Chapter 3-2: Programs

Soo-Ik Chae

Topics

- Program performance analysis.

Varieties of performance metrics

- Worst-case execution time (WCET):
  - Factor in meeting deadlines.
  - Schedulability
- Average-case execution time:
  - Load balancing, etc.
  - To find hot spot
- Best-case execution time (BCET):
  - Factor in meeting deadlines.

Performance analysis techniques

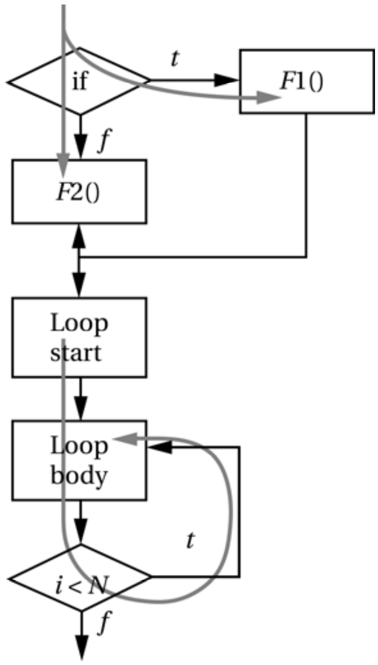
- Simulation.
  - Not exhaustive.
  - Cycle-accurate CPU models are often available.
- WCET analysis.
  - Formal method; may make use of some simulation techniques.
  - Bounds execution time but hides some detail.

WCET analysis approach

- Path analysis + path timing

- Path analysis determines worst-case execution path.
- Path timing determines the execution time of a path.

- The two problems interact somewhat.



Performance models

- Simple model --- table of instructions and execution times.
  - Ignores instruction interactions, data-dependent effects.
- Timing accident: a reason why an instruction takes longer than normal to execute.
- Timing penalty: amount of execution time increase from a timing accident.

SW estimation overview: approaches

- Two aspects to be considered
  - The structure of the code (program path analysis)
    - E.g. loops and false paths
  - The system on which the software will run (micro-architecture modeling)
    - CPU (ISA, interrupts, etc.), HW (cache, etc.), OS, Compiler
- Needs to be done at high/system level
  - Low-level
    - e.g. gate-level, assembly-language level
    - Easy and accurate, but long design iteration time
  - High/system-level
    - Reduces the exploration time of the design space

System-level software model

- Must be fast - whole system simulation
- Processor model must be cheap
  - “what if” my processor did X
  - future processors not yet developed
  - evaluation of processor not currently used
- Must be convenient to use
  - no need to compile with cross-compilers
  - debug on my desktop
- Must be accurate enough for the purpose

Accuracy vs Performance vs Cost

	Accuracy	Speed	$$$*
Hardware Emulation	+++	+ -	---
Cycle accurate model	++	--	--
Cycle counting ISS	++	+	-
Dynamic estimation	+	++	++
Static spreadsheet	-	+++	+++

*$$$ = NRE + per model + per design

Program path analysis

- Basic blocks
  - A basic block is a program segment which is only entered at the first statement and only left at the last statement.
  - Example: function calls
  - The WCET (or BCET) of a basic block is determined
- A program is divided into basic blocks
  - Program structure is represented on a directed program flow graph with basic blocks as nodes.
  - A longest / shortest path analysis on the program flow identify WCET / BCET

Program path analysis

- Program path analysis
  - Determine extreme case execution paths.
  - Avoid exhaustive search of program paths.

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

2^{100} possible worst case paths!

  - Eliminate *False Paths*:
    - Make use of path information provided by the user.

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

if (i)
  j++;
else
  j = j*j;
```

Always executed together!

Program path analysis

- Transform the problem into an integer linear programming (ILP) problem.
  - Basic idea: $\max\left(\sum_i c_i x_i\right)$

Single exec. time of basic block B_i (constant)

Exec. count of B_i (integer variable)

subject to a set of linear constraints that bound all feasible values of x_i's.

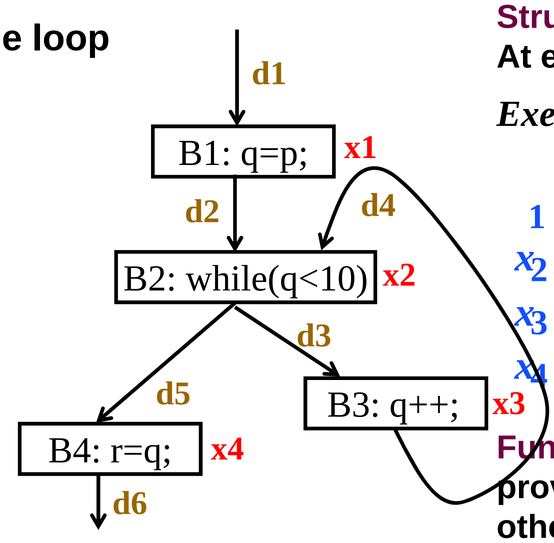
Assumption for now: simple micro-architecture model (constant instruction execution time)

Program path analysis: structural constraints

- Linear constraints constructed automatically from program's control flow graph.

Example: While loop

```
/* p >= 0 */
q = p;
while (q<10)
  q++;
r = q;
```



Source Code **Control Flow Graph**

Structural Constraints

At each node:

Exec. count of $B_i = \Sigma$ *inputs*

$= \Sigma$ *outputs*

1 1 2

$x_2 = d_2 + d_4 = d_3 + d_5$

$x_3 = d_3 = d_4$

$x_4 = d_5 = d_6$

Functional Constraints:

provide loop bounds and other path information

$0x_1 \le x_3 \le 10x_1$

Program path analysis: functional constraints

- Provide loop bounds (mandatory).
- Supply additional path information (optional).

Nested loop:

```
x1   for (i=0; i<10; ++i)
x2     for (j=0; j<i; ++j)
x3       A[i] += B[i][j];
```

$x_2 = 10x_1$

$0x_2 \le x_3 \le 9x_2$

loop bounds

$x_3 = 45x_1$

path info.

If statements:

```
x1   if (ok)
x2     i=i*i+1;
     else
x3     i=0;

x4   if (i)
x5     j=0;
     else
x6     j=j*j;
```

True statement executed at most 50%:

$x_2 \le 0.5x_1$

B_2 **and** B_5 **have same execution counts:**

$x_2 = x_5$

Path timing

- Includes processor modeling:
  - Pipeline state.
  - Cache state.
- Also includes loop iteration bounding.
  - Loops with conditionals, data-dependent bounds create problems.

Li/Malik ILP results

Program	Constraint sets	Estimated bound		Calculated bound		Pessimism	
		Lower	Upper	Lower	Upper	Lower	Upper
check_data	$4 \Rightarrow 2$	35	1,193	35	1,193	0.00	0.00
circle	1	431	15,958	431	15,726	0.00	0.01
des	2	73,912	672,298	75,033	667,127	0.01	0.01
dhry	$8 \Rightarrow 3$	314,266	1,326,475	314,266	1,326,475	0.00	0.00
djpeg	1	12,703,432	122,838,368	12,925,769	98,696,050	0.02	0.24
fdct	1	5,587	16,693	5,587	16,693	0.00	0.00
fft	1	1,589,026	3,974,624	1,593,122	3,974,601	0.00	0.00
line	1	380	9,148	380	9,148	0.00	0.00
matcnt	1	1,722,105	8,172,149	1,722,105	8,172,149	0.00	0.00
piksrt	1	236	5,862	236	5,862	0.00	0.00
sort	1	13,965	50,244,928	13,965	50,244,928	0.00	0.00
stats	1	1,007,815	2,951,746	1,007,815	2,951,746	0.00	0.00
whetstone	1	5,634,926	14,871,610	5,634,926	14,871,610	0.00	0.00

WCET bound vs. calculated bound

Program	Estimated bound		Measured bound		Pessimism	
	Lower	Upper	Lower	Upper	Lower	Upper
check_data	35	1,193	35	430	0.00	1.77
circle	431	15,958	585	14,483	0.26	0.10
des	73,912	672,298	111,468	243,676	0.34	1.76
dhry	314,266	1,326,475	575,492	575,622	0.45	1.30
djpeg	12,703,432	122,838,368	14,975,268	35,636,948	0.15	2.45
fdct	5,587	16,693	7,616	9,048	0.27	0.84
fft	1,589,026	3,974,624	1,719,813	2,204,472	0.08	0.80
line	380	9,148	929	4,836	0.59	0.89
matcnt	1,722,105	8,172,149	2,202,276	2,202,698	0.22	2.71
piksrt	236	5,862	337	1,705	0.30	2.44
sort	13,965	50,244,928	16,492	9,991,172	0.15	4.03
stats	1,007,815	2,951,746	1,158,142	1,158,469	0.13	1.55
whetstone	5,634,926	14,871,610	6,935,612	6,935,668	0.19	1.14

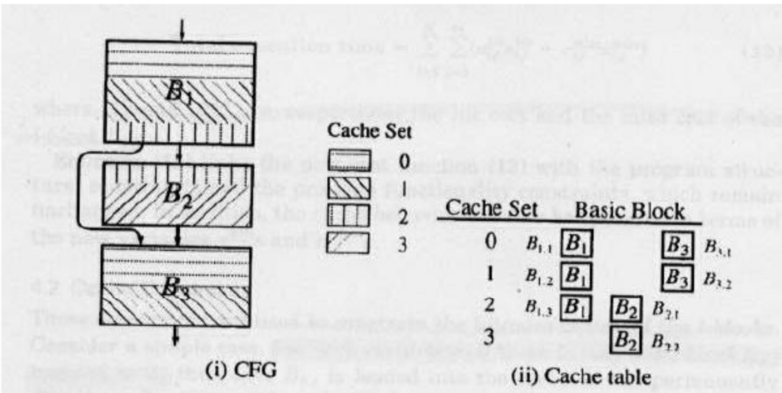
WCET bound vs. measured bound

Cache behavior and timing

- Cache affects instruction fetch time.
  - Time depends on state of the cache.
- Li and Malik break the program into units of cache lines.
  - Each basic block constitutes one or more I-blocks that correspond to cache lines.
  - Each I-block has hit, miss execution times.
  - Cache conflict graph models states of the cache lines.

Direct mapped Instruction caches

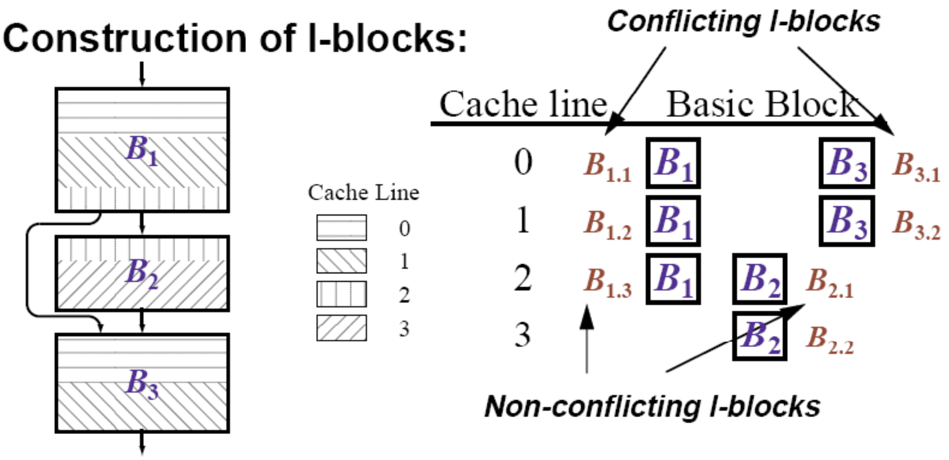
- the code in basic blocks are divided into a number of line-blocks
- the line blocks are assigned to cache lines
  – cache sets (cache lines) represent physical cache memory



(i) CFG
(ii) Cache table

Grouping Instructions: Line-blocks

- Line-block (l-block) = Basic block ∩ Cache line
  - » All instructions within a l-block have same cache hit/ miss counts.
- Construction of l-blocks:



Adding cache analysis to the ILP model

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

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

i = all basic blocks

j = all line blocks in block i

- the execution count of a basic block becomes

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

j = 1,2... n_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

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

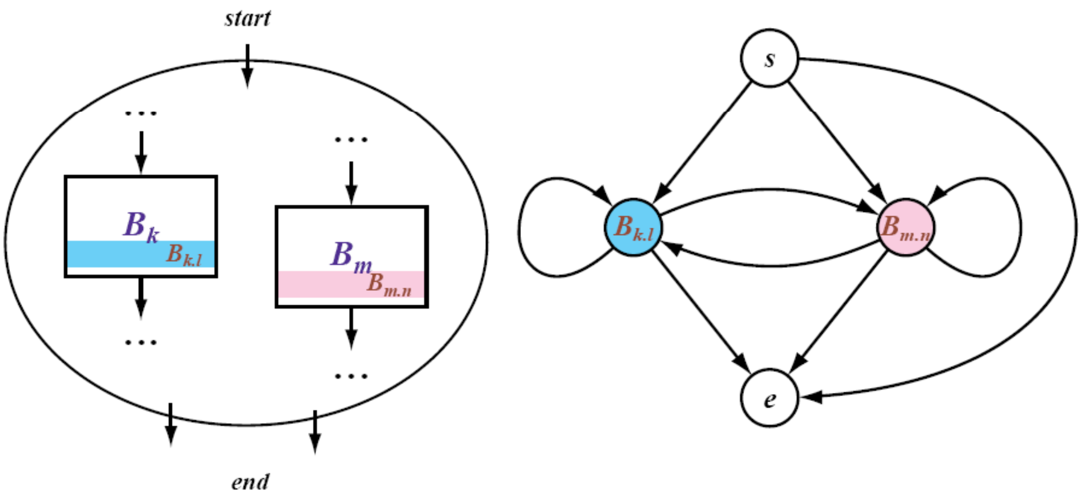
  - Two or more *nonconflicting* line-blocks are assigned to the same cache line
    - when a miss occurs in either block, the line-blocks will be loaded and no more cache misses will occur

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

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

Cache Conflict Graph

Capture control flow of l-blocks mapping to the same cache line only.

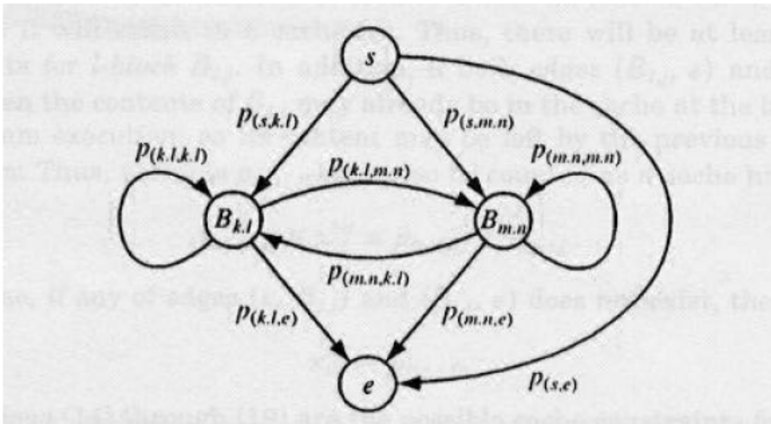


Control Flow Graph (CFG)

Cache Conflict Graph (CCG)

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

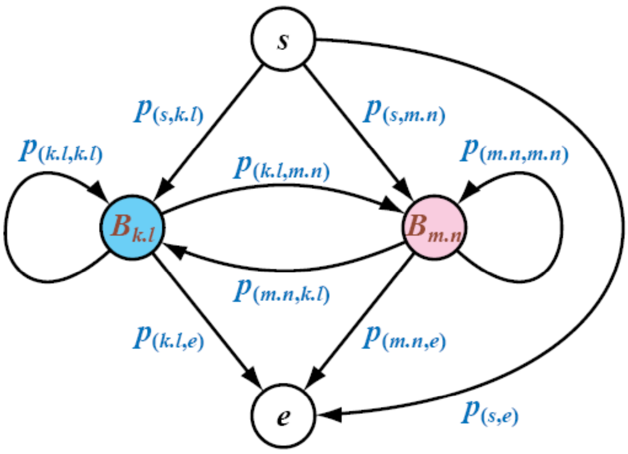


Constraints on cache conflict graphs

- 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

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

Constraints from CCG



Flow at node $B_{k.l}$:

$$x_k = p_{(s,k.l)} + p_{(m.n,k.l)} + p_{(k.l,k.l)}$$
$$= p_{(k.l,e)} + p_{(k.l,m.n)} + p_{(k.l,k.l)}$$

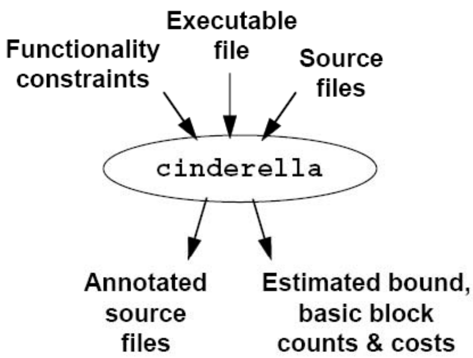
Cache hit count for *l-block* $B_{k.l}$:

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

Starting Condition:

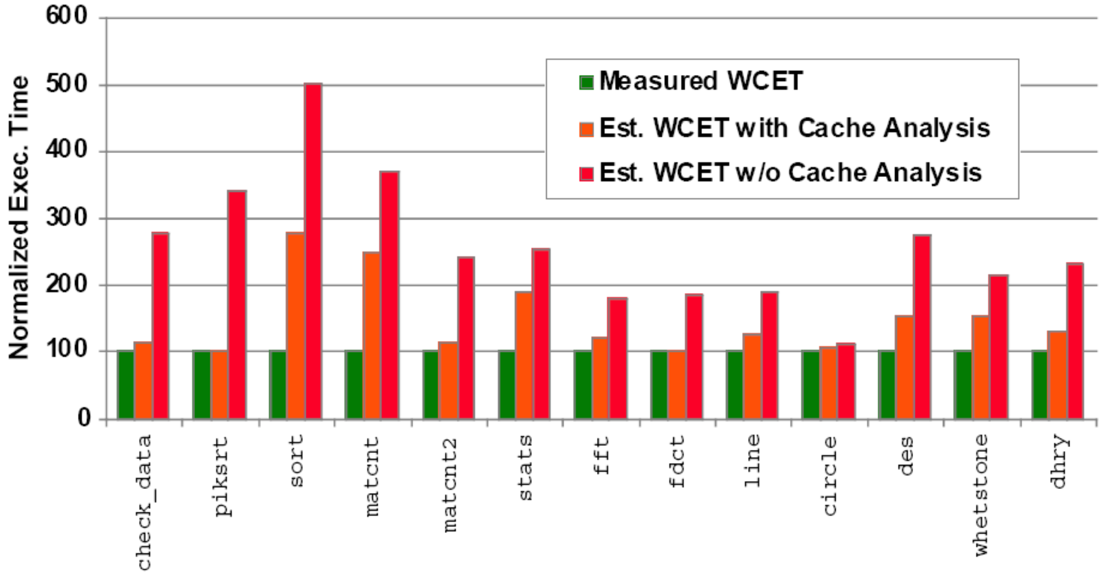
$$p_{(s,k.l)} + p_{(s,m.n)} + p_{(s,e)} = 1$$

Implementation – `cinderella'



- Target processor: Intel i960KB
- ~15,000 lines C++ code
- WWW page: `http://www.princeton.edu/~yauli/cinderella`

Experimental Results



Path Timing

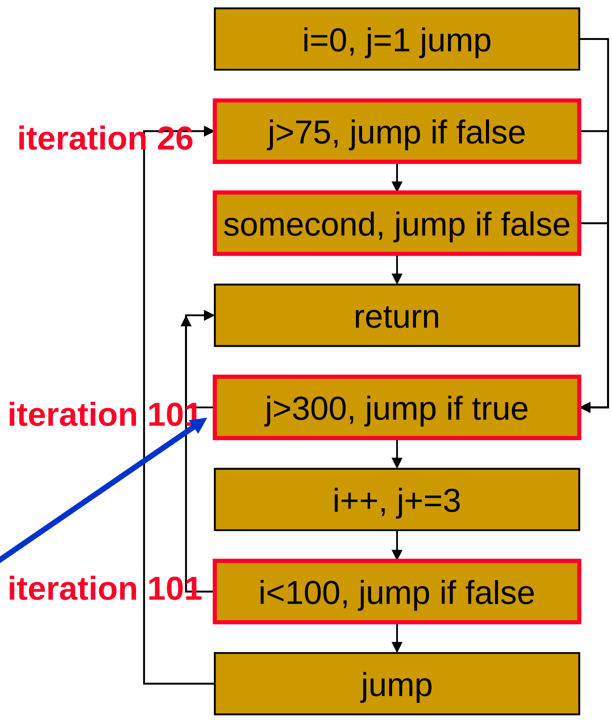
- Several techniques are used to analyze path timing at different levels of abstraction.
- Abstract interpretation: to understand the execution states of the program
- Data flow analysis: a more detailed view of how the program behaves
- Simulation: the most concrete technique

Healy et al. loop iteration bounding

- Use an iterative algorithm to identify branches that affect loop termination.
- Identifies the loop iteration on which those branches change direction.
- Determine whether these branches are reached.
- Calculate iteration bounds.

Loop iteration example (Healy et al)

```
for (i=0, j=1;
        i<100;
        i++, j+=3) {
    if (j>75 and somecond ||
        j>300)
        break;
}
```



Lower bound: 26
Upper bound: 101

Redundant code

Thieling et al. abstract interpretation

- Executes a program using symbolic values.
  - Allows behavior to be generalized.
  - Concrete state is full state; abstract state is one-to-many onto the concrete state.
- Cache behavior may be analyzed using abstract state.
  - Must analysis looks at upper bounds of memory block ages.
  - May analysis looks at lower bounds.
  - Persistence analysis looks at behavior after first access.