Efficient Instruction Cache Analysis with Model Checking

Mingsong Lv\textsuperscript{1}, Nan Guan\textsuperscript{1}, Wang Yi\textsuperscript{2}, Qingxu Deng\textsuperscript{1}, Ge Yu\textsuperscript{1} \\
\textsuperscript{1}Northeastern University, Shenyang, China  
\textsuperscript{2}Uppsala University, Uppsala, Sweden

Abstract

Cache analysis is one of the major tasks of static timing analysis. There are very efficient techniques based on abstract interpretation for the analysis of the cache replacement policy LRU. It is known that these techniques can not deal with cache replacement policies such as PLRU and FIFO with satisfactory precision. On the other hand, model checking technique may provide precise analysis results for these policies, but it doesn’t scale due to the search space of the analysis problem. In this paper, we identify a class of programs for which are able to reduce the branching structure of the programs without changing their WCET. The reduction technique allows to significantly reduce the state space of a program and thus model checking technique can be used for efficient WCET analysis with adequate precision. As an example, we focus on the FIFO replacement policy, but the technique can be extended to handle the other policies such as PLRU.

1. Introduction

Caches are commonly employed in most embedded processors to bridge the performance gap between the CPU and main memory by exploiting locality in memory accesses. Since this gap is getting bigger, the cache performance has a large influence on both real-time and average performance of a system. Timing analysis must take into account cache behaviors to produce tight and useful WCETs. Currently, abstract interpretation (AI) based methods can yield precise results efficiently in the analysis of set-associative caches with the LRU replacement policy, but it cannot handle FIFO and PLRU replacement policies that are commonly employed in real-life processors [1]. Even thought there are AI based techniques for FIFO, the analysis precision is still a serious problem. Model checking may provide precise analysis results, but it doesn’t scale due to the state space explosion problem [4].

In this paper, we will use model checking to solve the WCET analysis problem for programs running on platforms with FIFO cache replacement policy. To remedy the scalability problem, we studied the static structure of some programs and their influence on the WCET. We found that even though the combination of the branching structures and loops of a program contributes significantly to the state space of a program, in many cases, most branches of a branching structure have no influence on the WCET of the program. So we identified a class of programs whose program branches may be removed to reduce the search space without changing their WCET, and presented a “branch cutting” technique that implements the above idea. The advantage of our analysis method is that model checking can be used to estimate the WCET of such programs more precisely and efficiently without suffering from the state space explosion problem.

The rest of this paper is organized as follows. Section 2 shows how to model a program with cache behaviors using timed automata. The branch cutting technique under the FIFO replacement policy is detailed in Section 3. Experiment results are presented in 4. Conclusion and future work are presented in Section 5.

2. Modeling the Execution of a Program with Cache Behaviors

A program is first compiled to the binary file, which is further decompiled to generate the Control Flow Graph (CFG). The CFG of a program can be formally defined as follows:

**Definition 1 (CFG).** A CFG is a tuple $(B, Sta, Ter, T)$, where $B$ is the set of all basic blocks of the program, with each basic block denoted by $b_i$, $Sta$ and $Ter$ are the fictitious start and end block of the program, where $Sta, Ter \in B$, $T \subseteq B \times B$ is the transition relations, with $t_{i,j}$ denoting the transition from $b_i$ to $b_j$.

The CFG is further transformed to a timed automaton (TA) [2] that precisely simulates the behavior of the program. The WCET is calculated by letting the model checker to run the automaton and verify if the specifications on execution time are satisfied or not. We basically extended our
previous work [3] by adding data structures for caches and semantics for updating caches in the program model.

![Figure 1. Transformation from CFG to TA](image)

Figure 1 illustrates an example CFG. The CFG contains a loop and a branching structure in the loop body. The range on each basic block are the addresses of the starting and ending instruction. The static structure of the TA is directly mapped from the CFG. To model cache behaviors, we define an array Cache to maintain cache contents. On any incoming transition of a location except Ter, a function update(x, y) is invoked to simulate how the execution of the basic block changes the cache contents. The function reads in the instructions of a basic block (with address range [x, y]), check whether the instruction is hit or miss, and update the Cache data structure accordingly. The function also calculates the execution time of the basic block and sets it to a variable cost. We use a clock c to model the execution time of the basic block. The invariant “c<=cost” on a location and the guard “c==cost” on its outgoing transitions are used to guarantee that the automaton stays exactly cost time in this location. Another clock gc is defined to record the total time elapsed, i.e., the execution time of the program. To model other replacement policies, one only needs to modify the implementation of update() by changing the rules of updating cache contents.

We can input the TA to the UPPAAL model checker and ask UPPAAL to verify the safety property A [] gc<=N, which specifies that “for all paths, the execution time will never exceed the given bound N”. Apparently, if and only if N is the actual WCET, A [] gc<=N is proved true and at the same time A [] gc<=N-1 is proved false. We can then perform a binary search within a safe range by invoking the model checker iteratively to find the actual WCET of the given program.

3 Branch Cutting under the FIFO Replacement Policy

Results from our previous work [3] show that the main cause of state space explosion is the number of program paths. For instance, if a loop body has an “if-then-else” structure in it and the loop executes 100 times, there are 2^100 possible program paths. When cache analysis is added, the problem will be even worse. If we can prove that in any case the execution time of one path will definitely not change the WCET of the program, then this path can be cut from the program model; and now the loop contains only one single path, and there are only 100 possible program paths to be checked. In this way, the analysis efficiency is greatly improved. This is the basic idea of “branch cutting”.

![Figure 2. The safety issue of branch cutting](image)

To do branch cutting, some critical issues should be considered. First, we must guarantee that cutting a branch from the original CFG will never result in lower WCET values. This is called the “safety issue” of branch cutting. We use a simple example to explain it. Figure 2 illustrates a loop body with a branching structure. The loop body has two paths: one path executes A and then C, the other executes B and C. Assume that the WCET of A and B are tA and tB, and tA < tB. If we consider the branching structure alone, it is apparent that branch A can be cut. But the execution of A and B may evict cache contents of C, thus resulting longer execution time of C. Let’s assume that the WCETs of C after the execution of A and B are tC_A and tC_B, respectively. If (tA + tC_A) > (tB + tC_B), then cutting branch A will lead to underestimation. So considering the branch structures alone in branch cutting can not guarantee safe results, and the effects of the branch structures to other parts of the program (and vice versa) should be carefully treated.
Another issue is that one must safely compute the lower and upper bounds of the execution time of each branch. Assume that the actual bounds of the execution time of branch A and B are \([20, 50]\) and \([60, 90]\). Since \(50 < 60\), branch A can be safely cut. But if an ineffective estimation method is applied, the computed bounds of the two branches may be \([10, 65]\) and \([55, 95]\). Now we are not able to cut branch A, since \(65 > 55\) (which means sometimes A executes longer than B). So if the precision of the estimation method is low, the possibility of branch cutting is greatly reduced.

To summarize, we must safely and precisely estimate the lower and upper bounds of each branch to carry out efficient branch cutting. In the following part of this section, we show how branch cutting can be applied to the loops.

For FIFO or PLRU replacement policies, if there are contents of the program remaining in the cache, the hit/miss of the instructions will be very unpredictable. So in our discussion, we assume empty cache when a program starts. This assumption is reasonable since most embedded processors assume that the actual bounds of the execution time of branch A and upper bounds of the execution time of each path. This is done by statically predicting the hit/miss information of each instruction. Since cache sets are independent, we just discuss the situation for one cache set. The basic idea to predict hits or misses is to count the total number of cache blocks (from all paths) mapped to a set, and compare it to cache associativity (denoted by \(\text{Asso}\)). Let’s assume that there are two paths \(p_1\) and \(p_2\), \(S_1(S_2)\) is used to represent the sets of cache blocks mapped from \(p_1(p_2)\) to the cache set. Let \(n_1 = |S_1 \setminus S_2|\), \(n_2 = |S_2 \setminus S_1|\), and \(n_3 = |S_1 \cup S_2|\). Then we have the following hit/miss classification rules:

- \(n_1 > \text{Asso}\) (\(n_2 > \text{Asso}\)), the cache blocks of \(S_1 \setminus S_2 (S_2 \setminus S_1)\) are classified as “always miss”
- \(n_3 \leq \text{Asso}\), all the cache blocks of \(S_1 \cup S_2\) classified as “first miss”
- otherwise, all the cache blocks are “not classified”

For each instruction, we can calculate its execution time bounds according to the hit/miss information. Let \(t_M\) and \(t_M\) to denote the cache hit time and cache miss penalty. For “always miss”, “first miss”, and “not classified”, the corresponding bounds are \([t_M, t_M]\), \([t_M, t_M]\), and \([t_M, t_M]\), respectively. Note that we assume each cache block contains exactly one instruction, so no “always hit” can be predicted in this paper. Adding up the bounds of each instruction on a path, we get the bound for the execution time of the path.

When the bounds of all the paths are calculated, the third step is to decide which path can be safely cut. Before giving the rules for branch cutting, we define the cache connected relationship between two paths.

**Definition 2** (cache connected). Given two paths \(p_1\) and \(p_2\), and set \(S_{i,j}(S_{i,2})\) that represents the sets of cache blocks mapped from \(p_1(p_2)\) to cache set \(i\). \(p_1\) and \(p_2\) are defined as cache connected if any of the conditions is satisfied:

- \(\exists i, |S_{i,1} \cup S_{i,2}| > \text{Asso}\)
- \(\exists p_i, \text{both } p_1 \text{ and } p_2 \text{ are cache connected to } p_i\)

If \(p_1\) and \(p_2\) are cache connected, the execution of \(p_1\) may evict the cache blocks of \(p_2\) (either directly or indirectly), possibly making the execution time of \(p_2\) longer, and further resulting in longer execution time of the loop. So the following two conditions must be satisfied to safely cut \(p_1\): (1) the upper bound of \(p_1\) is smaller than the lower bound of \(p_2\); (2) \(p_1\) and \(p_2\) are not cache connected.

### 3.2 Branch Cutting with Loop Unrolling

It is true that branch cutting cannot take effect on all branching structures. For example in Figure 4, the loop body contains path \(p_1\) with bound \([20, 40]\) and path \(p_2\) with bound \([30, 60]\). In this case, we cannot cut \(p_1\) since \(40 > 30\). But if the loop body can perfectly fit into the cache (“perfectly fit” means for any cache set that contains cache blocks

---

**Figure 3. Example of the flattening operation**

After flattening, the next step is to calculate the lower and upper bounds of the execution time of each branch.
from the loop body, the number of cache blocks mapped to this set is not larger than the cache associativity), then all the cache accesses of any path will be “always hit” once the path is loaded into the cache. Here the upper bounds correspond to the time to load a path, and the lower bounds correspond to the execution time when a path is loaded into the cache. For such loops, we can predict that the scenario that leads to the WCET of the loop is: use one iteration for loading each path into the cache, and then execute the path with longest execution time (given that cache accesses are all hits) for the rest iterations.

\[ \text{Loop bound} = N \]

\[ \text{Loop bound} = N - 1 \]

\[ p_2 \]

\[ p_1 \]

\[ \text{[20, 40]} \]

\[ \text{[30, 60]} \]

**Figure 4. Branch cutting with loop unrolling**

For such loops, we can first transform the loop body and then apply branch cutting. The transformation is illustrated in Figure 4. Since when cache accesses are all hits path \( p_2 \) executes longer, this path is kept in the loop body. We unroll the loop body with one iteration and put path \( p_1 \) in it. The unrolled iteration corresponds to the loading of \( p_1 \), and the first iteration of the new loop body corresponds to the loading of \( p_2 \). The transformed structure has exactly the same WCET with the original one, while the number of possible program paths is greatly reduced since the new loop body contains only one path.

### 3.3 Branch Cutting of Nested Loops

For nested loops, the biggest problem for branch cutting is that the inner loop may be entered multiple times with different initial cache states. In this case, the hit/miss of the cache accesses of the inner loop is difficult to predict. Currently we can only deal with a special situation where the nested loop can perfectly fit into the cache. Such nested loops have similar properties to the loop illustrated in Figure 4. Then we can perform branch cutting from the inner most loops to the outer most loop by applying a method similar to the one in Section 3.2. The major difference is that: only the first time an inner loop is entered, we need to consider loading each path. We distinguish different situations in our model.

In nested loops, an inner loop may locate on one path of the outer loop. When we are deciding whether to cut one path of the outer loop, we need to calculate the bounds of the execution time of the inner loop. The upper bound corresponds to the scenario when the inner loop is entered for the first time when all paths of the inner loop are loaded into the cache; the lower bound corresponds to the scenario when the inner loop has been loaded and all the cache accesses of the inner loop are hits. Then branch cutting for the outer loop can be conducted similar to a non-nested loop.

### 4 Preliminary Experiment Results

We selected 5 programs from the Mälardalen WCET benchmark and computed their WCETs. Experiment results are listed in Table 1, where time is measured in second and memory in MB. The results exhibit a notable improvement in the analysis efficiency in terms of time and memory usage. We believe that branch cutting is a promising and effective technique in reducing the search space of the WCET analysis problem.

**Table 1. Preliminary experiment results**

<table>
<thead>
<tr>
<th>Prog.</th>
<th>WCET Without BC</th>
<th>Time</th>
<th>Mem</th>
<th>With BC</th>
<th>Time</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>fir</td>
<td>14,492</td>
<td>1,132</td>
<td>99.4</td>
<td>1.05</td>
<td>3.5</td>
<td></td>
</tr>
<tr>
<td>janne</td>
<td>13,454</td>
<td>145,032</td>
<td>99.7</td>
<td>1.18</td>
<td>3.5</td>
<td></td>
</tr>
<tr>
<td>expint</td>
<td>4,328</td>
<td>16,478</td>
<td>279.1</td>
<td>2.55</td>
<td>55.1</td>
<td></td>
</tr>
<tr>
<td>crc</td>
<td>10,432</td>
<td>11,170</td>
<td>280.1</td>
<td>4.309</td>
<td>130.1</td>
<td></td>
</tr>
<tr>
<td>bsort</td>
<td>9,180</td>
<td>2,518</td>
<td>71.8</td>
<td>0.58</td>
<td>3.5</td>
<td></td>
</tr>
</tbody>
</table>

### 5 Conclusion and Future Work

In this paper, we present a cache analysis framework by model checking, and use branch cutting to reduce the state space of the problem. Our future work include: (1) more precise methods to predict cache hit/miss in calculation of execution time bounds; (2) methods to deal with nested loops that cannot fit into the cache; (3) extending branch cutting to other policies, such as PLRU.

### References


