Timing Analysis for Data Caches and Set-Associative Caches

by

Randall T. White, Christopher A. Healy, David B. Whalley,
Florida State University
Department of Computer Science

Frank Mueller,
Humboldt-Universität zu Berlin
Institut für Informatik

and

Marion G. Harmon
Florida A&M University
Department of Computer & Information Systems

Goal

To obtain tight worst-case execution times (WCETs) for real-time applications on systems that use a data cache or a set-associative instruction cache.

- Automatic process of WCET prediction
- *Static* analysis only
- Work with fully optimized code
- Analysis on entire program control flow
- Detect and exploit spatial and temporal locality
- Tighter prediction of WCETs

Related Work

Research into predicting WCETs for programs has intensified, focusing recently on those using direct-mapped instruction caches and pipelines.

Data caches;

- Min et. al. (Timing Schema)
  - Cannot deal with optimizations or function calls
  - No detection of spatial locality
  - No results with data larger than cache size
- Li et. al. (ILP)
  - Constraints entered by hand
  - Scalability problems
  - No results given for data caches
  - Can incur large overhead

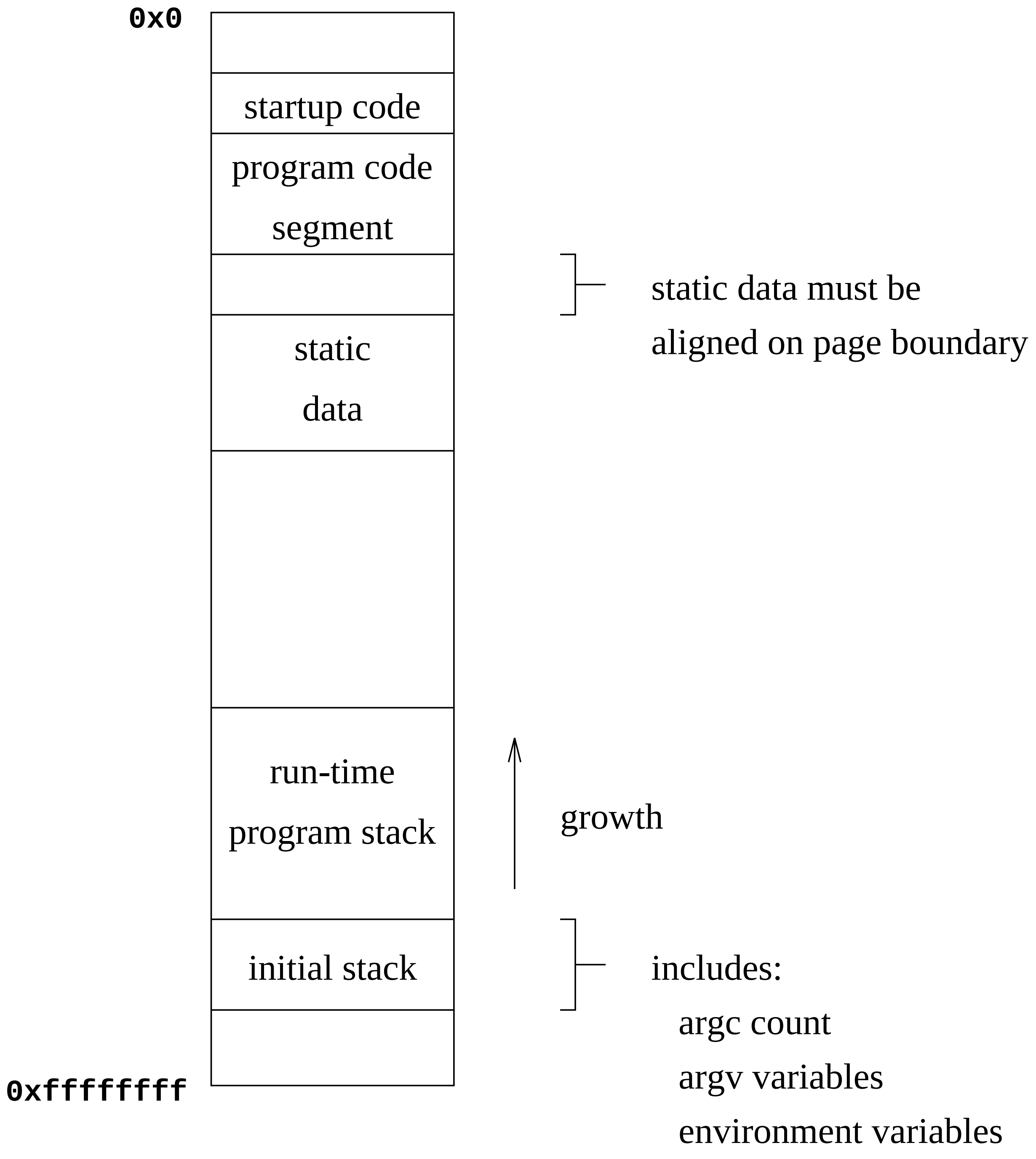
Set-associative instruction caches:

- Extension of Park's timing schema briefly mentioned but no formalization, implementation, or results reported.
- ILP approach extension
  - Only estimates cache misses
  - Higher overhead

Approach

- An optimizing compiler was modified to emit data information (bounded range of addresses), control-flow information, and the calling structure of functions in addition to regular object code generation.
- Virtual address ranges are calculated from the relative address ranges by examining the order of the assembly data declarations and the call graph of the entire program.
- The control flow of the program is analyzed to statically categorize the caching behavior of each data reference.
- These categorizations are used when calculating the pipeline performance of sequences of instructions representing paths within the program.
- The pipeline path analysis is used to estimate the worst-case execution performance of each loop and function in the program.

Virtual Address Space Organization



Calculation of Virtual Addresses

- Find Global Starting Address
- Find Stack Starting Address
- Compute Virtual Addresses of Global Scalars
- Compute Virtual Addresses of Local Scalars
- Compute Initial, Minimum, and Maximum Virtual Addresses for Calculateds
- Resolve Induction Variable Information to Give Access Pattern for Calculateds

Static Cache Simulation

Used to statically categorize the caching behavior of each data reference in a program for a specified cache configuration.

Two phases:

1. Iterative flow analysis to compute cache states
   - Modification of cache state representation
   - Additional cache state – *maybe* state (aka Calculated Cache State)
2. Categorization phase
   - Additional category for calculated data references

Algorithm to Calculate Data Cache States

```
WHILE any change DO
  FOR each basic block instance B DO
    IF B == top THEN
      input_state(B) = calc_input_state(B) = all invalid lines
    ELSE
      input_state(B) = calc_input_state(B) = NULL
    FOR each immed pred P of B DO
      input_state(B) += output_state(P)
      calc_input_state(B) += output_state(P)
                        + calc_output_state(P)
      IF P is in another loop THEN
        input_state(B) += calc_output_state(P)
                        + data_lines(remaining in that loop)
    output_state(B) = input_state(B)
    FOR each data reference D in B DO
      IF D is scalar reference THEN
        output_state(B) += data_line(D)
        output_state(B) -= data_lines(D conflicts with)
        calc_output_state(B) += data_line(D)
        calc_output_state(B) -= data_lines(conflicts with)
      ELSE
        output_state(B) -= data_lines(D could conflict with)
        calc_output_state(B) += data_lines(D could access)
        calc_output_state(B) -= data_lines(D could conflict with)
```

Data Reference Categories

- **Always Miss (`m`)**: The reference is not guaranteed to be in cache.
- **Always Hit (`h`)**: The reference is guaranteed to always be in cache.
- **First Miss (`f`)**: The reference is not guaranteed to be in cache the first time it is accessed each time the loop is entered, but is guaranteed thereafter.
- **First Hit (`i`)**: The reference is guaranteed to be in cache the first time it is accessed each time the loop is entered, but is not guaranteed thereafter.
- **Calculated (`c` <num> . . .)**: Indicates the maximum number of data cache misses that could occur at each loop level associated with the data reference.

Temporal Locality: recently accessed items are likely to be accessed in the near future.

```
int i, j, sum, same, a[50], b[50];
...
sum = 0;
for (i = 0; i < 50; i++)
   sum += a[i];                 /* ref 1 */

same = 0;
for (i = 0; i < 50; i++)
   for (j = 0; j < 50; j++)
      if (a[i] ==               /* ref 2 */
          b[j])                 /* ref 3 */
         same++;
```

Categorizations:

ref 1: c 13	from [m h m h h h ... m h h h]
ref 2: h	from [h h ... h h] due to temporal locality *across* loops
ref 3: c 13 13	from [m h h m h h h ... m h] on first execution of inner loop and [h h h h h h h ... h h] on all successive executions of it due to temporal locality *within* loops

Worst-Case Loop Analysis Algorithm

```
total_cycles = 0.
pipeline_information = NULL.
first_misses_encountered = NULL.
first_hits_encountered = NULL.
curr_iter = 0.
WHILE curr_iter != $n$ - 1 DO
    Find the longest continue path.
    first_misses_encountered += first misses that were misses in this path.
    first_hits_encountered += first hits that were hits in this path.
    IF a first miss or first hit was encountered in this path THEN
        curr_iter += 1.
        Subtract 1 from the remaining misses of each calculated reference in this path.
        Concatenate pipeline_information with the union of the information
            for all paths.
        total_cycles += additional cycles required by union.
    ELSE IF a calculated reference was encountered in this path as a miss THEN
        min_misses = the minimum of the number of remaining misses of each
                          calculated reference in this path that is nonzero.
        min_misses = min(min_misses, $n$ - 1 - curr_iter).
        curr_iter += min_misses.
        Subtract min_misses from the remaining misses of each calculated reference
            in this path.
        Concatenate pipeline_information with the union of the information
            for all paths min_misses times.
        total_cycles += (additional cycles required by union) * min_misses.
    ELSE
        break
Concatenate pipeline_information with the union of the pipeline information
    for all paths ($n$ - 1 - curr_iter) times.
total_cycles += (additional cycles required by union) * ($n$ - 1 - curr_iter).
FOR each set of exit paths that have a transition to a unique exit block DO
    Find the longest exit path in the set.
    first_misses_encountered += first misses that were misses in this path.
    first_hits_encountered += first hits that were hits in this path.
    Concatenate pipeline_information with the union of the information
        for all exit paths in the set.
    total_cycles += additional cycles required by exit union.
    Store this information with the exit block for the loop.
```

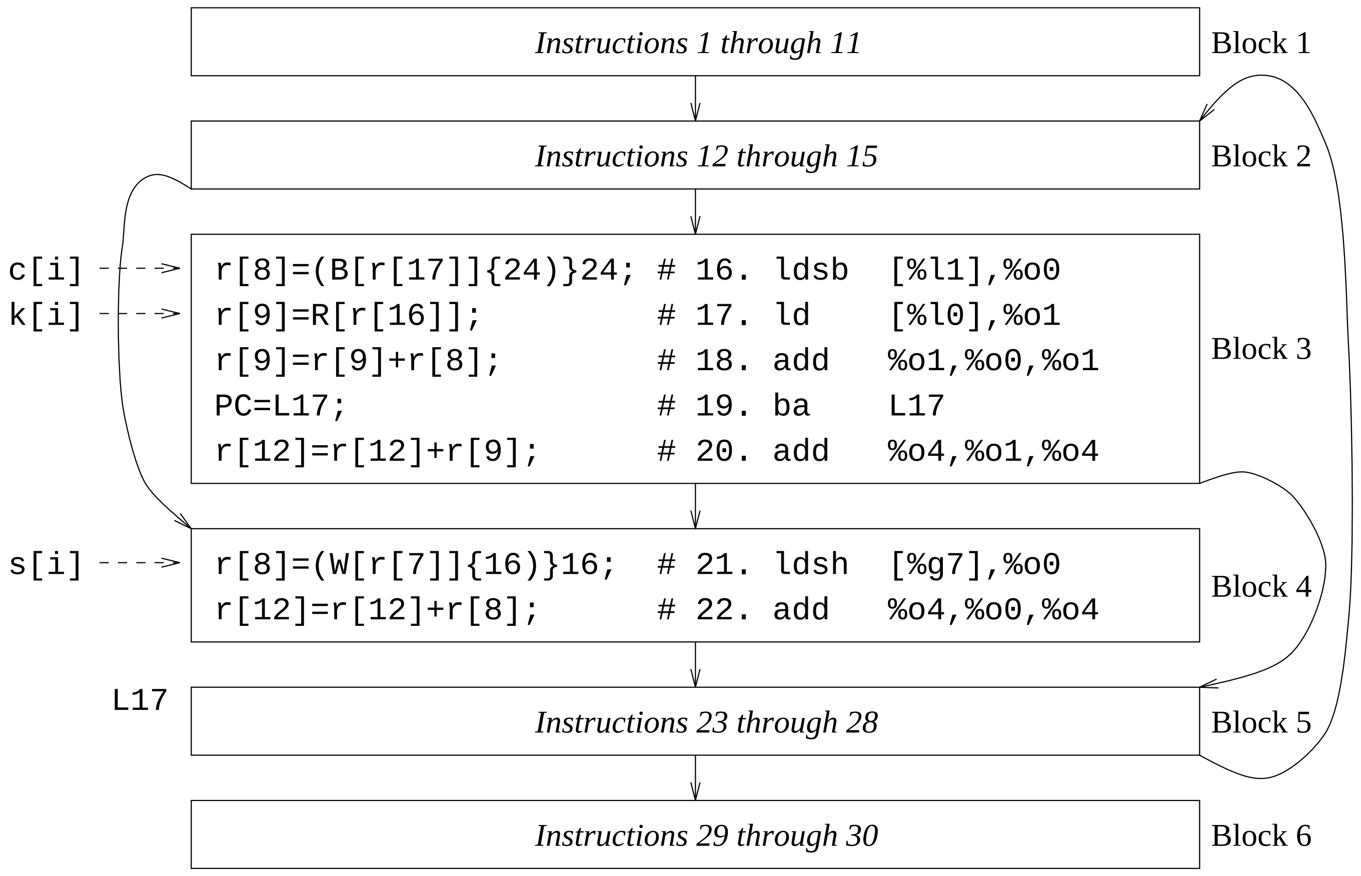
C Source for WCLA Algorithm Example

```
int k[100];
short s[100];
char c[100];

main()
{
   int i, sum;

   sum = 0;
   for (i=0; i<100; i++)
      if ((i & 3) != 1)
         sum += k[i]+c[i];
      else
         sum += s[i];
}
```

RTLs and SPARC Assembly for WCLA Algorithm Example


Instructions 1 through 11
Block 1
Instructions 12 through 15
Block 2
c[i]
k[i]
Block 3
s[i]
Block 4
L17
Instructions 23 through 28
Block 5
Instructions 29 through 30
Block 6

```
r[8]=(B[r[17]]{24)}24; # 16. ldsb  [%l1],%o0
r[9]=R[r[16]];         # 17. ld    [%l0],%o1
r[9]=r[9]+r[8];        # 18. add   %o1,%o0,%o1
PC=L17;                # 19. ba    L17
r[12]=r[12]+r[9];      # 20. add   %o4,%o1,%o4
```

```
r[8]=(W[r[7]]{16)}16;  # 21. ldsh  [%g7],%o0
r[12]=r[12]+r[8];      # 22. add   %o4,%o0,%o4
```

Paths in the loop:

- Path A: Blocks 2, 3, & 5
- Path B: Blocks 2, 4, & 5

Pipeline Diagrams for Paths A and B

Pipeline Diagram for Path A: Instructions 12-20 and 23-28 (blocks 2,3,5)

		cycle																			
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
stage	IF	12	13	14	15	16	17	18	19	19	20	23	24	25	26	27	28				
	ID		12	13	14	15	16	17	18	18	19	20	23	24	25	26	27	28			
	EX			12	13		15	16	17		18		20	23	24	25	26		28		
	MEM				12	13		15	16	17		18		20	23	24	25	26		28	
	WB					12	13		15	16	17		18		20	23	24	25	26		28

20 cycles

Pipeline Diagram for Path B: Instructions 12-15 and 21-28 (blocks 2,4,5)

		cycle																
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
stage	IF	12	13	14	15	21	22	23	23	24	25	26	27	28				
	ID		12	13	14	15	21	22	22	23	24	25	26	27	28			
	EX			12	13		15	21		22	23	24	25	26		28		
	MEM				12	13		15	21		22	23	24	25	26		28	
	WB					12	13		15	21		22	23	24	25	26		28

17 cycles

Hit Ratio Number of hits divided by number of memory accesses (loads and stores). Obtained from execution simulator.

Observed Cycles Obtained from execution simulation of data cache and pipeline effects.

Estimated Cycles Obtained from timing analyzer.

Estimated Ratio Quotient of Estimated divided by Observed. Shows how well the timing analyzer performed.

Naive Ratio Obtained by running timing analysis assuming all data cache references were misses and dividing those cycles by the observed cycles. Shows advantage of doing data cache analysis.

Average prediction improvement of 30% for programs in the test suite.

Overhead

- Primarily from static cache simulation
- Average of 2.89 seconds for the test suite.
  Max: 10.3 (*Matcnta*) Min: 0.2 (*Matsumb*)
- Average timing analysis time 1.05 seconds

Static Instruction Cache Simulation

- **Extended to set-associative caches**
- Same assumptions as for data caches
- Addresses of instruction inferred from control-flow graph
- Call graph ⇒ function instance tree
- Data-flow analysis ⇒ cache states
- Categorize cache references

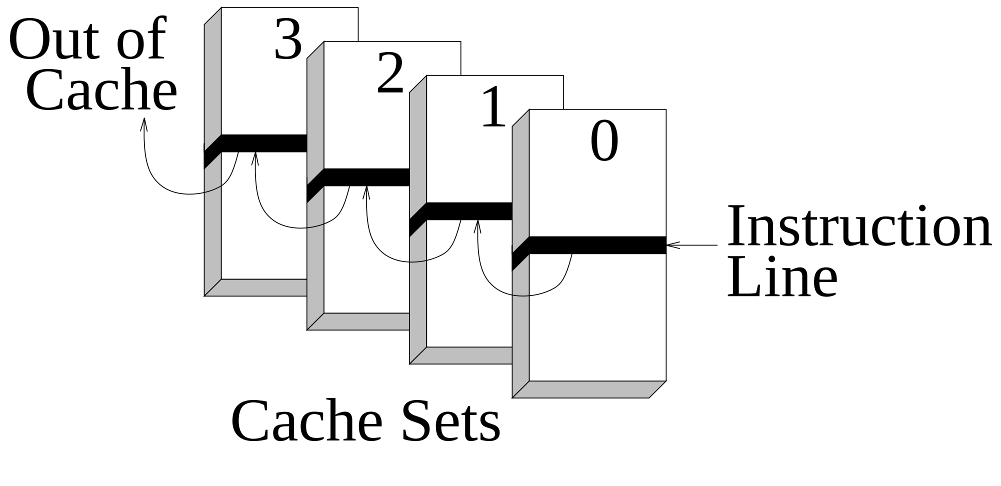
Associativity	Processors
1	most SPARC, MIPS and Alpha chips
2	Intel Pentium, AMD K6, Alpha 21264, PowerPC 602/603, MIPS R5000/R10000
4	AMD K5, Motorolla 68040/68060, PowerPC 604, Cyrix x86, SPARC R1 (HaL)
8	PowerPC 601/620

Data-flow Analysis for Instruction Caches

- Abstract cache state (ACS) := instructions that may be cached
- Linear cache state:= instructions that may be cached without loops
- Post dominator set:= instructions that must still be executed
- Input state(B) := ACS before block B
- Output state := input state + instructions in block - conflicts
- **Aging of lines through associativity levels (LRU)**
- Worst-case categorization of instructions
- Conservative analysis, efficient

Aging for Set-Associative Caches

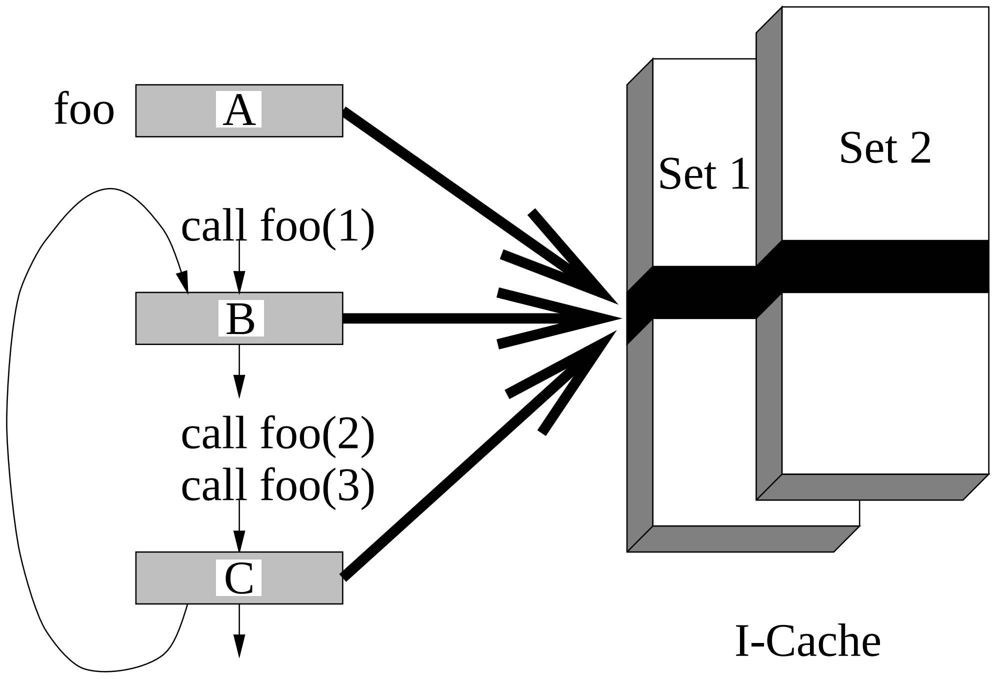
- For each associativity level
  - input(B) $\cup=$ output(predecessors)
- For each instruction in B
  - add inst to level 0 (youngest age)
  - shift conflicting instruction to next higher level
- Instruction "may be cached" if in ACS of any associativity level
- Formalized in paper



Instruction Categorization

- Automatically determined from data-flow analysis
- Categories for each loop nesting level:
  - always-hit: always in cache
  - always-miss: never in cache
  - first-hit: in cache on 1st reference, not in cache otherwise
  - first-miss: not in cache on 1st reference, in cache otherwise
- Timing analysis uses categorization

Example: Categorization

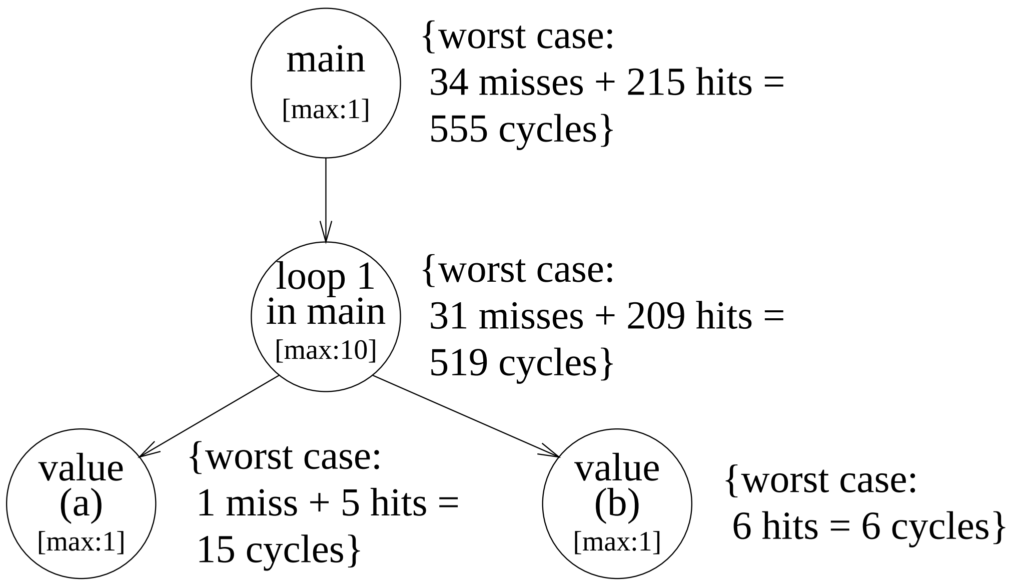


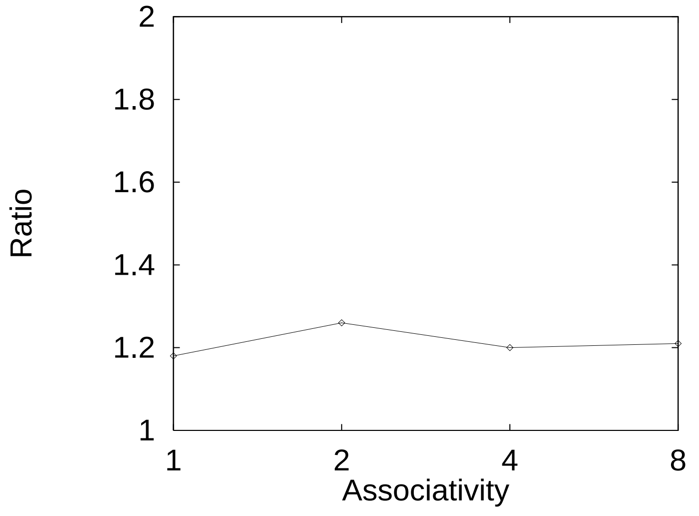
- A foo(1): always-miss
- A foo(2): first-hit
- A foo(3): always-hit
- B, C: always-misses

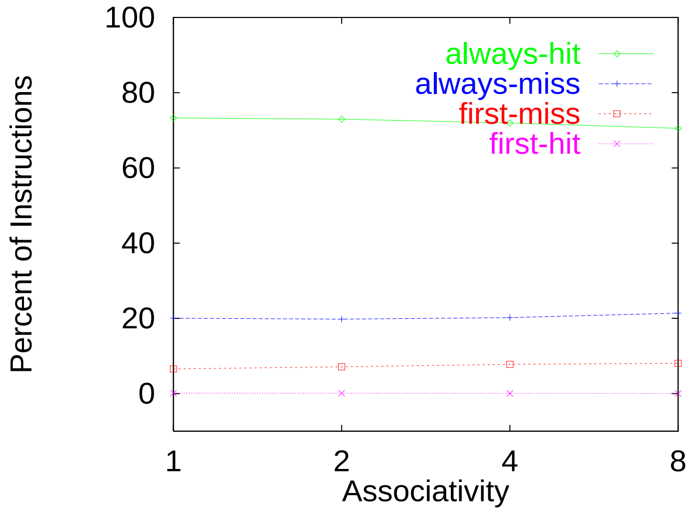
Timing Analysis

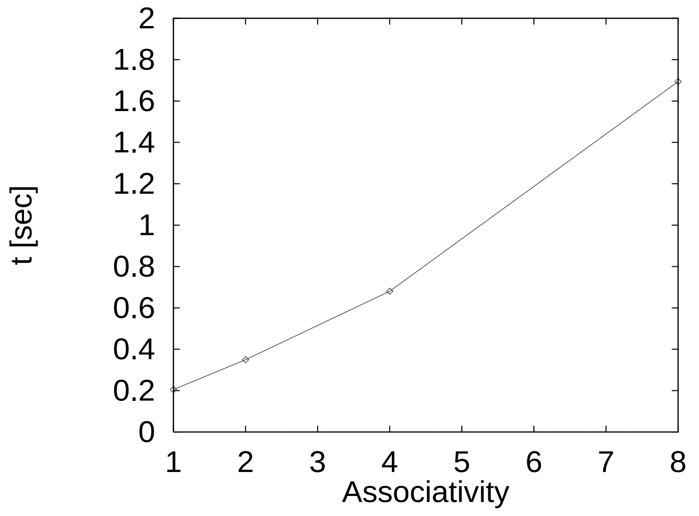
- constructs timing tree
- determines cycles bottom-up (for each node)
  - considers instruction categories
  - simulates pipeline
- path traversal for loops (fix-point algorithm)
- adjustments of categories between loop levels
- ⇒ conservative estimations

Timing Tree




Ratio
2
1.8
1.6
1.4
1.2
1
1
2
4
8
Associativity


Percent of Instructions
100
80
60
40
20
0
always-hit
always-miss
first-miss
first-hit
1
2
4
8
Associativity


t [sec]
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
5
6
7
8
Associativity

Conclusions

- Formal method to predict cache behavior
- Extended to data caches and set-associative instruction caches
- Integrates with timing analysis
- Yields tight WCET predictions
- Scales well with large data sizes and higher levels of associativity
- Provides verifyable WCETs for schedulability analysis
- Allows higher utilization of real-time applications
- Enables use of cached architectures for hard RT

Future Work

- Extending Set-Associative Analysis to Data Caches
- Merging Instruction and Data Caching Prediction and Simulation
- Wrap-Around Fill for Data Caches
- Write Buffer
- Best Case

Annulled Branches

If an annulled branch is not taken, then the instruction in the delay slot will be *annulled*. This means that although it will occupy all stages in the pipeline, the results of the instruction will not be committed. If this instruction is a load or a store, it will be flushed out of the pipeline before a read from or write to memory is performed, respectively.

Example:

```
                 ⋮
         add       %o2,%o0,%o2      # 9
         cmp       %o2,%g1          # 10
         ble,a     L15              # 11
         ld        [%o2],%o0        # 12
         sethi     %hi(_a),%o3      # 13
                 ⋮
L15:     add       %o1,%o5,%o1      # 27
                 ⋮
```

If branch taken: 9, 10, 11, 12, and 27 will execute

If branch *not* taken: 9, 10, 11, and 13 will execute

Cache Configuration Assumptions

- **Direct-mapped** cache
  - Each data line maps to only one cache line
- Write policy is **Write-Through**
  - Results always written to memory
- Write miss policy is **No-Write Allocate**
  - If cache miss on write, cache is *not* updated

Consequences:

- Cache writes (stores) will always cause a pipeline delay in the MEM stage
- Cache writes do not have any effect on the cache state.