ILP-Based Program Path Analysis for Bounding Worst-Case Inter-Task Cache Conflicts*

Hiroyuki TOMIYAMA††, Member and Nikil DUTT†††, Nonmember

SUMMARY  The unpredictable behavior of cache memory makes it difficult to statically analyze the worst-case performance of real-time systems. This problem is further exacerbated in the case of preemptive multitask systems because of inter-task cache interference, called Cache-Related Preemption Delay (CRPD). This paper proposes an approach to analyzing the tight upper bound on CRPD which a task might impose on lower-priority tasks. Our method finds the program execution path which requires the maximum number of cache blocks using an integer linear programming technique. Experimental results show that our approach provides up to 69% tighter bounds on CRPD than a conservative approach.

**key words:** cache-related preemption delay, real-time systems, embedded software, program path analysis

1. Introduction

Due to the continuously increasing speed gap between processors and memories, the impact of cache hits/misses on the overall system performance has also been increasing. Indeed, cache memory improves the average system performance and is employed in most mid- to high-performance computer systems. However, its inherently unpredictable behavior makes it difficult to statically estimate the tight bound on worst-case performance of the systems. Especially, worst-case performance analysis is extremely important for efficient implementation of hard real-time systems where deadline violation results in a critical failure. There are a number of previous research efforts, for example, [2], [4],[10],[11],[16], to estimate the tight bound on worst-case execution time of a given task in a single-task environment. However, they can not directly be applied to preemptive multitask systems because they do not take into account inter-task cache interference, called Cache-Related Preemption Delay (CRPD). CRPD is the time required to reload necessary data/code which was present in the cache but is displaced from the cache by the other tasks. CRPD can not be neglected in many real-time systems, particularly embedded controller systems consisting of rather small tasks which are frequently activated by sensors.

One approach to avoid CRPD is cache partitioning [6], [17]. In this approach, the cache is divided into several disjoint partitions, each of which is dedicated to a specific task. Thus, there exists no CRPD in case every task has its own cache partition. Although cache partitioning makes it easier to statically analyze the cache behavior in a preemptive multitask environment, it causes serious degradation of the cache performance (therefore, degradation of the overall system performance) due to the limited cache capacity available to each task.

Some recent studies incorporated CRPD into schedu-lability analysis of fixed-priority, periodic, preemptive real-time systems [3],[7],[8]. They focus on estimation of worst-case CRPD of instruction cache memory. In [3], CRPD which a task \( \tau \) might impose on lower-priority tasks is estimated by multiplying the number of cache blocks used by \( \tau \) by cache refill time. This estimation implicitly assumes that whole program code and data of \( \tau \) is loaded into the cache. Obviously, this is a pessimistic estimation because most programs involve conditional statements in which the execution path depends on input data, and therefore, not all of the program code may be executed. Lee et al.’s work [7], [8] also suffers from this kind of pessimism although their work is more sophisticated than [3].

This paper proposes a new approach to estimate a tight bound on worst-case CRPD which a given task might impose on lower-priority tasks. Our work determines the program execution path which uses the maximum number of cache blocks using an Integer Linear Programming (ILP) technique. By solving the ILP problem, a tight bound on worst-case CRPD is obtained.

This paper is organized as follows. Section 2 presents a couple of motivating examples and describes the path analysis problem addressed in this paper. Section 3 formulates the path analysis problem for instruction-CRPD as an ILP problem. Section 4 presents experiments that demonstrate the effectiveness of our approach. In Sect. 5, we conclude this paper with a summary and future directions.

2. Problem Description

Consider a preemptive real-time system consisting of multiple tasks, and let us assume the following situation.

1. A task \( \tau_0 \) is running.
2. Another task \( \tau_1 \) which has higher priority than \( \tau_0 \) arrives at time \( t \), and preempts \( \tau_0 \).
3. \( \tau_1 \) finishes execution, and \( \tau_0 \) resumes execution.

In this case, the CRPD caused by \( \tau_1 \) is defined by

\[

time required to reload necessary data/code which was present in the cache but displaced from the cache by the other tasks.
\]

...
where CRT denotes the cache refill time, UCB(t) a set of useful cache blocks at time t, and CB(τ₁) a set of cache blocks used by τ₁. A block is the minimum unit of information that can be either present or not present in the cache-main memory hierarchy [5]. We call a block in main memory a memory block, and also call a block in cache a cache block. A cache block is called useful at time t if the content of the cache block is re-referenced before being displaced from the cache in case no preemption occurs.

According to Formula (1), CRPD depends on a set of useful cache blocks at the time of preemption, as well as a set of cache blocks used by the preemption task. However, in general, it is impossible to statically know the useful cache blocks at the time of preemption because applications with conditional behaviors may execute different program paths at run-time (and hence result in different sets of useful cache blocks). In this paper, we aim at tightly bounding worst-case CRPD in order for designers to efficiently implement preemptive real-time systems. In order to guarantee real-time constraints, we need to conservatively assume that all cache blocks are useful at the time of preemption. Then, the problem which we tackle in this paper is to determine the upper bound of the number of cache blocks used by a given task, i.e., the maximum number of elements of CB(τ₁) in Formula (1).

In this paper, we only focus on CRPD caused by instruction cache misses, and CRPD for data caches are ignored. Also, we assume that instruction caches are direct mapped.

For example, let us assume two tasks τ₀ and τ₁ whose location in main memory and cache mapping are are shown in Fig. 1. The task τ₀ is running and all cache blocks \{c₀, · · · , c₇\} are useful. If τ₁ preempts, six cache blocks may be displaced from the cache. Then, the six blocks have to be reloaded into the cache after τ₀ resumes its execution. However, this is a pessimistic scenario: not all of the six useful cache blocks may be displaced. This is because, in general, programs involve conditional statements (e.g., if-then-else) in which the execution path depends on input data, and therefore, not all of the program code may be executed. For example, if program fragments located in memory blocks m₀ and m₁₀ are mutually exclusive, only five blocks need to be reloaded.

Let us consider another example. Assume that a task is running and all cache blocks are useful. Further assume that another task whose control structure is shown in Fig. 2 preempts. In the figure, nodes and edges denote basic blocks and control-flow dependencies, respectively. The size of the nodes represents the size of basic blocks (in terms of the number of instructions). There are two execution paths in the program. We see that the right path contains more instructions to be executed. However, the left path uses more cache blocks (i.e., four cache blocks) than the right path (i.e., three cache blocks). Therefore, the left path leads to longer CRPD than the right path even though the execution path length of the left path is shorter.

In the above example, we see that CRPD caused by a task depends on the program execution path of the task, and that the length of the execution path it not a feasible metric to estimate CRPD. Hence, in order to tightly bound worst-case CRPD, we need to determine the program execution path which uses the maximum number of cache blocks.

This paper proposes an ILP-based approach to determine the program execution path which uses the maximum number of cache blocks. By solving the ILP problem, we can obtain a tight upper bound on CRPD.

It is well recognized that static analysis of program paths is in general undecidable and equivalent to the halting problem. However, it becomes decidable by posing the following restrictions on programs: no recursion, no dynamic data structure, and bounded loops [13]. These conditions hold for many real-time systems, hence we assume these restrictions to make the problem decidable.

### 3. Problem Formulation

This section describes an Integer Linear Programming (ILP) formulation of our program path analysis problem.

#### 3.1 Objective Function

The objective of our formulation is to determine a program execution path which uses the maximum number of cache blocks. Let \(x_i\) be 1 if the cache block \(c_i\) is used by the program, otherwise 0. Then, the number of cache blocks used

\[ CRPD = \text{max} \{ \sum_{i=1}^{n} x_i | UCB(t) \cap CB(\tau) \} \]

where CRT denotes the cache refill time, UCB(t) a set of useful cache blocks at time t, and CB(τ₁) a set of cache blocks used by τ₁.
is defined by
\[
N \sum_{i=0}^{N-1} x_i \quad (2)
\]

where \( N \) denotes the number of cache blocks. Formula (2) is the objective function of our ILP formulation to be maximized.

Obviously, \( x_i \)'s depend on both the program execution path and the location of the program code. Let \( y_j \) denote the execution frequency of the basic block \( b_j \) (i.e., the number of times \( b_j \) is executed), and \( B(C_i) \) the set of basic blocks which use the cache block \( c_i \). For example in Fig. 2, \( B(c_0) \) is \( \{b_0, b_1, b_3\} \) and \( B(c_1) \) is \( \{b_1\} \). Since we assume that the location of program code is fixed, \( B(c_0) \) can be obtained statically. Then, \( x_i \)'s are defined as follows.

\[
x_i = \begin{cases} 1 & \text{if } \sum_{j \in B(C_i)} y_j > 0 \\ 0 & \text{otherwise} \end{cases} \quad (3)
\]

According to Formula (3), \( x_i \)'s are not linear functions. However, they can be linearized in the following manner.

\[
\sum_{j \neq b_j \in B(C)} y_j - U \cdot x_i \leq 0 \quad (4)
\]

\[
\sum_{j \neq b_j \in B(C)} y_j + 1 - x_i > 0 \quad (5)
\]

\[
\sum_{j \neq b_j \in B(C)} y_j = 0 \quad (6)
\]

Here, \( U \) is a large integer number\(^\text{vii}\).

3.2 Constraints on Program Structure and Functionality

Clearly, \( y_j \)'s can not be any value and are constrained by the structure and functionality of the program. The rest of this section describes the constraints all of which must be satisfied. The constraints are based on Li and Malik’s work in [9] and refined to handle more general cases. As an example for explanation, we use the control-flow graph of a program shown in Fig. 3 where the basic block \( b_0 \) is the entry and \( b_4 \) is the exit of the program. \( f_k \)'s are control-flow dependencies between basic blocks, and \( z_k \) denotes the execution frequency of the control flow \( f_k \).

3.2.1 Control-Flow Frequency Constraints

For each basic block \( b_j \) except the entry (the first basic block) and the exit (the last basic block) of the program, total execution frequency of incoming edges to \( b_j \) must be equal to the execution frequency of \( b_j \). Similarly, the total execution frequency of outgoing edges from \( b_j \) must be equal to the execution frequency of \( b_j \). Let \( IN_j \) denote the set of edges coming to the basic block \( b_j \), and \( OUT_j \) the set of edges going from \( b_j \). Then, the following equations must be satisfied.

\[
y_j = \sum_{k \in f_k \in IN_j} z_k \quad (7)
\]

\[
y_j = \sum_{k \in f_k \in OUT_j} z_k + 1 \quad (8)
\]

For example, the basic block \( b_6 \) in Fig. 3, the following equations must be satisfied.

\[
y_6 = z_7 + z_{10} = z_8 + z_9 \quad (9)
\]

Special attention should be paid to the entry and the exit of the program. Let us assume that the basic block \( b_j \) is the entry. The execution frequency of \( b_j \) is defined as follows.

\[
y_j = \sum_{k \in f_k \in IN_j} z_k + 1 \quad (10)
\]

\[
y_j = \sum_{k \in f_k \in OUT_j} z_k \quad (11)
\]

The extra one added to \( \sum_{f_k \in IN_j} z_k \) denotes the control flow given from the operating system. Note that \( IN_j \) may not be empty, e.g., in case there is a branch to \( b_j \). Similarly, the execution frequency of the exit basic block \( b_j \) is defined as follows.

\[
y_j = \sum_{k \in f_k \in OUT_j} z_k \quad (12)
\]

\[
y_j = \sum_{k \in f_k \in OUT_j} z_k + 1 \quad (13)
\]

The extra one added to \( \sum_{f_k \in OUT_j} z_k \) denotes the control flow from the program to the operating system. \( OUT_j \) may not be empty, e.g., in case the last instruction is a conditional branch.

For example in Fig. 3, the following equations must be satisfied for the entry basic block \( b_0 \).

\[
y_0 = z_0 + z_1 = 1 \quad (14)
\]

Formulas (12) and (13) implicitly assume that there exists only one exit basic block. It should be noted that this assumption loses no generality. In case there are multiple exit basic blocks, we introduce a dummy exit basic block in which no instruction exists. Then, we insert a control-flow edge from each real exit to the dummy exit, and regard the dummy exit as an exit. On the other hand, there can not be more than one entry.

\(^{vii}\)\( U \) can not be smaller than \( \sum_{j \in B(C_i)} y_j \) for any \( c_i \). Note that \( U \) is constant.
3.2.2 Loop Count Constraints

In Fig. 3, there is a loop in function subroutine(). The basic block $b_6$ judges the condition of the loop, and $f_3$ ($f_8$) is taken if the condition holds (does not hold). For the basic blocks $b_6$ and $b_7$, there are constraints as follows.

\begin{align}
 y_6 & = z_7 + z_{10} = z_8 + z_9 \tag{15} \\
 y_7 & = z_9 = z_{10} \tag{16}
\end{align}

Using only the above equations, the value of $y_7$ (also $z_9$ and $z_{10}$) cannot be determined because there is no information on the loop count. Recall that every loop is assumed to be bounded, i.e., the maximum number of iterations is known. Let $l$ denote the maximum number of iterations of the loop in Fig. 3. Then, there is a constraint on the loop count as follows.

$$z_9 \leq l \times z_8 \tag{17}$$

If the number of iterations is exactly $l$, the constraint should be

$$z_9 = l \times z_8. \tag{18}$$

3.2.3 Function Call Constraints

We assume that there is no jump across functions except function calls. For each point of function calls in the program code, the execution frequency of the function call edge must be equal to that of the corresponding return edge.

For example in Fig. 3, there are two function calls from main() to subroutine(). One is called at the end of $b_2$ and the other is at the end of $b_3$. In this example, there are constraints on function calls as follows.

$$z_3 = z_5 \tag{19}$$

$$z_4 = z_6 \tag{20}$$

It should be noted again that there can exist multiple exits.

3.2.4 Unreachable Basic Block Constraints

A basic block $b_j$ is unreachable if there is no incoming edge to $b_j$ except the entry of the program, or all basic blocks which precede $b_j$ are unreachable. Such unreachable basic blocks should be removed by the compiler, but sometimes there remain some unreachable basic blocks due to some reasons such as lack of global optimization ability of compilers or linkers.

For each unreachable basic block $b_j$, there is an additional constraint, i.e., $y_j = 0$.

4. Experiments

This section presents a set of experiments demonstrating the effectiveness of our approach. In our experiments, we used the SPARC instruction set architecture [15] with a 2 KB direct-mapped instruction cache as a target processor. The block size of the memory hierarchy was set to be 32 bytes. The benchmark programs were collected from several sources: crc, fir, and qurt are from SNU real-time benchmark suites [14]; wavelet and laplace from 1995 HLS benchmarks [12]; tv-ctrl from [18].

The experiments were achieved by the following step. First, each benchmark program was compiled into assembly code using Sun C Compiler with the “-O2” optimization option. Next, a control-flow graph was constructed from the assembly code, and then, constraints for the ILP problem were derived. Finally, the ILP problem was solved with a public domain ILP solver, named lp_solve [1]. The ILP solver was executed on a 2 GHz Mobile Pentium-IV processor.

To the best of our knowledge, there is no previous work that addresses program path analysis for bounding CRPD. Therefore, our CRPD analysis method was compared with a conservative method which assumes that whole program code is executed regardless of conditional statements. The experimental results are summarized in Table 1. The first column shows the name of benchmark programs. The qurt-kernel benchmark is a kernel of qurt com-

<table>
<thead>
<tr>
<th>Program</th>
<th>C lines</th>
<th>ILP problem size</th>
<th>Cache blocks</th>
<th>CPU time [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Conservative</td>
<td>ILP-based</td>
</tr>
<tr>
<td>crc</td>
<td>125</td>
<td>94</td>
<td>24</td>
<td>24 (1.00)</td>
</tr>
<tr>
<td>fir</td>
<td>314</td>
<td>179</td>
<td>51</td>
<td>50 (0.98)</td>
</tr>
<tr>
<td>qurt</td>
<td>164</td>
<td>91</td>
<td>30</td>
<td>30 (1.00)</td>
</tr>
<tr>
<td>qurt-kernel</td>
<td>138</td>
<td>73</td>
<td>26</td>
<td>21 (0.81)</td>
</tr>
<tr>
<td>laplace</td>
<td>38</td>
<td>22</td>
<td>11</td>
<td>11 (1.00)</td>
</tr>
<tr>
<td>wavelet</td>
<td>41</td>
<td>27</td>
<td>12</td>
<td>8 (0.67)</td>
</tr>
<tr>
<td>tv-ctrl-1</td>
<td>234</td>
<td>353</td>
<td>39</td>
<td>32 (0.31)</td>
</tr>
<tr>
<td>tv-ctrl-2</td>
<td>234</td>
<td>353</td>
<td>39</td>
<td>18 (0.46)</td>
</tr>
<tr>
<td>tv-ctrl-3</td>
<td>234</td>
<td>353</td>
<td>39</td>
<td>21 (0.54)</td>
</tr>
<tr>
<td>tv-ctrl-4</td>
<td>234</td>
<td>353</td>
<td>39</td>
<td>24 (0.62)</td>
</tr>
<tr>
<td>tv-ctrl-5</td>
<td>234</td>
<td>353</td>
<td>39</td>
<td>27 (0.69)</td>
</tr>
<tr>
<td>tv-ctrl-10</td>
<td>234</td>
<td>353</td>
<td>39</td>
<td>35 (0.90)</td>
</tr>
<tr>
<td>tv-ctrl-15</td>
<td>234</td>
<td>353</td>
<td>39</td>
<td>38 (0.97)</td>
</tr>
</tbody>
</table>

Table 1: Comparison of the number of cache blocks between different estimation methods.
puting roots of quadratic equations, and is executed three times in qurt. The tv-ctrl benchmark contains an unbounded loop whose body is executed once when an input data is given. In the experiments, we assumed that the loop is bounded, and varied the upper bound on the loop count from 1 to 15. The program name tv-ctrl-\(n\) in Table 1 denotes that the maximum loop count is set to \(n\).

The second and third columns show the complexity of the ILP problems in terms of the number of decision variables and the number of constraint equations or inequalities.

The fourth column in the table gives the number of cache blocks estimated by the conservative method. The fifth column presents the number of cache blocks estimated by our method. The figures in parenthesis give the ratio describing how tight our method analyzes CRPD compared with the conservative method. For crc, qurt, and laplace, our method generates the same results as the conservative one. Although there exist some conditional statements in crc, qurt, and laplace, those statements are in loops or subroutines that are executed several times, and all paths of the conditional statements can be covered. In fir, there remains an unreachable basic block, and this is why our method generates tighter bound. In tv-ctrl, there are a large number of conditional statements in a loop. Therefore, our method leads to much tighter bound on CRPD when the number of iterations of the loop is small. For tv-ctrl-1, our method obtains 69\% tighter bound. In fact, our approach gives the exact lower bounds in the experiments. However, this is not always true since, in general, a program may have false execution paths which are never taken during execution and our formulation can not exclude such false paths.

The last column in Table 1 shows the CPU time required to solve the ILP problems. For all programs except tv-ctrl, solutions were found within a second. Since tv-ctrl contains much more conditional statements than the other benchmark programs, it takes much longer time to solve the ILP problems for tv-ctrl. For large application programs including a lot of conditionals, it may be impossible for ILP solvers to yield the exact solutions in a reasonable amount of time. However, we believe that our approach is still effective because of the following reason: In general, there is a trade-off between the quality of solutions and the CPU time. Many ILP solvers (e.g., lp_solve) employ iterative or branch-and-bound search algorithms and are capable of generating all intermediate solutions during the search. Therefore, we can run the ILP solvers as long as time permits in order to obtain tight CRPD estimations. Another approach to overcome the long run-time problem is to develop a more efficient algorithm, which is one of our future work.

5. Conclusions

Unpredictable behavior of cache memory makes it difficult to statically analyze the worst-case performance in the design of real-time systems. This problem is exacerbated in the case of preemptive multitask systems due to inter-task cache interference, called Cache-Related Preemption Delay (CRPD). This paper proposed an approach to analyzing the tight upper bound on CRPD which a task might impose on lower-priority tasks. Our work determined the program execution path of the task which requires the maximum number of cache blocks. We formulated the path analysis problem as an Integer Linear Programming (ILP) problem, and the tight bound on CRPD is obtained by solving the ILP problem. Experimental results show that our approach provides up to 69\% tighter bounds on CRPD than a conservative approach. We expect that our approach will yield tighter bounds on CRPD for real-time applications that have many conditions.

Our paper currently assumes only direct-mapped instruction caches. Our ongoing and future work will extend the analysis to handle set-associative instruction caches as well as data caches. We also plan to incorporate this work into schedulability analysis of preemptive multitask systems.

References

kr/realtine


