Determining the Worst-Case Instruction Cache Miss-Ratio

Filip Sebek and Jan Gustafsson
Dept. of Computer Engineering, Mälardalen University
Västerås, Sweden
{fsk,jgn}@mdh.se

Abstract

A high cache miss-ratio of a program will lead to longer execution time and a higher power consumption. By knowing the cache miss-ratio, performance can be estimated in advance and can be used as input for compilers and system developers. This paper presents a method to bound the worst-case instruction cache miss-ratio of a program. The method is static, needs no manual annotations in the code and is safe in the meaning that no under-estimation is possible.

1 Motivation

The cache miss-ratio is an important property of a program. The worst-case cache miss-ratio (WCCMR) is useful for instance in the following areas:

- Power-aware systems. The energy consumption grows with higher cache miss-ratio since cache misses leading to more bus and main memory activity [1].
- Multiprocessor systems on a shared bus. A high cache miss-ratio will congest the bus with a performance loss in the average case.
- Real-time systems. Cache memories are in many cases avoided in real-time systems due to their highly complex behavior. Even if most applications have less than 10% miss-ratio in the instruction cache, there is still a chance that an application can miss more during a certain section of a program. If these bursts of misses occur during a critical phase of the execution, the program might miss its deadline with possibly a malfunction as a result.

One should observe that the execution path with the highest miss-ratio must not be equal to the one with the longest execution path in time or number of instructions executed. A simple example is a polling task that in the average case only makes conditional jumps over a very complicated algorithm that is executed only once a fortnight.

2 Related and adjacent work

Much research has been performed to calculate the worst-case execution time (WCET) that is an important property of a task or process in a real-time system where the timing constraints must not be exceeded to have a correct function. Adding caches to a real-time system is a non-trivial task since the execution time will become variable depending if the executing instruction or accessed data is in the cache or not. Some methods have nevertheless successfully been able to make system with caches analyzable [2, 3, 4, 5, 6, 7], but they all aim at WCET-analysis. To the best of our knowledge no one has tried to bound WCCMR, even if many of the WCET-analysis algorithms with some modifications would be able to perform such analysis.

The major difference between the adjacent work and our proposition is the simplicity of our approach.

3 The concept and approach

When the CPU fetches a new instruction or data, and it is not in the cache memory, a *miss* occurs and reload from the main memory must be performed. To reduce the penalty, not only the missing instruction is fetched but a complete *line*[1] of instructions is transferred since the main memory can burst consecutive memory locations to the cache. This line might be 2, 4, 8 or 16 words large. The *spatial locality* is exploited which refers to the fact that nearby memory locations are more likely to be accessed than more distant locations.

If, for instance, the line size is 4 words and the program is one sequence of consecutive instructions without branches ("a single basic block"), a miss will occur every 4:th instruction. This situation leads to a miss-ratio of $\frac{1}{4} = 25\%$ and if the line size is 8 words, the miss-ratio will be $\frac{1}{8} = 12.5\%$

If only a part of the instructions in the line will be executed and a jump to another cache line occurs, this line will have a higher miss-ratio. If, for instance, the second instruction in a 8-word-line is a branch, this particular cache line will have a

[1] A cache *line* is sometimes also called *block*, but since the term might be mixed up with for instance *basic block*, this paper will use the term "line".

miss-ratio of $\frac{1}{2} = 50\%$. The worst-case scenario is when the first instruction of a cache line is a jump; in this case the miss-ratio is 100%. To suffer from 100% cache misses of a complete program means that it only performs jumps to uncached lines and such program will never be implemented since it cannot do anything useful. The myth of the always cache missing program in a real-time system should hereby be unveiled[8].

The bottom line of this argumentation is that many jumps and early jumps in a cache line leads to a higher miss-ratio. By analyzing the jumps in a code, the cache miss-ratio can be estimated.

3.1 Limitations

The proposed method will not exploit temporal locality and will therefore assume that all new cache line accesses are misses and will only take advantage of the spatial locality. The estimation of the miss-ratio will never be lower than $\frac{1}{linesize}$.

Set-associativity, non-blocking caches, data caches and prefetching are not handled. The analyzed code must always terminate so endless loops are not permitted. A non-terminating process (common in real-time systems) can however be analyzed by removing the “big loop”.

4 The algorithm

This section presents the algorithm to calculate the worst-case cache miss ratio in detail.

4.1 Overview

1. Construct a Control Flow Graph (CFG) of the program at machine code level.
2. Identify all conditional and unconditional branches and determine their position in the cache line.
3. Identify all jump target addresses and determine their position in the cache line.
4. Calculate all instructions’ “local miss-ratio”.
5. Determine the maximum and minimum number of iterations in loops (performed at intermediate code level).
6. Construct a binary tree of possible execution paths that might lead to a worst-case cache miss-ratio.
7. Traverse the tree and find the execution path with the highest cache miss-ratio.

4.2 A Control Flow Graph

A control flow graph (CFG) [9] describes the possible execution paths through a program. The nodes in a CFG represent *basic blocks*[2], and the edges represent the flow of control.

4.3 The “local miss-ratio”

Each instruction is associated to a miss-ratio that is equal to the inversion of the distance between the incoming and the outgoing arrow relatively to the instruction. We will in this paper call this miss-ratio of each instruction for the *“local miss-ratio”*. See the examples in Figure 1. The (a) figure illustrates a cache line without any branches and target addresses. The (b) example with a conditional jump in the second word of the cache line assess the first two words as $\frac{1}{2}$ since worst-case is the use of only two out of four words in that cache line. If no jump is performed all instructions should have the local miss-ratio $\frac{1}{4}$, but since both scenarios are possible, only the worst-case miss-ratio is assigned to a word. In (c) the third word is a target address and since there is a possibility that only two words are being used in this cache line, the two last words are assigned $\frac{1}{2}$ as local miss-ratio.

A way to bound the WCCMR tighter is to distinguish short forward and backward jumps within a cache line. If for instance Figure 1(f) would be such a case the miss-ratios would be $\{\frac{1}{3}, \frac{1}{3}, \frac{1}{4}, \frac{1}{3}\}$ and for a short backward loop, as in 1(d), all words in the line would be assessed $\frac{1}{4}$.

4.4 Loops

If the loop has a higher miss-ratio than the rest of the program it is important to know how large this part of the execution is relative to the rest of the program. By using the *maximum* possible number of iterations in the loop will render a safe estimation if the miss-ratio in the loop is higher than in the rest of the program. If the miss-ratio in the loop is lower than in the rest of the program, the *minimum* possible number of iterations must be used. It is non-trivial to determine which of those that should be used until the complete program is analyzed, and that is why to the best of our knowledge no major pruning or substitution method is possible.

It is therefore necessary to know the number of iterations for the loops in the analyzed program. To determine the number of iterations is trivial for a `for`-loop with a simple counter, assuming the the counter is not changed in the loop body. For loops with more general termination conditions, the number of iterations is not that obvious.

A common method to handle such cases is to add manual annotations to the code (see for example[10]). There are however

[2] A *basic block* is a linear sequence of instructions without halt or possibility of branching except at the end

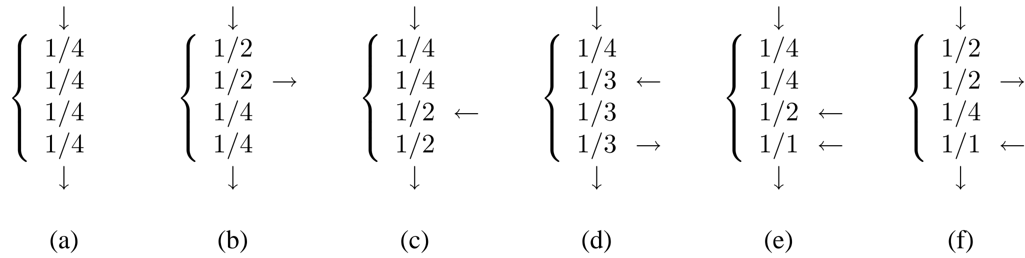

Figure 1: Some examples of *"local cache miss-ratio"* on 4-word cache lines. Conditional and unconditional jumps are symbolized as (→) and jump targets as (←).

automated methods that can determine the maximum number of iterations without manual annotations (see for example [11, 12]). Such methods could be used prior to our analysis.

4.5 The possible-execution-paths tree

All basic blocks in the CFG associate to a tuple $\langle mr, weight, min, max\rangle$ where *mr* is *miss-ratio*, *weight* the number of instructions in the basic block, *min* the minimum number of execution times and *max*, the maximum number of execution times of the basic block. Non-iterating sequences of code will have *min* and *max* assigned to '1'.

A binary tree of possible execution scenarios is generated. An if-statement generates two possible branches and loops generate also two branches: the minimum-iteration-path and the maximum-iteration-path.

Some optimizations can be performed to reduce the size of the tree.

- A loop with a fixed number of iterations can be simplified to a basic block following the previous basic block. Such a loop (for instance a `for`-statement) is identified when the minimum number of iterations of a basic block is equal to the maximum.
- If two paths have identical tuple-descriptions, those can be reduced to one path and the branch in the tree can be omitted.
- If it can be proved that the rest of the program (at a certain stage) will not be able to reach a higher WCCMR than another path, the search in that branch can be aborted and the execution path may be omitted in the tree. Such a proof is possible to make if the number of executable instructions are so few that the already calculated part of the program will outweigh the rest even if the remaining program will have a miss-ratio of 100%. This information can be available through a generation of an execution path tree that is built reversal from the end and built upon the number of instructions executed in each basic block. This "weight tree" must not be complete, but the closer it is reaching the start of the program the more optimizations can be performed at the possible-execution-path-tree. The "weight tree" can easily be optimized since only the heaviest node of all duplicated nodes are necessary — all other nodes and paths to light weight nodes can be omitted.

These optimizations can be performed directly after the loop analysis, before the tree generation, to simplify the tree generation process.

4.6 The overall miss-ratio

The last step is to compute the WCCMR in all possible execution paths to find its' maximum value. Every node is only once traversed but observe that the complete path must be traversed to calculate a correct value of WCCMR.

4.7 Algorithm performance

The performance consuming parts of the algorithm are focused on two parts. The first is the construction of the CFG and assign all assembly instructions with a local miss-ratio, which can be performed with the efficiency of O(n) where n is the number of assembly instructions in the program.

The second part is the traversing of the CFG to build and analyze the tree, which is built with combinations of basic blocks in all execution paths. Since each node in the CFG can occur multiple times in the tree, there is a possibility of an exponential growth of the tree O(2^n), which seems to be the algorithms bottleneck. Even a small program but with a complex structure of loops and branches can take very long time to analyze. The situation is however not as bad as it might first appear since only a fraction of all execution paths are analyzed; In loops only the minimum and maximum number of iterations are considered, not the complete interval, and several branch situations can be omitted by optimization.

5 Example

The code in Figure 2 is transformed to the CFG in Figure 3.

A straight-line edge in the figure represents a basic block and a curved edge represents a conditional or unconditional jump in the code. The `do-while` loop is analyzed to iterate anything between 4-15 times and the `while` loop will iterate 1 or 2 times.

```
...
...
if(a>b) {
  ...
  ...
  do{
    ...
  }while(c>d);
}
else {
  ...
  ...
  while(e<3){    .
    ...
  }
}
...
```

Figure 2: A code written in C that will be analyzed as an illustration of the method.

In this example we have chosen the cache line to be 8 words as in for instance in Motorola PowerPC 750. A CFG is constructed and from this each instruction is assigned a local cache miss-ratio (Figure 3). The program is analyzed to determine each loop's maximum and minimum number of iterations.

The binary tree in Figure 4 describes all possible execution paths to terminate the program. The `if-else`-statement leads to two branches (true/false) and the loops will also generate two branches each (maximum and minimum number of iterations). Each edge in the tree symbolizes a basic block and it's associated cache miss-ratio weight.

Table 1 shows the calculated cache miss-ratio for the different execution paths. The execution time calculation should just give a hint about what is going on; we let each miss gives ten "time units" penalty.

As shown in Table 1, execution path (e) will yield the highest cache miss-ratio. One can notify that path (c) has a shorter execution time than (f), but a higher miss-ratio, and the conclusion from this is that only a combination of many instructions and a high miss-ratio will render a long execution time. An other view is that the worst-case execution time path must not be the same as the worst-case cache miss-ratio path. Observe that the path (e) doesn't contain the basic blocks with the worst miss-ratio, and the reason is that those blocks weighted little because of the complete programs execution behavior.

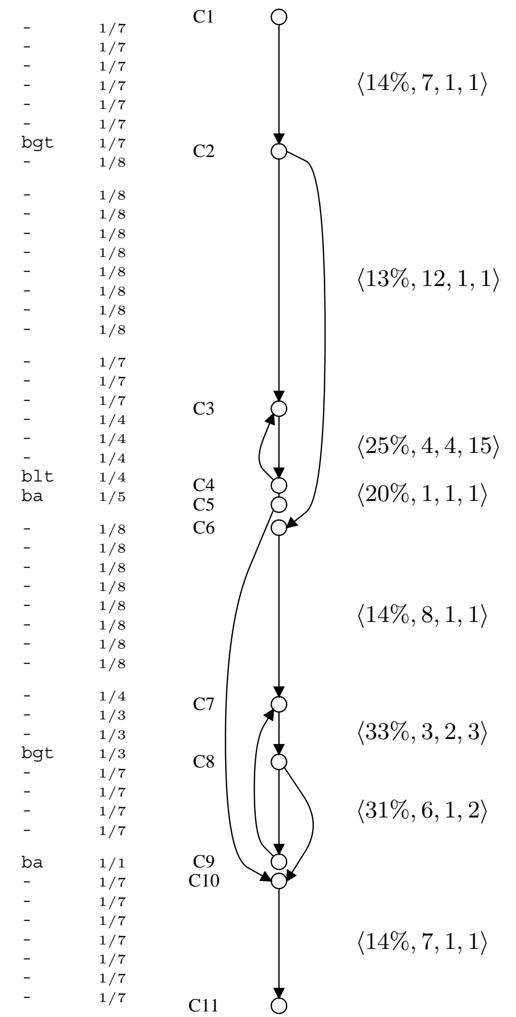

Figure 3: The core of the proposed method. The column to the very left is the *assembly language* of the C-code in Figure 2. The next column describes each assembly instructions *"local miss-ratio"* that is derived from the *control flow graph* (CFG) in the middle. The last column to the right describes each *basic block's miss-ratio and its' potential share weight* of the complete program.

6 Future work

A reduction method of the tree to keep it manageable and reduce the analysis time will be developed. It might however not be possible without approximations with a looser bound of WCCMR as a result.

The method will also be developed to handle temporal locality. This can be achieved by including a static cache simulator as for instance in [3]. With this extension a tighter bound of

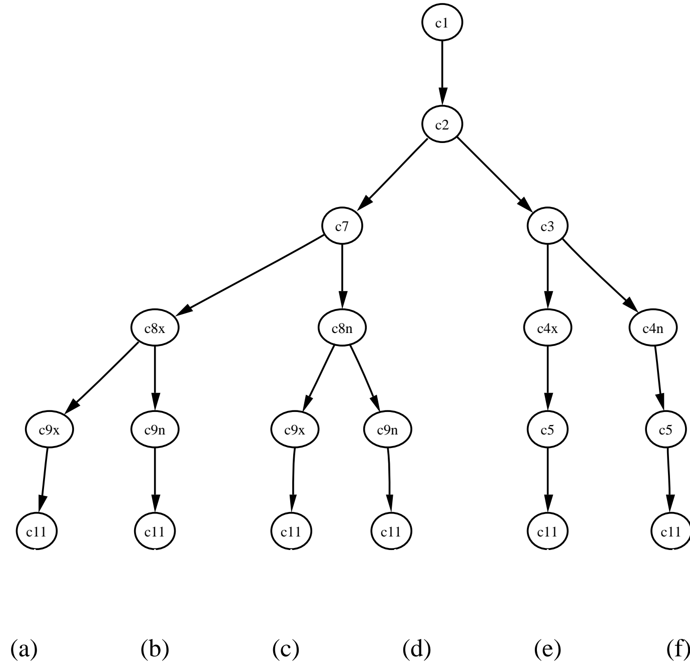

Figure 4: The binary search tree to find the WCCMR. The node labels corresponds to the node labels in Figure 3. The suffix 'n' and 'x' in some nodes indicates the path of the minimum and maximum number of iterations in a loop.

Path	Miss-ratio	Number of Instructions	Execution Time
(a)	$(14 \cdot 7 + 14 \cdot 8 + 33 \cdot 3 \cdot 3 + 31 \cdot 6 \cdot 2 + 14 \cdot 7)/43 = 20.6\%$	43	132
(b)	$(14 \cdot 7 + 14 \cdot 8 + 33 \cdot 3 \cdot 3 + 31 \cdot 6 \cdot 1 + 14 \cdot 7)/37 = 18.9\%$	37	107
(c)	$(14 \cdot 7 + 14 \cdot 8 + 33 \cdot 3 \cdot 2 + 31 \cdot 6 \cdot 2 + 14 \cdot 7)/40 = 19.7\%$	40	119
(d)	$(14 \cdot 7 + 14 \cdot 8 + 33 \cdot 3 \cdot 2 + 31 \cdot 6 \cdot 1 + 14 \cdot 7)/34 = 17.6\%$	34	94
(e)	$(14 \cdot 7 + 13 \cdot 12 + 25 \cdot 4 \cdot 15 + 20 \cdot 1 + 14 \cdot 7)/87 = \mathbf{21.5}\%$	87	275
(f)	$(14 \cdot 7 + 13 \cdot 12 + 25 \cdot 4 \cdot 4 + 20 \cdot 1 + 14 \cdot 7)/43 = 18.0\%$	43	121

Table 1: Calculated WCCMR for each of the execution paths of the binary tree in Figure 4.

the WCCMR can be accomplished, but to the price of a more performance intensive analysis.

7 Conclusion

The cache miss-ratio is an important property of a program that controls performance, execution time and power consumption among many other properties.

This paper proposes a simple analysis technique to find a worst-case cache miss-ratio execution path in a program. The cache miss-ratio can for instance directly be used to estimate the highest possible power consumption of a program, but also be used as an input for a compiler optimizition. The method is based on the fact that spatial locality is exploited when several instructions in a cache line are executed consecutively. The more instructions that executes without a jump, the lower cache miss-ratio. All instructions can hereby be assigned a "local miss-ratio" that can be used to compute over-all cache miss-ratio for different execution paths. To cope with the problem that parts of the program may be more used than others in for instance loops, a method based on abstract interpretation computes the minimum and maximum number of iterations. The method needs no manual annotations and can be fully automated.

This paper also demonstrates with an example that the worst-case execution time path must not be the same as the worst-case cache miss-ratio path.

The major drawback of the proposed method at this stage is the exponential growth of the search tree that demands high performance to solve complex program structures in a reasonable time. The method as presented is suitable for programs that is partitioned into many, small tasks[3]. Indirect pre-emption effects must not be concerned since temporal locality is not included in the method and will by this not suffer from cache-related pre-emption delay and hereby still yield safe values.

References

[1] Paolo D'Alberto, Alexandru Nicolau, Alexander Veidenbaum, and Rajesh Gupta. Static analysis of parameterized loop nests for energy efficient use of data caches. In *Proceedings of Workshop on Compilers and Operating Systems for Low Power (COLP)*, Barcelona, Spain, September 2001.

[2] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th Real Time System Symposium*, pages 298–307, December 1995.

[3] Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. Bounding worst-case instruction cache performance. In *Proceedings of the IEEE Real-Time Systems Symposium 1994*, pages 172–181, December 1994.

[4] Sung-Soo Lim, Sang Lyul Min, Minsuk Lee, Chang Park, Heonshik Shin, and Chong Sang Kim. An accurate instruction cache analysis technique for real-time systems. In *Proceedings of the Workshop on Architectures for Real-time Applications*, April 1994.

[5] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Journal of Real-Time Systems*, pages 183–207, November 1999. Special Issue on Timing Validation.

[6] Greger Ottosson and Mikael Sjödin. Worst-case execution time analysis for modern hardware architectures. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.

[7] Friedhelm Stappert and Peter Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. Technical Report 27/97, C-Lab, Paderborn, Germany, December 9th 1997.

[8] Filip Sebek. When does a disabled instruction cache outperform an enabled? In *Submitted to USENIX/WIESS 2002*, Boston, Massachusetts, USA, December 8th 2002.

[9] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers–Principles, Techniques, and Tools*. Addison - Wesley, 1986.

[10] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1(2):159–176, September 1989.

[11] Andreas Ermedahl and Jan Gustafsson. Deriving Annotations for Tight Calculation of Execution Time. In *EUROPAR97*, pages 1298–1307, August 1997.

[12] Jan Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. Doctorial thesis, Uppsala University and Mälardalen University, Västerås, Sweden, May 2000.

[3] also referred as *tasks* in real-time systems