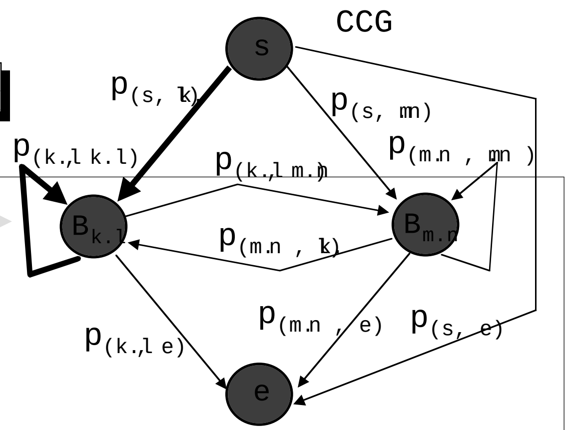


Cache Conflict Graph

- The p-variable $p_{(k.l,\, k.l)}$ represents the number of times that the control flows into l-block $B_{k.l}$ after executing l-block $B_{k.l}$, without entering any other conflicting l-block in between

$$x_{k.l}^{hit} = p_{(k.l,\, k.l)}$$

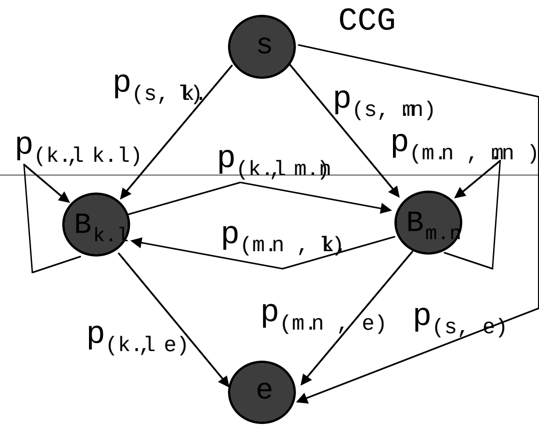
$$p_{(k.l,\, k.l)} + p_{(s,\, k.l)} \geq x_{k.l}^{hit} \geq p_{(k.l,\, k.l)}$$



Cache Conflict Graph

- Starting condition indicating that the program is executed once

$$\sum_{u.v} p_{(s,u.v)} = 1$$



Cache Constraints

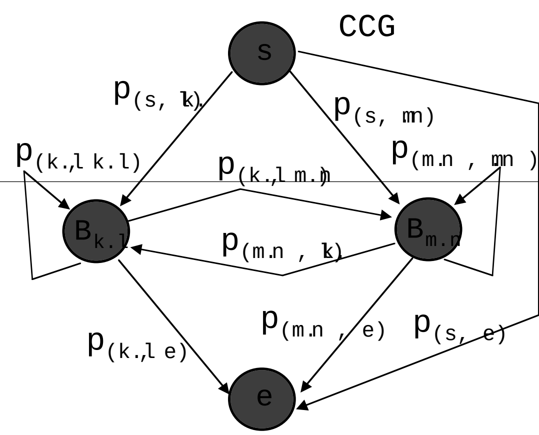
- $$x_i = \sum_{u.v} p_{(u.v,i)} = \sum_{u.v} p_{(i,u.v)}$$

$$p_{(k.l,k.l)} + p_{(s,k.l)} \ge x_{k.l}^{hit} \ge p_{(k.l,k.l)}$$

$$x_{k.l}^{hit} = p_{(k.l,k.l)}$$

- $$\sum_{u.v} p_{(s,u.v)} = 1$$

Bounding the cache miss penalty



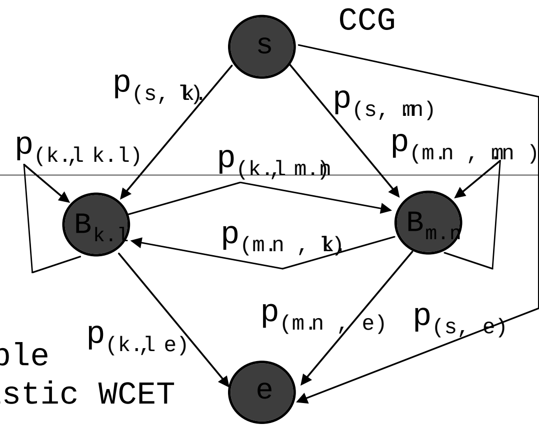
Additional Constraints

- p-variables represent whether or there is a path between two conflicting blocks

The path represented by the p-variable may actually pass through a sequence of basic blocks...

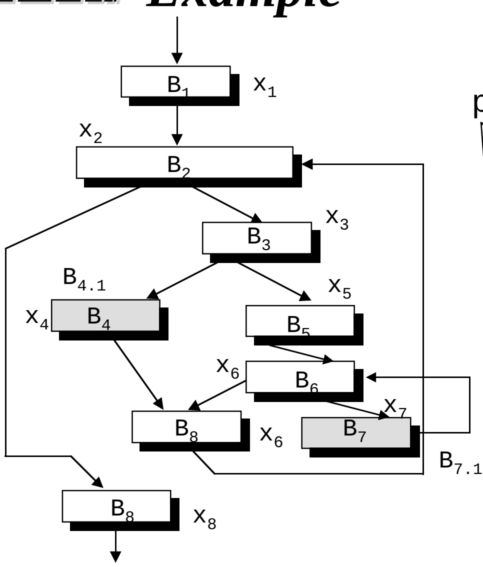
Maximum value of the p-variable is bounded by the minimum of these BBs execution counts

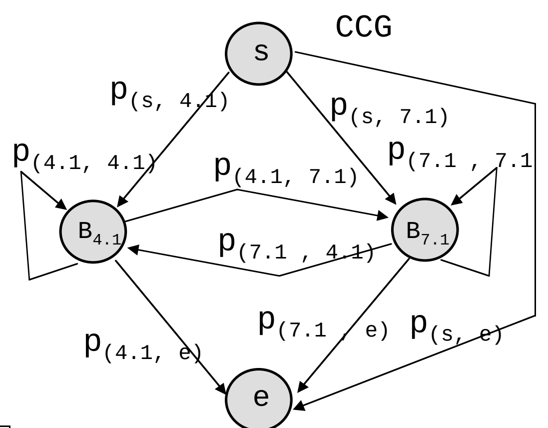
$$0 \le p_{(m.n,\, k.l)} \le \min(x_m, x_k)$$



Or ILP may return unfeasible l-block count and overly pessimistic WCET

Example

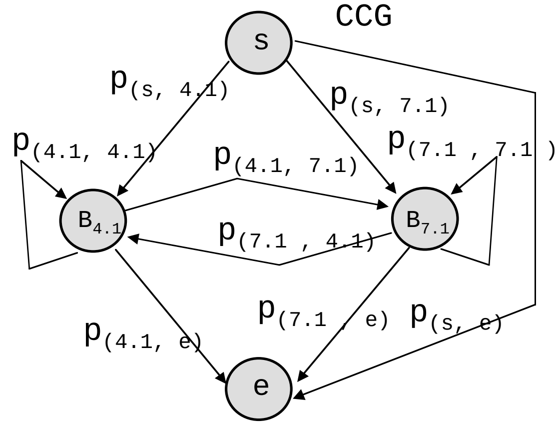




$$X_4 = p(s,4.1) + p(4.1,4.1) + p(7.1,4.1)$$
$$= p(4.1,e) + p(4.1,4.1) + p(4.1,7.1)$$

$$X_7 = p(s,7.1) + p(7.1,7.1) + p(4.1,7.1)$$
$$= p(7.1,e) + p(7.1,7.1) + p(7.1,4.1)$$

Example



Cache Constraints

$X_4 = p(s,4.1) + p(4.1,4.1) + p(7.1,4.1)$
$\quad = p(4.1,e) + p(4.1,4.1) + p(4.1,7.1)$

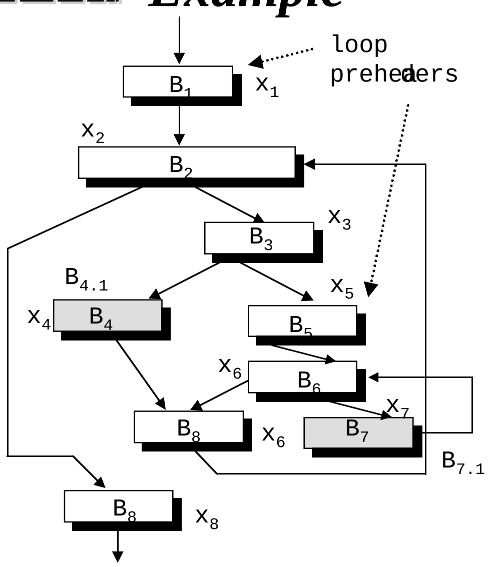
$X_7 = p(s,7.1) + p(7.1,7.1) + p(4.1,7.1)$
$\quad = p(7.1,e) + p(7.1,7.1) + p(7.1,4.1)$

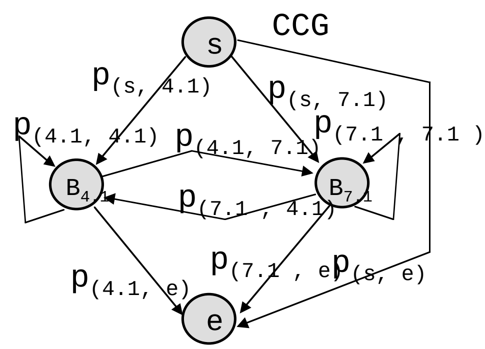
$p(s,4.1) + p(s,7.1) + p(s,e) = 1$

$p(4.1,4.1) \le x_4^{hit} \le p(s,4.1) + p(4.1,4.1)$

$p(7.1,7.1) \le x_7^{hit} \le p(s,7.1) + p(7.1,7.1)$

Example



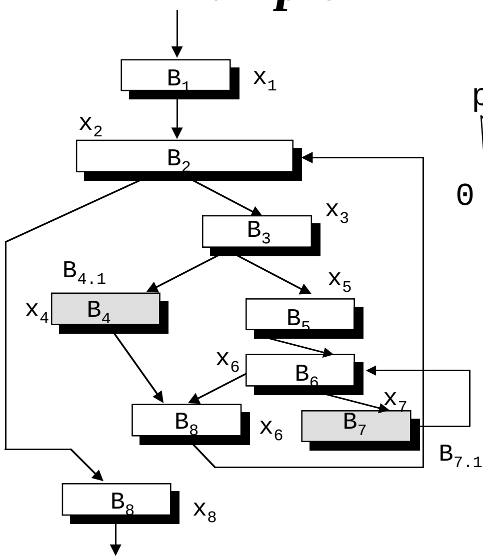


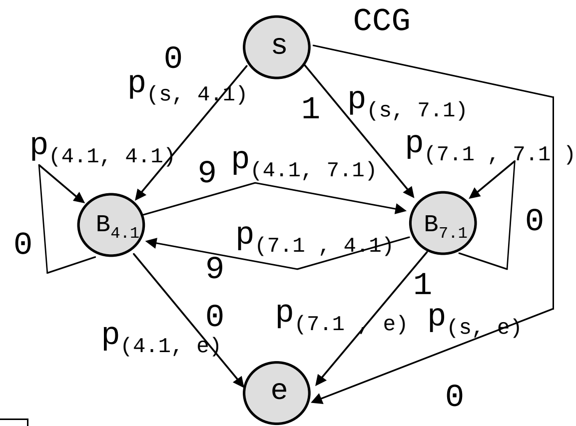
Functional Constraints:
- Both loops will be executed 10 times each time they are entered
- Basic block B4 will be executed 9 times each time the outer loop is entered

$x_3 = 10x_1 \quad x_7 = 10x_5$

$x_4 = 9x_1$

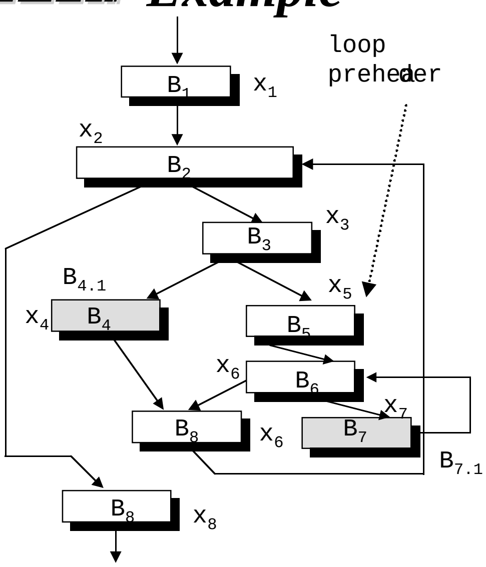
Example

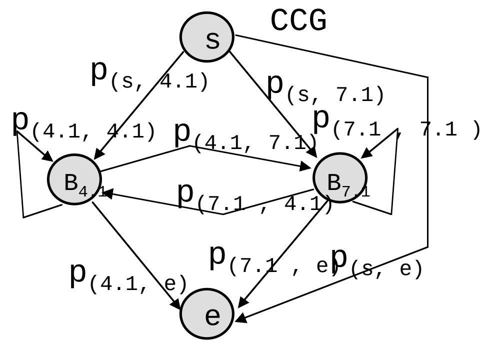




WC Solution: maximum number of cache misses: impossible execution sequence

Example





Placing bounds on variables:

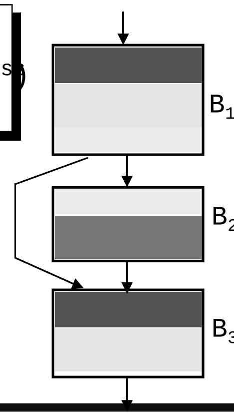
- All paths entering the loop must pass through the preheader (ph)
- Sum of the flows can at most be equal to execution count of ph

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

CPU Pipelining

- Assumption: time required to execute a sequence of instructions in the CPU pipeline is always a constant through the execution of the program

$$\text{Total execution time} = \sum_{i}^{N} \sum_{j}^{n_i} (c_{i.j}^{hit} x_{i.j}^{hit} + c_{i.j}^{miss} x_{i.j}^{miss})$$



N = number of BBs
n_i = number of l-blocks of block B_i
$x_{i.j}^{hit}$ = cache hit count of l-block $B_{i.j}$
$x_{i.j}^{miss}$ = cache miss count of l-block $B_{i.j}$
$c_{i.j}^{hit}$ = hit cost of l-block $B_{i.j}$
$c_{i.j}^{miss}$ = miss cost of l-block $B_{i.j}$

CPU Pipelining

$c_{i.j}^{hit}$ = hit cost of l-block $B_{i.j}$
$c_{i.j}^{miss}$ = miss cost of l-block $B_{i.j}$

- Found by adding up the effective execution times of the instructions in the l-block
- Some instructions (especially floating point instructions) are data dependent
  - conservative approach: worst case execution time (worst case execution time of floating point operation maybe 30-40% its average execution time.)
- Additional time is added to the last l-block of each BB so as to ensure that all the buffered load/store instructions are completed when the control reaches the end of the BB

CPU Pipelining

c_{ij}^{hit} = hit cost of l-block $B_{i.j}$
c_{ij}^{miss} = miss cost of l-block $B_{i.j}$

- Equal to corresponding hit cost plus time needed to load the instructions of the l-block(s) in the cache memory

Cinderella

- **Estimates the WCET of programs running on an Intel QT960 development board containing**
  - **20 MHz Intel i960**
    - **32-bit RISC processor**
    - **on-chip 512-byte direct-mapped instruction cache, organized as 32x16-byte lines**
    - **4 stage instruction pipeline**
    - **floating point unit**
  - **128K of main memory**
- **Uses a public domain ILP solver**

Experimental Results

- "Actual WCET" was identified by authors, and the program execution time was measured for this worst case data set

Benchmarks

Function	Description	Lines	Bytes
check_data	Example from Park's Thesis	**17**	**88**
picksrt	Insertion Sort	**15**	**80**
line	Line drawing routine in Gupta's Thesis	**143**	**1,556**
circle	Circle drawing routine in Gupta's Thesis	**88**	**1,588**
fft	Fast Fourier Transform	**56**	**544**
des	Data Encription Standard	**185**	**1,796**
fullsearch	MPEG2 encoder frame search routine	**204**	**1,436**
whetstone	Whetstone benchmark	**245**	**2,760**
drystone	Drystone benchmark	**480**	**1,360**
matgen	Matrix routine in Linpack benchmark	**50**	**248**

Experimental Results

tighter bounds

Function	Measured WCET	Estimated WCET with cache analysis	Estimated WCET without cache analysis
check_data	**4.41×10^2**	**4.91×10^2**	**11.9×10^2**
picksrt	**1.79×10^3**	**1.82×10^3**	**5.01×10^3**
line	**4.85×10^3**	**6.09×10^3**	**9.15×10^3**
circle	**1.45×10^4**	**1.53×10^4**	**1.59×10^4**
fft	**2.08×10^6**	**2.71×10^6**	**4.04×10^6**
des	**2.42×10^5**	**3.66×10^5**	**6.69×10^5**
fullsearch	**6.25×10^4**	**9.57×10^4**	**29.0×10^4**
whetstone	**6.83×10^6**	**10.2×10^6**	**14.9×10^6**
drystone	**5.52×10^5**	**7.53×10^5**	**13.3×10^5**
matgen	**9.28×10^3**	**10.9×10^3**	**17.2×10^3**

clock cycles

Experimental Results

small integer progr

Function	Measured WCET	Estimated WCET with cache analysis	Estimated WCET without cache analysis
check_data	**4.41×10^2**	**4.91×10^2**	**11.9×10^2**
picksrt	**1.79×10^3**	**1.82×10^3**	**5.01×10^3**
line	**4.85×10^3**	**6.09×10^3**	**9.15×10^3**
circle	**1.45×10^4**	**1.53×10^4**	**1.59×10^4**
fft	**2.08×10^6**	**2.71×10^6**	**4.04×10^6**
des	**2.42×10^5**	**3.66×10^5**	**6.69×10^5**
fullsearch	**6.25×10^4**	**9.57×10^4**	**29.0×10^4**
whetstone	**6.83×10^6**	**10.2×10^6**	**14.9×10^6**
drystone	**5.52×10^5**	**7.53×10^5**	**13.3×10^5**
matgen	**9.28×10^3**	**10.9×10^3**	**17.2×10^3**