Real-Time and Embedded Systems

Lab 7

Alex Hoffman
alex.hoffman@tum.de

TU München
Institute for Real-Time Computer Systems (RCS)

2020

Lehrstuhl für Realzeit-Computersysteme

WCET with Caches

Problem:
Due to caches, execution time of individual basic blocks can vary at different points in the program execution.

WCET with Caches

Problem:
Due to caches, execution time of individual basic blocks can vary at different points in the program execution.

What can we do?

WCET with Caches – What can we do?

WCET with Caches – What can we do?

1. Just ignore them
   - no: This will lead to huge over- or underestimations up to factor of 100 on modern processors.

WCET with Caches – What can we do?

1. Just ignore them
   - no: This will lead to huge over- or underestimations up to factor of 100 on modern processors.
2. Find longest path with ILP technique, and then simulate it to get cache misses/hits

WCET with Caches – What can we do?

1. Just ignore them
   - no: This will lead to huge over- or underestimations up to factor of 100 on modern processors.
2. Find longest path with ILP technique, and then simulate it to get cache misses/hits
   - no: Longest path does not necessarily have most cache misses
   - e.g., second-longest path could have more cache misses and in fact take longer

WCET with Caches – What can we do?

1. Just ignore them
   - no: This will lead to huge over- or underestimations up to factor of 100 on modern processors.
2. Find longest path with ILP technique, and then simulate it to get cache misses/hits
   - no: Longest path does not necessarily have most cache misses
   - e.g., second-longest path could have more cache misses and in fact take longer
3. Find all possible paths, and then simulate them to get cache misses/hits

WCET with Caches – What can we do?

1. Just ignore them
   - no: This will lead to huge over- or underestimations up to factor of 100 on modern processors.
2. Find longest path with ILP technique, and then simulate it to get cache misses/hits
   - no: Longest path does not necessarily have most cache misses
   - e.g., second-longest path could have more cache misses and in fact take longer
3. Find all possible paths, and then simulate them to get cache misses/hits
   - no: computationally infeasible, too many paths (see lecture 6)

WCET with Caches – What can we do?

1. Just ignore them
   - no: This will lead to huge over- or underestimations up to factor of 100 on modern processors.
2. Find longest path with ILP technique, and then simulate it to get cache misses/hits
   - no: Longest path does not necessarily have most cache misses
   - e.g., second-longest path could have more cache misses and in fact take longer
3. Find all possible paths, and then simulate them to get cache misses/hits
   - no: computationally infeasible, too many paths (see lecture 6)
4. Extend ILP: provide constraints for min/max number of hits/misses

WCET with Caches – What can we do?

1. Just ignore them
   - no: This will lead to huge over- or underestimations up to factor of 100 on modern processors.
2. Find longest path with ILP technique, and then simulate it to get cache misses/hits
   - no: Longest path does not necessarily have most cache misses
   - e.g., second-longest path could have more cache misses and in fact take longer
3. Find all possible paths, and then simulate them to get cache misses/hits
   - no: computationally infeasible, too many paths (see lecture 6)
4. Extend ILP: provide constraints for min/max number of hits/misses
   - **advantage:** tight estimate

WCET with Caches – What can we do?

Use two variables per instruction (hit/miss count for each instruction: x_i^{hit}, x_i^{miss}) $\rightarrow$ might get computationally infeasible

- New ILP Problem:

$$\text{WCET} := \max_{x^{hit}, x^{miss}} \sum_{i=1...N} c_i^{hit} x_i^{hit} + c_i^{miss} x_i^{miss} \tag{1}$$

- very inefficient
- practically, this only works for small programs, not in industry

WCET with Caches – L-Blocks

Solution:

WCET with Caches – L-Blocks

Solution:

Group instructions to blocks again

WCET with Caches – L-Blocks

Solution:

Group instructions to blocks again - but how?

WCET with Caches – L-Blocks

Solution:
Group instructions to blocks again - but how?

Observation:
Only two kinds of instructions can have a cache miss:

- instructions that map to the beginning of a cache line
- instructions that are the first one in a BB

All other instructions will have a cache hit.

WCET with Caches – L-Blocks

Solution:

Group instructions to blocks again - but how?

Observation:

Only two kinds of instructions can have a cache miss:

- instructions that map to the beginning of a cache line
- instructions that are the first one in a BB

All other instructions will have a cache hit.

- If we group all instructions that are in the same BB and same cache line (called *line-blocks* or *l-blocks*), each group has only two possible execution times:
  1. first instruction = miss & all subsequent = hit
  2. first instruction = hit & all subsequent = hit

WCET with Caches – L-Blocks

Alternative definition:
l-blocks = max. sequence of code within a BB, s.t. when 1st instruction of it is accessed, either entire *l*-block is in cache or none

WCET with Caches – L-Blocks

Alternative definition:
l-blocks = max. sequence of code within a BB, s.t. when 1st instruction of it is accessed, either entire *l*-block is in cache or none
re-formulate the ILP-problem with *l*-blocks

$$\text{WCET} := \sum_{i=1}^{N} \sum_{j=1}^{n_i} c_{i,j}^{hit} x_{i,j}^{hit} + c_{i,j}^{miss} x_{i,j}^{miss}, \quad x_i = x_{i,j}^{hit} + x_{i,j}^{miss} \qquad (2)$$

Now, i sums over BBs again, and j over the *l*-blocks. More efficient than *per instruction*.

WCET with Caches – L-Blocks

Alternative definition:
l-blocks = max. sequence of code within a BB, s.t. when 1st instruction of it is accessed, either entire *l*-block is in cache or none
re-formulate the ILP-problem with *l*-blocks

$$\text{WCET} := \sum_{i=1}^{N} \sum_{j=1}^{n_i} c_{i,j}^{hit} x_{i,j}^{hit} + c_{i,j}^{miss} x_{i,j}^{miss}, \quad x_i = x_{i,j}^{hit} + x_{i,j}^{miss} \qquad (2)$$

Now, i sums over BBs again, and j over the *l*-blocks. More efficient than *per instruction*.

But: How do we know if an access will be a cache hit or miss?

WCET with Caches – Cache Conflicts

so far: we only looked at caches, but not how they depend on control flow

- **remember:** cache is small. Often the program does not fit in there.
- this means, sometimes data in the cache has to be replaced ("eviceted")
- evictions may result in cache misses
- the *control flow* (CFG) defines the evictions
- we need to link the CFG to the objective function seen before, i.e., we need further constraints for our ILP (What if we don't provide them?)

WCET with Caches – Cache Conflicts

The *Cach-Conflict-Graph* connects control flow with the cache effects

- **example:** see lecture
- essentially we construct a smaller CFG again, with nodes and edges
- resulting equations = more constraints for our ILP

Finally: Extended ILP objective function + structural constraints from CFG + more constraints from CCG = our new, cache-aware WCET analysis