Real-Time and Embedded Systems @ SIT

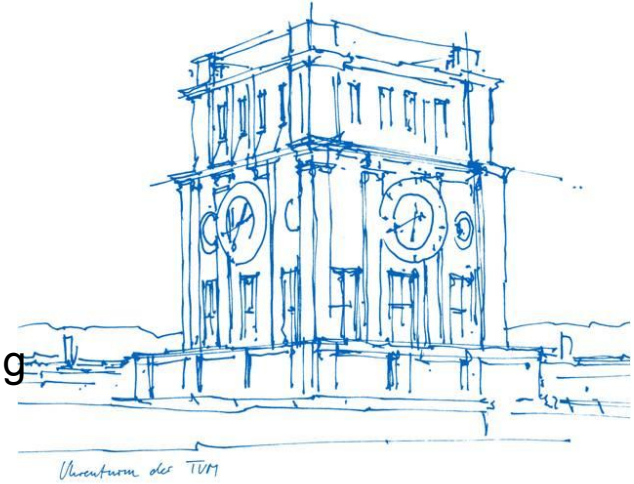
Worst Case Execution Time (WCET)

Alexander Hoffman

Technical University of Munich

Department of Electrical and Computer Engineering

Chair of Real-Time Computer Systems



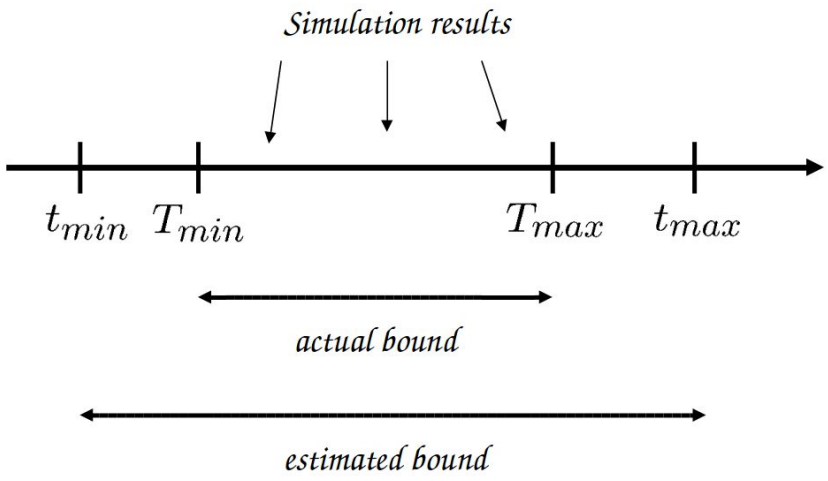
Worst Case Execution Time Analysis

- Look at how to estimate the WCET of a task
  - Similar to BCET
- WCET of a task depends on:
  - The **program code**
    - Depends on the input, thus the WCET is the maximum execution for *ALL POSSIBLE INPUTS (*often infinite possibilities)
  - The **architecture of the processor** on which the task is running
    - Eg. more cache typically results in smaller WCET

WCET Estimation

- Determining from observations?
  - Run the program for all possible inputs
  - There can be infinitely many inputs
  - Thus, not practically feasible
- Need to *analytically* determine or estimate the WCET
  - Using static analysis, ie. mathematically analyze program code and processor architecture without running the program
- Aim is to obtain *safe* bounds on the estimates
  - Pessimistic
  - Important in safety-critical applications

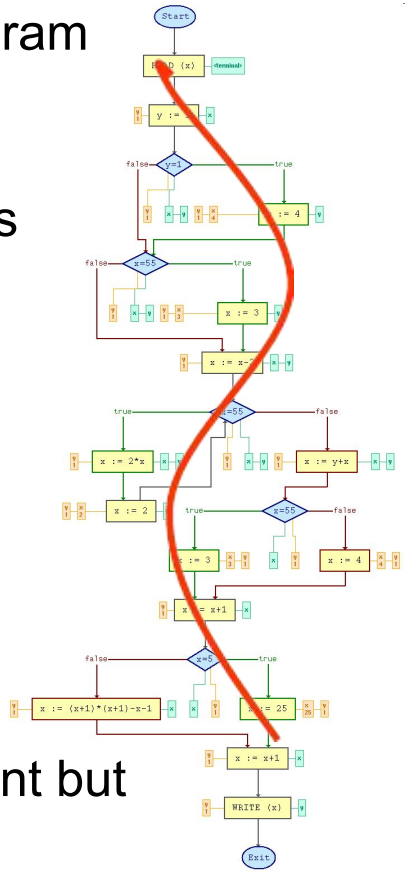
WCET Estimation



- Estimated bounds should *enclose* the actual bounds (*safe*)
- Aim to obtain estimated bounds that enclose the actual bounds as *tightly* as possible
  - t_{min} as close to T_{min} as possible
  - t_{max} as close to T_{max} as possible

Estimating Execution Time

Path through program



Two sub-problems:

- Estimating the execution times of different program paths
  Challenges:
  - Exponentially many paths (no path enumeration)
  - Paths might be infeasible
- Estimating the execution time of each instruction along an execution path (microarchitecture modeling)
  Challenges:
  - Modelling effects of caches, pipelines …
    (ie. the execution time of an instruction is not constant but dependant on many factors of the system's context)

Path Enumeration

```
for (i = 0; i < 100; i++){
   if (rand() > 0.5)
      j++;
   else
      k++;
}
```

- How many paths are in this program?
  - No of feasible paths = 2^{100} (that's a lot!)
- In general, path count is exponential to the size of the program
- Enumerating all of them and identifying the max execution time is not feasible

Correlations Between Paths

- Several program paths might be infeasible
  Eg. `i = i*i + 1;`
  Followed by
  `j = j*j;`

- Infeasible paths will lead to overestimation of WCET

```
if (ok)
  i = i*i + 1;
else
  i = 0;
/* ... */
if (i)
  j++;
else
  j = j*j;
```

Execution Count Analysis

- Assumption: execution of an instruction is constant and independent of other instructions on the systems (ie. no microarchitecture modeling)
- Observation: Instructions within a single line of code (ie. no branches) have the same instruction count
- A “straight-line code sequence” is referred to as a **basic block**:
  - maximum sequence of instructions where the entry point is the first instruction and the exit point is the last (cannot be jumped into or out of)
  - Each instruction executes before those in later positions
  - No other instructions execute between two instructions in the sequence

Execution Count Analysis

- Basic blocks execution time and count tells us program execution time

Total execution time of a program = $\sum_{i=1}^{N} c_i x_i$

$\mathbf{X_i}$ is the execution count of the basic block B_i
$\mathbf{c_i}$ is the execution time of the basic block B_i

Path Analysis Versus Count Analysis

- Number of program paths = 2^{100}
- Count-based analysis
  - Let *a* = number of times j++ is executed
  - Let *b* = number of times k++ is executed
  - *a* + *b* is always equal to 100
  - Number of different possible valuations of *a* and *b* is 101, ie. (*a,b*)=(0,100), (1,99), …, (100, 0)
  - These 101 different valuations capture all 2^{100} paths
  - Execution time of program can be computer from these 101 valuations (instead of 2^{100} paths). Reduced number of possibilities

```
for (i = 0; i < 100; i++){
   if (rand() > 0.5)
      j++;
   else
      k++;
}
```

WCET Analysis as an Optimization Problem

- Based on the count analysis, the WCET analysis problem may be formulated as the following optimization problem:

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

  $\mathbf{X_i}$ is the execution count of the basic block B_i
  $\mathbf{c_i}$ is the execution time of the basic block B_i
  For feasible values of x_is (no infeasible paths so not all x_is may be feasible)
- Feasible values of x_is are obtained from the program's control flow and logical flow (next)
- *Control flow* - order in which instructions of a program are executed

Program Control Flow and Logical Flow

- Thus, $x_i s$ represents the execution counts of the instructions
- **Control flow constraints** - derived from the program's structure
  - Eg. x3 = x4 + x5
- **Logical constraints** - arisings from the *logical flow* of the program
  - Eg. x5 <= 1 (false statement executed at most once)
  - X4 >= 99 (ie. the true statement is executed at least 99 times)

```
x1      i = 0;
x2      while (i < 100) {
x3         if (ok) {
x4            j++;
         } else {
x5            k++; ok = true;
         }
x6           i++;
        }
```

Integer Linear Programming Formulation

- Now we know our problem:
  Maximize $\sum_{i}^{N} = 1 c_i x_i$ <u>subject to control flow and logical flow constraints</u>

- Objective function is linear and all constraints are also linear -> integer linear programming (ILP) problem
- ILP solver is guaranteed to determine the extreme case solution (ie. maximize the objective function)
- Note: two classes of linear constraints
  - *Structural* or *control flow constraints:* related to program's control flow
  - *Logical or functionality constraints:* related to program's logical flow
- A number of ILP solvers freely available, Eg. LPSolve

Structural/Control Flow Constraints

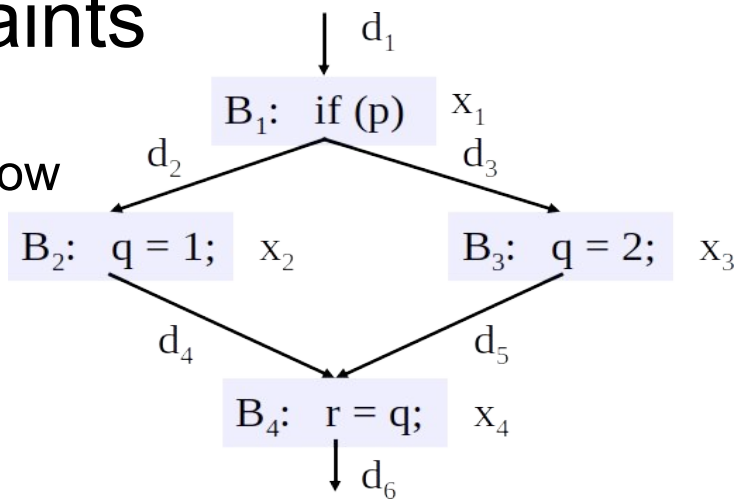
- Provides a complete characterization of bounds on basic block variables imposed by the program structure
  - Program structure refers to the structure of the assembly language program (instructions to be run on CPU)
  - Analysis starts with creating the control flow graph (CFG) of the program (we use source code to illustrate the principle, but in practice the CFG is generated from program's assembly code)

(Integer) Linear Programming

- Linear programming: a mathematical model where all relationships (constraints) are given by linear relationships
  - Linear relationships: functions whose graph gives a straight line, Ie. has a polynomial degree of zero or one.
- Finds a value in the feasible objective space, defined by the linear equality and inequality constraints, where the linear objective function is its smallest (or largest) value, if such a point exists.
- Integer linear program means that all variables are restricted to/take integer values (values represented without using a fraction)

Structural Constraints

Control Flow Graph

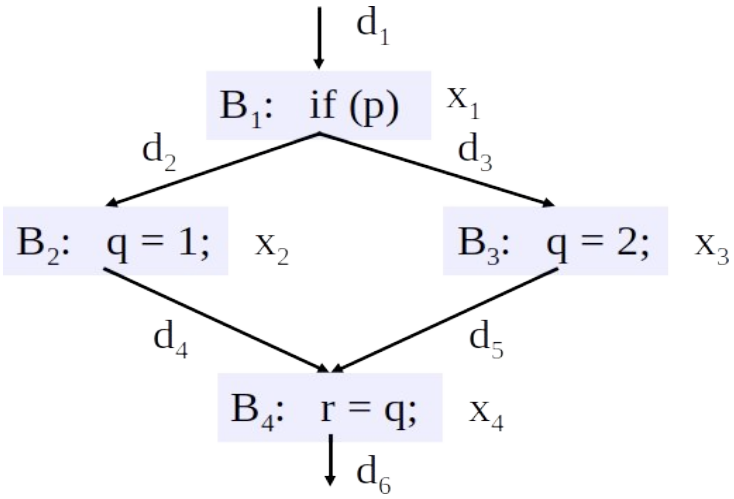



```
if (p)
  q = 1;
else
  q = 2;
r = q;
```

x_i is the number of times basic block B_i is executed
d_i is the number of times the particular edge is followed

At each node of the CFG, the sum of the control flow going into the node must be equal to the control flow exiting the node. This sum must also equal the execution count of the basic block.

Structural Constraints



$$x_1 = d_1 = d_2 + d_3$$
$$x_2 = d_2 = d_4$$
$$x_3 = d_3 = d_5$$
$$x_4 = d_4 + d_5 = d_6$$

8 structural constraints

Logical Constraints

- Provide a mechanism to bound basic block variables based on the program's logical flow
- Related to data variables, which in many cases depend on the input data set
- Typical constraints: loop bounds and path information
- t45
- Do not pinpoint extreme execution point paths but rather bounds all possible execution paths
- Loop bounds are *mandatory* information, required to make the problem solvable - otherwise the ILP solver assumes that loops iterate forever
- Note: since all loop bounds cannot be statically determined, determining the WCET of a program, in general, is *undecidable*

Logical Constraints

- Unlike structural constraints, which are formulated automatically from the assembly code, logical constraints are formulated at a source code level
- Therefore, a mapping method is required to project the assembly code back to the source codes

Solving the ILP Problem

- In general, problem is NP-hard, but some cases permit polynomial time solutions
  - NP-hard: non-deterministic polynomial time
- Since the structural constraints are derived from a control flow graph, they exhibit very good integer properties
  - Ie. if the ILP is solved as an LP then it returns an integer valued solution. Thus, the ILP collapses to an LP of polynomial complexity
    - Solvable in polynomial time, $T(n) = O(n^k)$, for some positive constant k

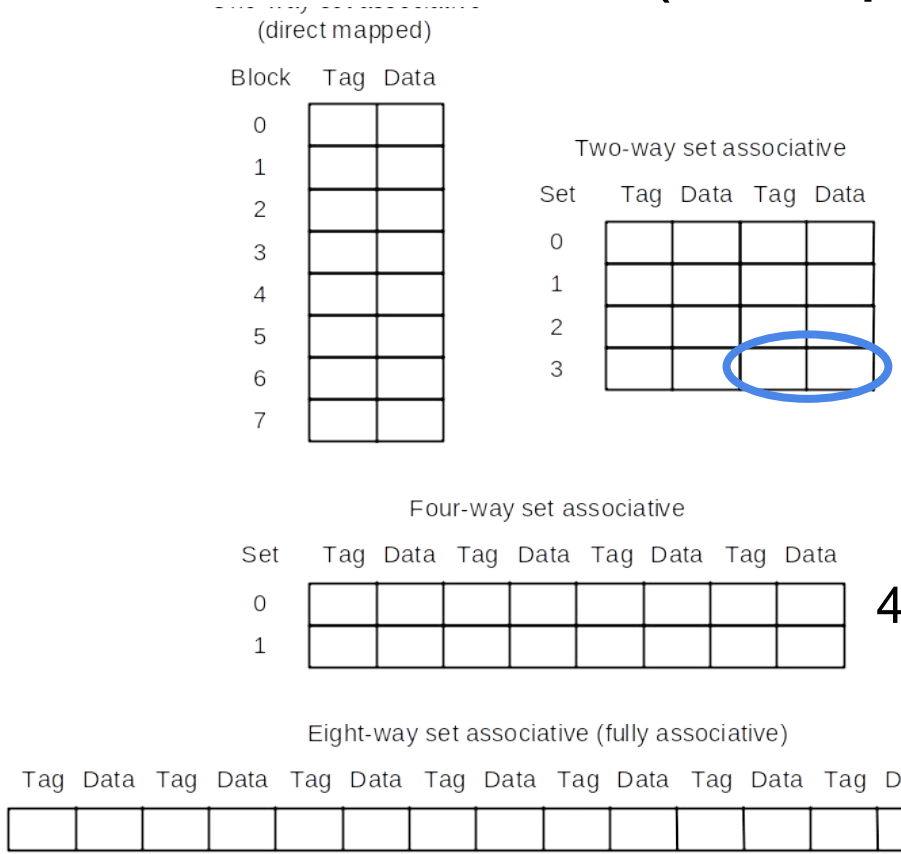
Microarchitecture Modeling - Caches

- Types of caches:
  - Split cache - physically separate at a hardware level, logically one cache
    - Instruction cache - reads program instructions from memory, usually done in chunks (sequential instructions)
    - Data cache - reads and writes data from memory
  - Unified cache - not split at hardware level
- We will only look at the influence of instruction cache

Basics of Caches (Recap)

- Cache organization (Recap):
  - A cache consists of a number of *cache sets*
  - *n*-way set associative cache: each cache set consists of *n* cache lines
  - 1-way set associative cache: (direct-mapped cache):
    - Each cache set consists of exactly one cache line
  - Each cache line holds a fixed number of bytes (4, 8, .., 128, etc.)
  - Cache size = number of sets X set associativity X line size

Basics of Caches (Recap)



2 blocks/set

block

4 blocks/set

8 blocks/set

Basics of Caches (Recap)

- Each address of the main memory is mapped to a unique cache set, and to a unique offset position in a cache line
- For a n-way set associative cache, each memory location can be mapped to any of the n cache lines within the unique cache set to which this memory location is mapped
- For a direct mapped cache, there is just one cache line and therefore the memory location is mapped to this line
- If the contents of a memory location is found in the cache, it results in a cache *hit*, otherwise resulting in a cache *miss*.
    - In the case of a *miss* the portion of required memory, equal to the cache line size, is read into cache from memory

Difficulty in Cache Modeling for WCET Analysis

- We will study how to model direct mapped cache
- Difficulty comes from:
  - Identifying the “worst case” execution path and generate the memory trace due to this path
  - Simulating this memory trace on a cache simulator to obtain the number of cache hits and misses
  - However, a longer executing path (more instructions) might not incur as many cache misses while a shorter path might incur more and thus have a longer execution time, despite its shorter length
  - Therefore, program path analysis and microarchitecture modelling cannot be done separately

Difficulty in Cache Modeling for WCET Analysis

- This would lead us to an approach that:
  - Identifies all possible paths
  - Generates memory traces for each path
  - Compute the number of cache hits and misses for each cache model
- Problem: The number of execution paths might be exponential to the size of the program making this method not computationally feasible

Using the ILP Technique Again….

- Another approach:
  - Use the ILP formulation developed in program path analysis phase
  - Use linear expressions to *bound* the feasible cache activity
    - Ie. provide constraints on the maximum and minimum number of cache hits and misses
  - Structural and functionality constraints are also derived from program path analysis
  - Given the known constraints and an objective function, let the ILP solver explore the solution space to a hopefully optimal solution (when possible)

Using the ILP Technique Again….

- Gives us the advantages:
  - ILP solver considers the problem at a program level, “global”
  - Retains path information derived during the program path analysis phase
  - Leads to a tighter estimation on the execution time

Direct Mapped Cache: Recap

- System has 2^m addresses
- Block has a size of 2^n
- Cache has 2^k entries (each entry holds one block)
- An address:

Tag, remaining bits (2^{m-n-k} possibilities)	Index, k bits (2^k possibilities)	Offset, n bits (2^n possibilities)

- Offset: where in block data is located
- Index: where our block would be located in cache (ie. which cache line)
- Tag: identifies that the correct memory block is/is not in cache

Direct Mapped Cache: Recap

Address	Line
00000	
00001	
00010	
00011	
00100	
00101	
00110	
00111	
01000	
01001	
01010	
...	

Example: m = 5, k = 2, n = 1

Direct Mapped Cache: Recap

Address	Line
00000	
00001	
00010	
00011	
00100	
00101	
00110	
00111	
01000	
01001	
01010	
...	

Example: m = 5, k = 2, n = 1

Block size: $2^1 = 2$

Direct Mapped Cache: Recap

Address	Line
00000	
00001	
00010	
00011	
00100	
00101	
00110	
00111	
01000	
01001	
01010	
...	

Example: m = 5, k = 2, n = 1

Block size: $2^1 = 2$

Offset is either 0 or 1 in each block

Direct Mapped Cache: Recap

Address	Line
00000	
00001	
00010	
00011	
00100	
00101	
00110	
00111	
01000	
01001	
01010	
...	

Example: m = 5, k = 2, n = 1

No of entries: $2^2 = 4$, we have 4 cach lines, ie. 4 blocks can be in cache at any time

Each block can only be on the line that matches its index

Direct Mapped Cache: Recap

Address	Line
00000	
00001	
00010	
00011	
00100	
00101	
00110	
00111	
01000	
01001	
01010	
...	

Example: m = 5, k = 2, n = 1

No of entries: $2^2 = 4$, we have 4 cach lines, ie. 4 blocks can be in cache at any time

Each block can only be on the line that matches its index

Direct Mapped Cache: Recap

Address	Line
00000	
00001	
00010	
00011	
00100	
00101	
00110	
00111	
01000	
01001	
01010	
...	

Example: m = 5, k = 2, n = 1

No of entries: $2^2 = 4$, we have 4 cach lines, ie. 4 blocks can be in cache at any time

Each block can only be on the line that matches its index

We have lines 00, 01, 10 and 11

Direct Mapped Cache: Recap

Address	Line
00000	
00001	
00010	
00011	
00100	
00101	
00110	
00111	
01000	
01001	
01010	
...	

Example: m = 5, k = 2, n = 1

Remaining bits are the **tag**

Used to see if a block in cache is the one we want

Direct Mapped Cache: Recap

Address	Line
00000	
00001	
00010	
00011	
00100	
00101	
00110	
00111	
01000	
01001	
01010	
...	

Example: m = 5, k = 2, n = 1

Remaining bits are the **tag**

Used to see if a block in cache is the one we want

We have $2^{5-3-1} = 4$ unique tags

Direct Mapped Cache: Recap

Address	Line
00000	
00001	
00010	
00011	
00100	
00101	
00110	
00111	
01000	
01001	
01010	
...	

Example: m = 5, k = 2, n = 1

Eg. the block at address 0 is already in cache, on cache line 00

If we wanted to load the block at 01000 it would also get put into the same cache line, but since its tag (01) is different to the other tag (00) we get a cache miss

Direct Mapped Cache: Recap

Thus, process of checking if cache hits or misses is:

- Use index to find appropriate cache line
- Compare tag to determine if we have a cache hit or miss
- If hit:
  - Use offset to access data we are after
- If miss:
  - Load data into cache from memory

Direct Mapped Instruction Cache Analysis

- New objective function:
  - Each instruction can have two execution times:
    - Cache hit
    - Cache miss
  - Thus, total execution time given by

$$\sum_{i=1\ldots N} c_i^{hit} x_i^{hit} + c_i^{miss} x_i^{miss}$$

C_i^{hit} execution time when instruction *i* results in cache hit

C_i^{miss} execution time when instruction *i* results in cache miss

x_i^{hit} number of times instruction *i* results in cache hit

x_i^{miss} number of times instruction *i* results in cache hit

Direct Mapped Instruction Cache Analysis

- In previous method we used one variable for each instruction -> too many variables! Can we reduce this number?
- Grouping instructions together
  - Recall that from the program path analysis we considered execution times and counts of basic blocks, not individual instructions!
  - Can we group instructions in a similar way now?
  - It would reduce the number of variables in the ILP formulation, thus reducing its complexity
  - How? Group adjacent instructions which have identical cache hit and miss counts
  - Grouped instructions must definitely come from the same basic block

Direct Mapped Instruction Cache Analysis

- But will this method group many instructions together?
  - No
  - Why?
    - Cache controller loads a full cache line into the cache whenever there is a cache miss (to exploit locality of reference)
  - As a result, only accesses to
    - Instructions at the beginning of cache lines
    - First instructions in a basic block

    Can result in cache misses
  - Remaining instructions will hit (memory locality)

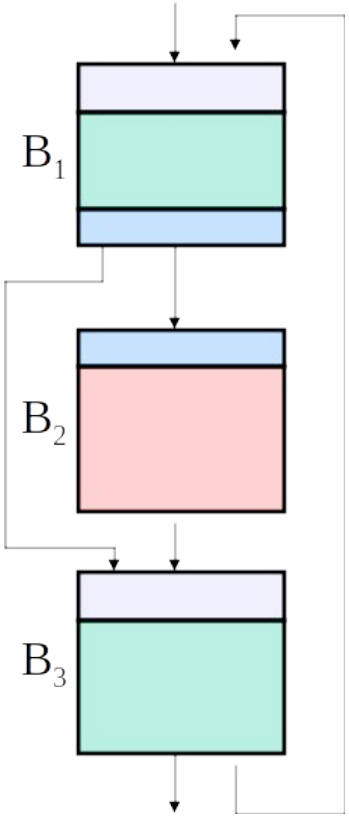
Direct Mapped Instruction Cache Analysis

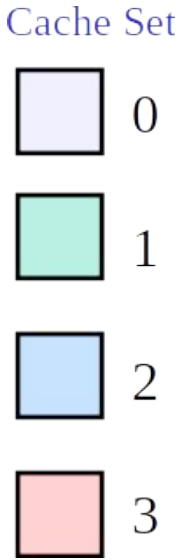
- A different view of grouping
  - Instead of “each instruction” having two possible execution times, each “block of instructions” to have two possible execution times
    - The first instruction of the block has a cache miss and all others have cache hits
    - The first instruction of a block has a cache hit and all others also have a cache hit
  - Only two possibilities for each block
  - Based on this:
    - If a basic block maps onto *l* cache sets, then it will be partitioned exactly into *l* smaller blocks, each which gets a single variable in the ILP

Direct Mapped Instruction Cache Analysis

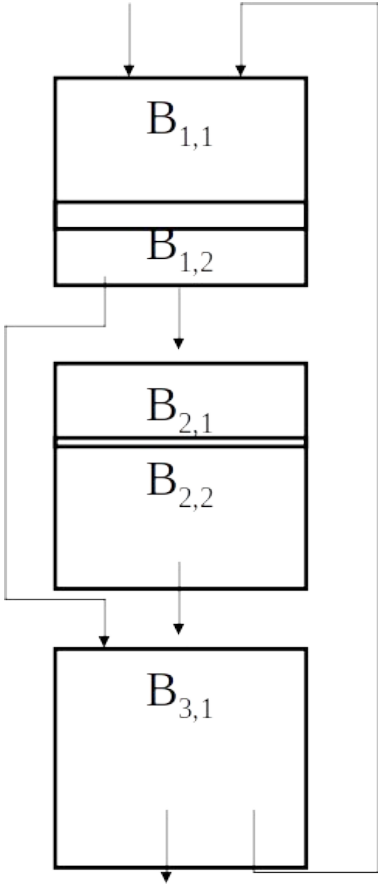
- Can we do even better?
  - Or requirement: after grouping each block can have only two possible execution times
  - Observation: if two or more cache sets are mapped onto the same set of basic blocks then these cache sets must have identical cache activities
    - Ie. if there is a cache hit on the first cache then there is also a cache hit on the next cache
  - Thus: when the first instruction in the first basic block is accessed, either all instructions in the block are in cache or none of them
  - Conclusion: there are only two possible execution times for these blocks, thus they can be grouped!

Example of Grouping

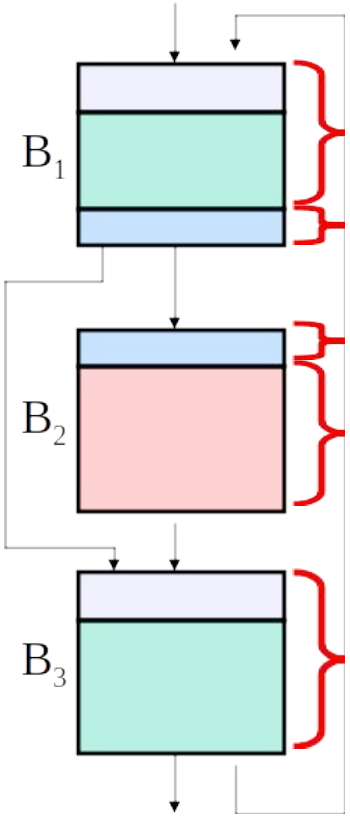




Cache Set	Basic block	
0	B1	B3
1	B1	B3
2	B1	B2
3		B2



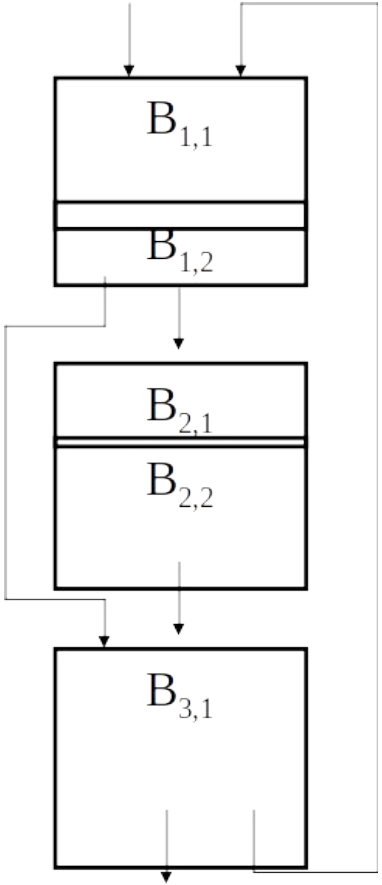
Example of Grouping



Groups represent instruction groupings that are ALWAYS executed together

Cache Set

Cache Set	Basic block	
0	B1	B3
1	B1	B3
2	B1	B2
3		B2



Direct Mapped Instruction Cache Analysis

- Better groupings (into line blocks)
  - These grouping we just looked at are dependent on the cache lines that they use
  - We can refer to them as "**line-block**" or "**l-block**"
  - A l-block is the maximum sequence of code within a basic block such that when the first instruction of the l-block is accessed, either the whole l-block is in cache or none of its contents is
  - A basic block B_i is thus partitioned into l-blocks $B_{i,1}$, $B_{i,2}$, … , $B_{i,n}$
  - The execution times of an l-block $B_{i,j}$ are given by $c^{hit}_{i,j}$ and $c^{miss}_{i,j}$

Our New WCET Objective Function

Hence, our new objective function is as follows:

$$\text{WCET} = \sum_{i=1}^{N} \sum_{j=1}^{n_i} (c_{i,j}^{hit} x_{i,j}^{hit} + c_{i,j}^{miss} x_{i,j}^{miss})$$

Where there are N basic blocks in the program and each basic block B_i contains n_i line blocks

Linking with Program Path Analysis

- Since an l-block $B_{i,j}$ is inside basic block B_i we can derive:

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

- Further, x_is is constraints by the structural and logical constraints we have seen before

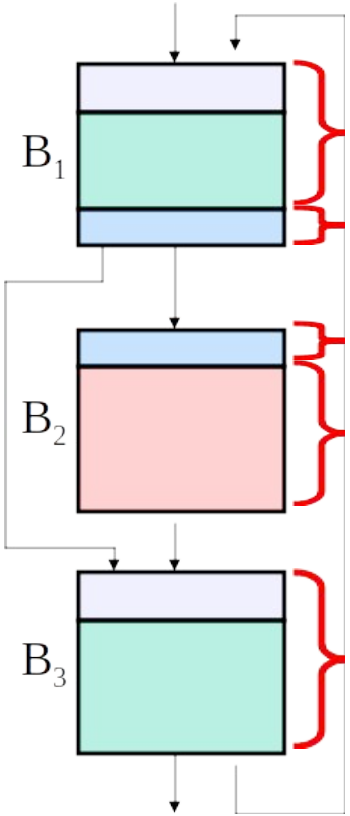
Cache Constraints

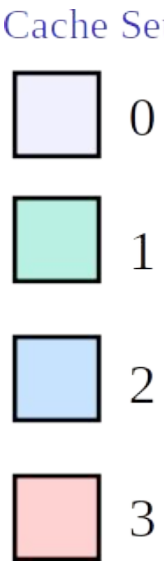
- For any two I-blocks that map onto the same cache set, we say that they *conflict* with each other if they have different address tags
  - Remember: address tags are the portion of the address that identifies the unique memory chunks that map into the same cache lines
- Two I-blocks might also not conflict with each-other even if they map onto the same cache set. Such I-blocks are called non-conflicting

Example

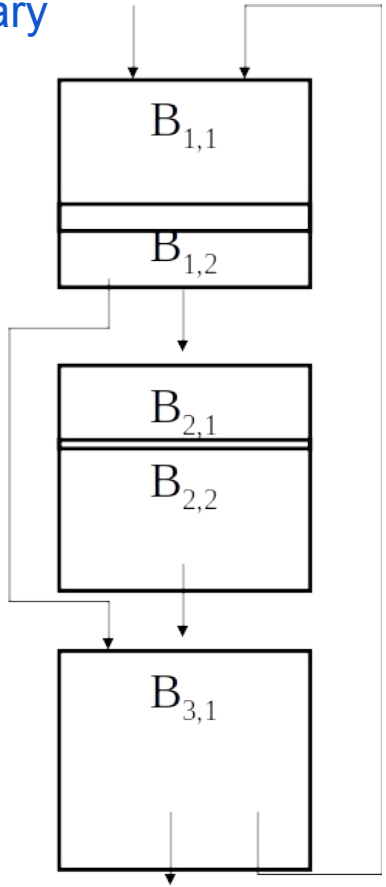
$B_{1,1}$ and $B_{3,1}$ are conflicting

Can happen when basic block boundary is not aligned with cache boundary

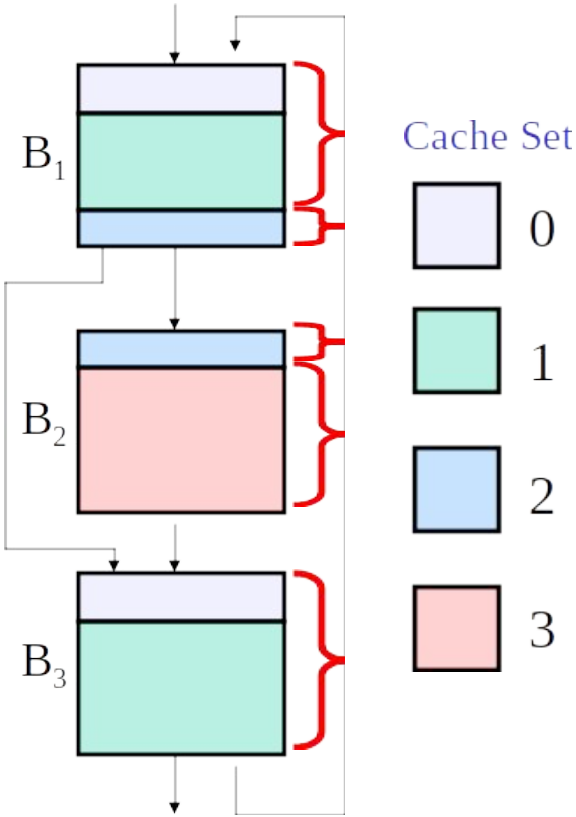




Cache Set	Basic block	
0	B1	B3
1	B1	B3
2	B1	B2
3		B2

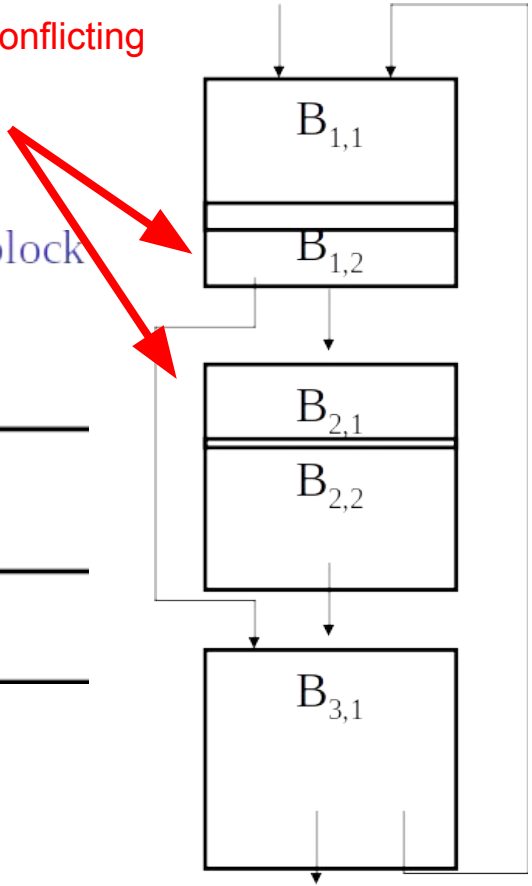


Example



$B_{1,2}$ and $B_{2,1}$ are non-conflicting

Cache Set	Basic block	
0	B1	B3
1	B1	B3
2	B1	B2
3		B2



Cache Constraints

- For any cache set, if there is only one l-block mapped onto it, once this block is loaded into the cache, it will stay there forever
  - This is given by $x_{k,l}^{miss} \leq 1$ if the l-block is $B_{k,l}$
- Now consider the case where two or more *non-conflicting* blocks map into the same cache set. Since the cache controller always fetches a line of code into the cache, the sum of their cache miss counts is at most one

$$\sum_{u,v} x_{u,v}^{miss} \leq 1$$

where $B_{u,v}$*s* are the non-conflicting l-blocks

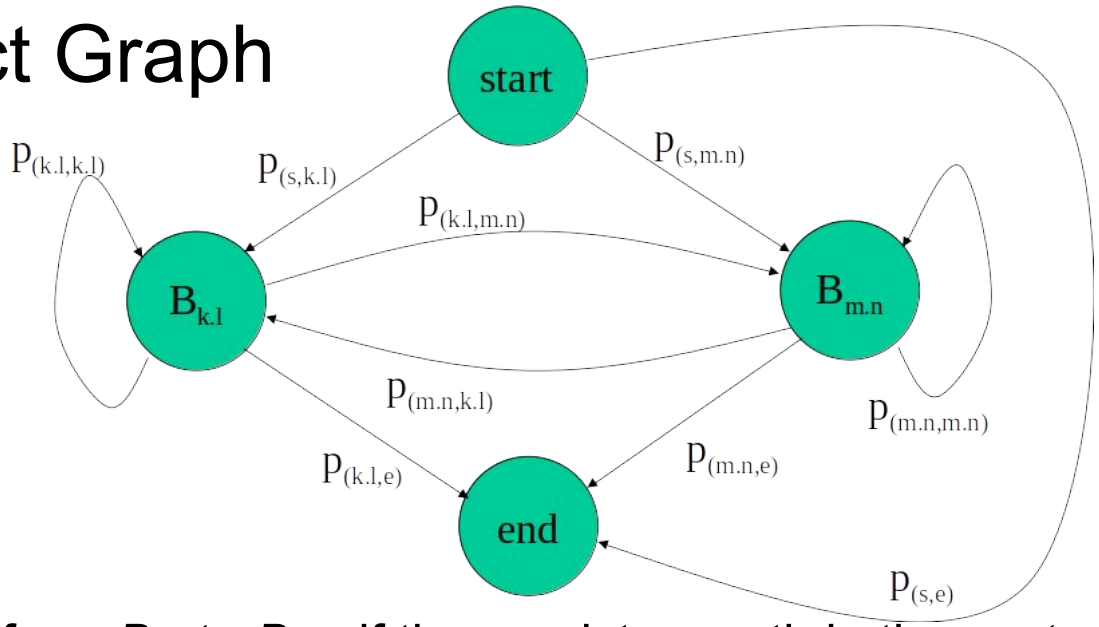
Cache Constraints

- Observation: when a cache set contains two or more conflicting l-blocks, the hit and miss counts of all l-blocks mapping onto this set depends on the execution sequence of these l-blocks
- This is captured in a cache conflict graph

Cache Conflict Graph

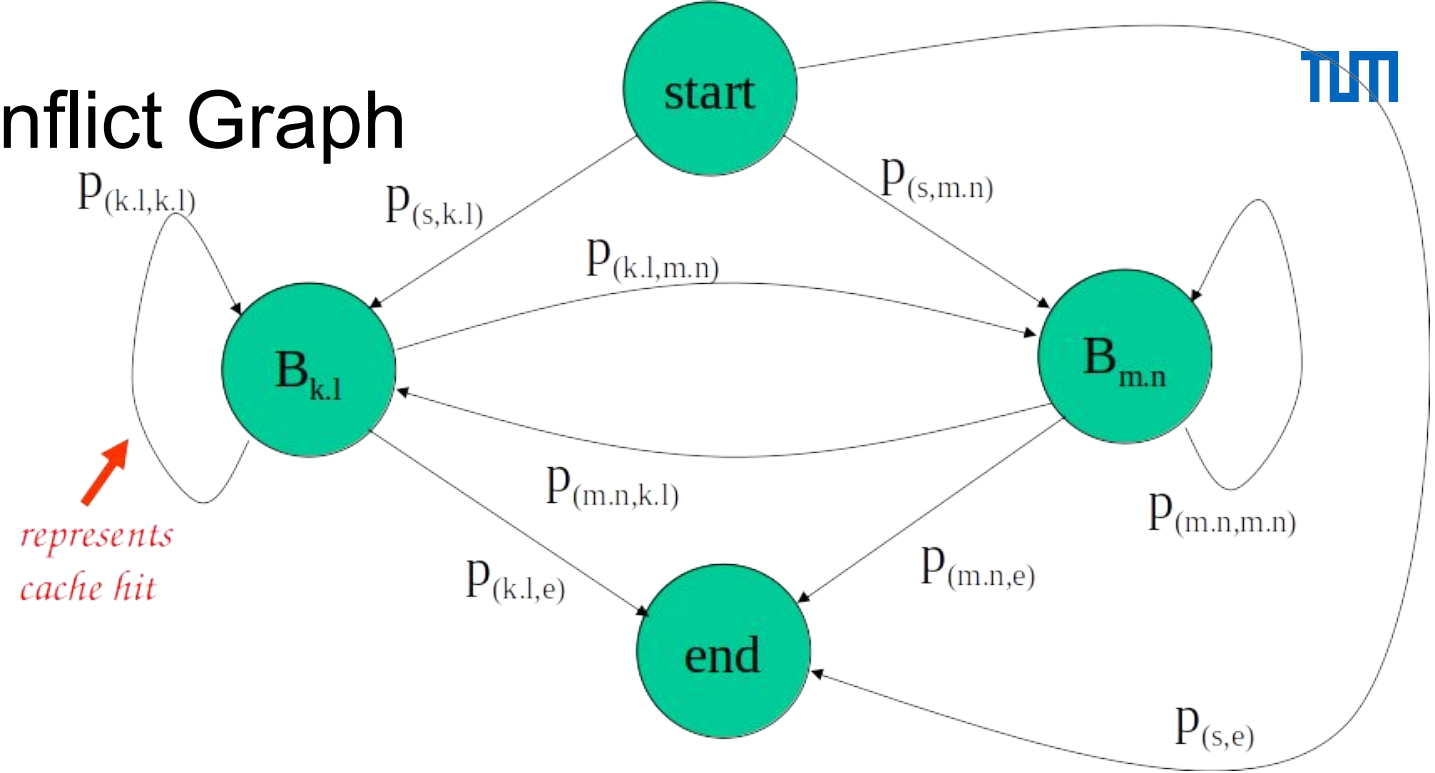
- Constructed for every cache set containing two or more conflicting I-blocks
- It is a reduced control flow graph capturing only the control flow of I-blocks mapped onto the same chace set
- Thus, one graph per cache set

Cache Conflict Graph



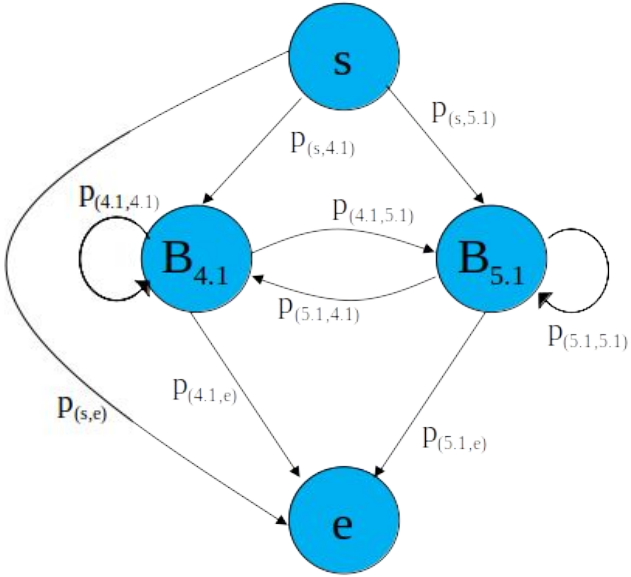
- There is an edge from $B_{k,l}$ to $B_{m,n}$ if there exists a path in the control flow graph from basic block B_k to basic block B_m without passing through the basic blocks of any other l-blocks of the same cache set
- Essentially: we are wanting to generate a graph that is applicable to only a single cache set

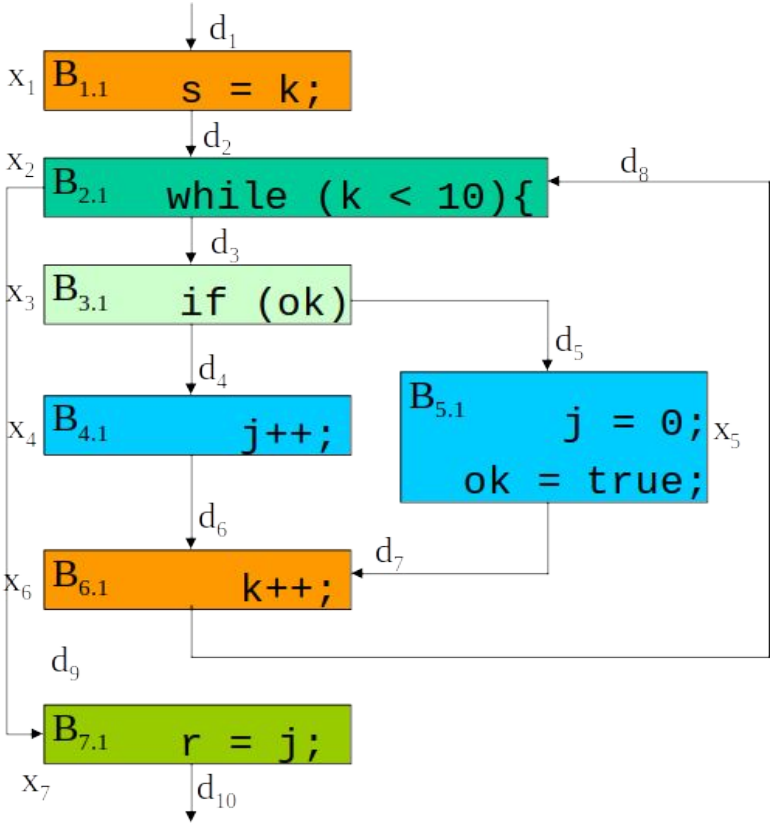
Cache Conflict Graph



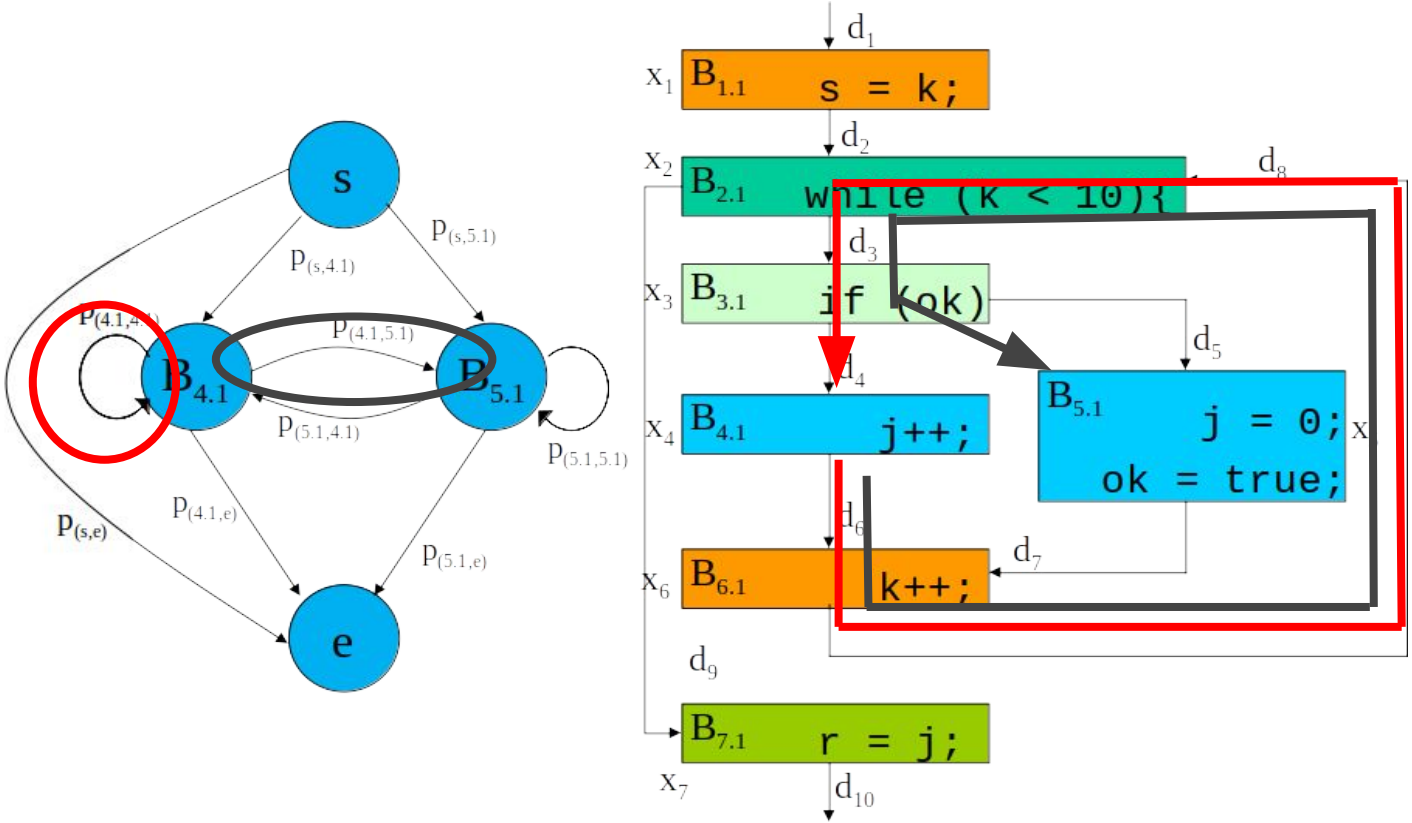
- Self-loops to a node denote guaranteed cache hits

Example Cache Conflict Graph

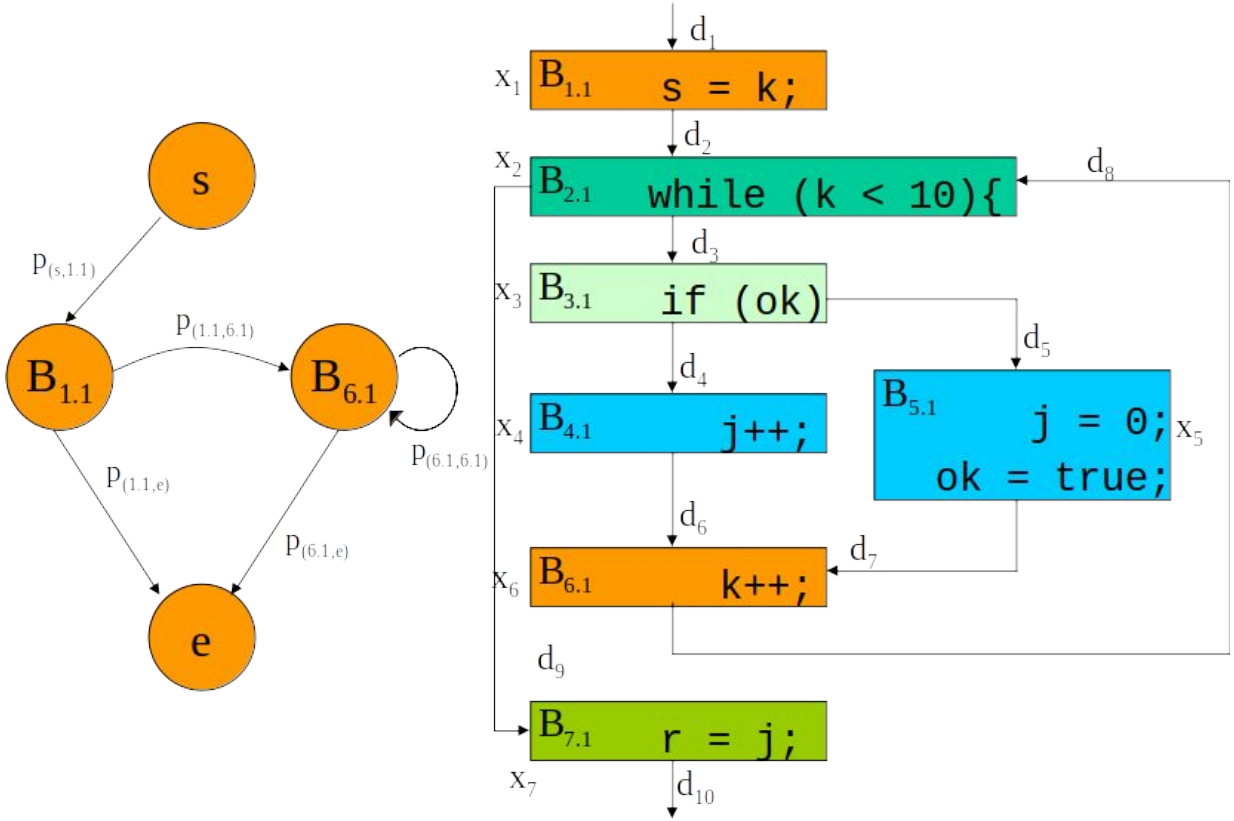




Example Cache Conflict Graph



Example Cache Conflict Graph



Cache Conflict Graph

- For each edge from node $B_{i,j}$ to node $B_{u,v}$ we assign a variable *p(i,j,u,v)* to represent the number of times the control passes through that edge
- These “p-variables” are similar to the “d-variables” associated with the edges of the control flow graph in program path analysis
- Sum of control flow going into a now is similarly equal to the sum of the sum of control flow going out of a node

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

- Sum of flow entering any l-block $B_{i,j}$ must be equal to the total execution count of that l-block, which is equal to the execution time of that basic block B_i

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

Cache Conflict Graph

- Similarly, if two I-blocks do not conflict, then an edge from one node to the other will also represent a cache hit
- If both edges ($B_{i,j}$,e) and (s, $B_{i,j}$) exist, then the contents of $B_{i,j}$ might already be in the cache at the beginning of the program's execution as its contents may be left from the previous program execution. Hence pariable $p(s,i,j)$ may be counted as a cache hit
- Hence

$$\sum_{u,v} p_{(u,v,i,j)} \leq x_{i,j}^{hit} \leq p_{(s,i,j)} + \sum_{u,v} p_{(u,v,i,j)}$$

Where $B_{u,v}$ does not conflict with Bi,j

These constraints, along with our objective function is our new ILP problem
Solution to this ILP problem returns the WCET of the program

Tools and Further Extensions

- A number of WCET analysis tools are available
  - Chronos is a freely available tool based on the SimpleScalar (MIPS) processor model
    Chronos
    SimpleScalar
  - A German company called Absint offer WCET tools and various processor models
- We have only discussed how direct mapped caches are modeled. More advanced techniques are required for set associative caches, pipelines and speculative execution, such as branch prediction