Program Path Analysis to Bound Cache-Related Preemption Delay in Preemptive Real-Time Systems*

Hiroyuki Tomiyama† Nikil D. Dutt
Center for Embedded Computer Systems
University of California, Irvine, CA 92697–3425, USA
{tomiyama, dutt}@cecs.uci.edu

ABSTRACT

Unpredictable behavior of cache memory makes it difficult to statically analyze the worst-case performance of real-time systems. This problem is exacerbated in case of preemptive multitask systems due to intertask cache interference, called Cache-Related Preemption Delay (CRPD). This paper proposes an approach to analysis of the tight upper bound on CRPD which a task might impose on lower-priority tasks. Our method determines the program execution path of the task 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 previous approach.

1. INTRODUCTION

Due to the growing gap of speed between processors and memories, the impact of cache hits/misses on overall system performance has been increasing. Indeed, cache memory improves the average performance of systems 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 in which real-time constraints have to be satisfied. There are a number of previous research efforts, for example [1, 3, 9, 10], to estimate the tight bound on worst-case execution time of a given task in a single-task environment. However, they cannot directly be applied for preemptive multitask systems because they do not take into account intertask 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. One approach to avoid CRPD is cache partitioning [5, 15]. In this approach, the cache is divided into several disjoint partitions each of which is dedicated to a specific task. Although cache partitioning makes it easier to analyze cache behavior in a preemptive multitask environment, it causes serious degradation of cache performance (therefore, degradation of overall system performance) due to limited cache capacity available to each task.

Some recent studies incorporated CRPD into schedulability analysis of fixed-priority, periodic, preemptive real-time systems [2, 6, 7]. They focus on estimation of worst-case CRPD of instruction cache memory. In [2], CRPD which a task τ might impose on lower-priority tasks is estimated by multiplying the number of cache blocks used by τ by cache refill time. This estimation implicitly assumes that whole program code of τ is loaded into instruction 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 [6, 7] also suffers from this kind of pessimism although their work is more sophisticated than [2].

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 of the task which uses the maximum number of cache blocks using an Integer Linear Programming (ILP) technique.

This paper is organized as follows. Section 2 presents a few motivating examples and describes the path analysis problem addressed in this paper. Section 3 formulates the path analysis problem as an ILP problem. Section 4 presents experiments that demonstrate the effectiveness of our approach. We conclude this paper with a summary in Section 5.

2. PROBLEM DESCRIPTION

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

1. A task τ_0 is running.
2. Another task τ_1 which has higher priority than τ_0 ar-

*This work is supported in part by a grant from NSF (MIP-9708067).
†Research at UCI is supported by JSPS postdoctoral fellowships for research abroad.

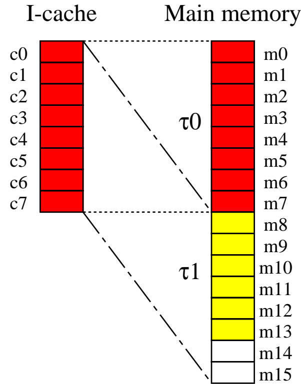

Figure 1: A motivating example.

rives at time t, and preempts τ_0.

3. τ_1 finishes execution, and τ_0 resumes execution.

In this case, the CRPD caused by τ_1 is defined by

$$CRT \times \mid UCB(t) \cap CB(\tau_1) \mid \quad (1)$$

where CRT denotes the cache refill time, $UCB(t)$ a set of *useful* cache blocks at time t, and $CB(\tau_1)$ a set of cache blocks used by τ_1. A *block* is the minimum unit of information that can be either present or not present in the cache-main memory hierarchy [4]. 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 preempting 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 $CB(\tau_1)$ 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 τ_0 and τ_1 whose location in main memory and cache mapping are are shown in Figure 1. The task τ_0 is running and all cache blocks $\{c_0, \cdots, c_7\}$ are useful. If τ_1 preempts, six cache blocks may be displaced from the cache. Then, the six blocks have to be reloaded into the cache after τ_0 resumes its execution.

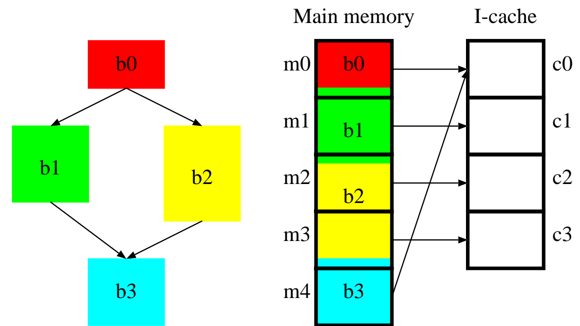

Figure 2: An example showing that longer execution paths do not always result in longer CRPD.

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 statements) 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 m_9 and m_{10} 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 Figure 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 [12]. These conditions hold for many real-time systems, hence we assume these restrictions to make the problem decidable.

3. PROBLEM FORMULATION

3.1 Objective Function

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

$$\sum_{i=0}^{N-1} x_i \tag{2}$$

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 Figure 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.

$$x_i = \begin{cases} 1 & \text{if} \sum_{j|b_j \in B(c_i)} y_j > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

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

$$x_i \in \{0, 1\} \tag{4}$$

$$\sum_{j|b_j \in B(c_i)} y_j - U \cdot x_i \leq 0 \tag{5}$$

$$\sum_{j|b_j \in B(c_i)} y_j + 1 - x_i > 0 \tag{6}$$

Here, U is a large integer number[1].

3.2 Constraints on Program Structure and Functionality

Clearly, y_j's cannot 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 [8] and refined to handle more general cases. As an example for explanation, we use the control-flow graph of a program shown in Figure 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

[1] U cannot be smaller than $\sum_{j|b_j \in B(c_i)}$ for any c_i. Note that U is constant.

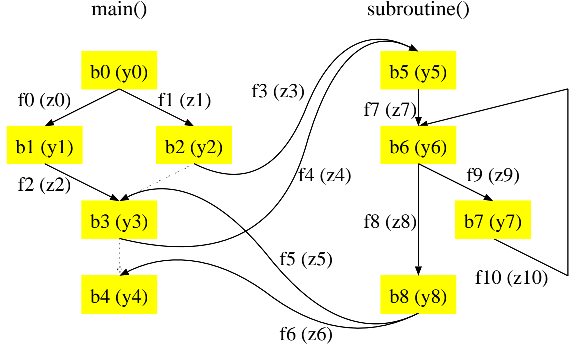

Figure 3: An execution path analysis example.

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.

$$y_j = \sum_{k|f_k \in IN_j} z_k \tag{7}$$

$$= \sum_{k|f_k \in OUT_j} z_k \tag{8}$$

For example, the basic block b_6 in Figure 3, the following equations must be satisfied.

$$y_6 = z_7 + z_{10} = z_8 + z_9 \tag{9}$$

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.

$$y_j = \sum_{k|f_k \in IN_j} z_k + 1 \tag{10}$$

$$= \sum_{k|f_k \in OUT_j} z_k \tag{11}$$

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.

$$y_j = \sum_{k|f_k \in IN_j} z_k \tag{12}$$

$$= \sum_{k|f_k \in OUT_j} z_k + 1 \tag{13}$$

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 Figure 3, the following equations must be satisfied for the entry basic block b_0.

$$y_0 = z_0 + z_1 = 1 \tag{14}$$

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 cannot be more than one entry.

3.2.2 Loop Count Constraints

In Figure 3, there is a loop in function `subroutine()`. The basic block b_6 judges the condition of the loop, and f_9 (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.

$$y_6 = z_7 + z_{10} = z_8 + z_9 \quad (15)$$

$$y_7 = z_9 = z_{10} \quad (16)$$

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 Figure 3. Then, there is a constraint on the loop count as follows.

$$f_9 \leq l \times f_8 \quad (17)$$

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

$$f_9 = l \times f_8. \quad (18)$$

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 Figure 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.

$$z_3 = z_5 \quad (19)$$

$$z_4 = z_6 \quad (20)$$

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., $y_j = 0$.

Table 1: Comparison of the number of cache blocks between different estimation methods

Program	Conservative analysis	ILP-based analysis	CPU time [sec]
`crc`	24	24 (1.00)	0.1
`fir`	51	50 (0.98)	0.2
`qurt`	30	30 (1.00)	0.1
`qurt-kernel`	26	21 (0.81)	0.1
`wavelet`	12	8 (0.67)	0.1
`laplace`	11	11 (1.00)	0.1
`tv-ctrl-1`	39	12 (0.31)	1.8
`tv-ctrl-2`	39	18 (0.46)	7.2
`tv-ctrl-3`	39	21 (0.54)	33.0
`tv-ctrl-4`	39	24 (0.62)	163.8
`tv-ctrl-5`	39	27 (0.69)	423.2
`tv-ctrl-10`	39	35 (0.90)	9,062.3
`tv-ctrl-15`	39	38 (0.97)	21,910.5

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 [14] with a 2KB 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 [13]; `wavelet` and `laplace` from [11]; `tv-ctrl` from [16].

The experiments were achieved by the following step. First, each benchmark program was compiled into assembly code using Sun WorkShop Compiler C ver. 4.2 with the level-two 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`[2]. The ILP solver was executed on a 233MHz Pentium processor with a 64MB memory.

To the best of knowledge, there is no previous work that exploits 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` computing 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 column in the table gives the number of cache blocks estimated by the conservative method. The third column presents the number of cache blocks estimated by our method. The figures in parenthesis give the ratio describ-

[2]`ftp://ftp.es.ele.tue.nl/pub/lp_solve`

ing how tight our method analyzes CRPD compared with the conservative method. In many cases, our method obtains tighter bound than 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.

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.

5. SUMMARY

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 case of preemptive multitask systems due to intertask cache interference, called Cache-Related Preemption Delay (CRPD). This paper proposed an approach to analysis of a tight upper bound on CRPD which a task might impose on lower-priority tasks. Our work determines 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 conditionals.

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

[1] R. D. Arnold, F. Mueller, D. B. Whalley, and M. G. Harmon, "Bounding worst-case instruction cache performance," In *Proc. of Real-Time Systems Symp.*, pp. 172–181, 1994.

[2] J. V. Busquets-Mataix, J. J. Serrano-Martin, R. Ors, P. Gil, and A. Wellings, "Adding instruction cache effect to schedulability analysis of preemptive real-time systems," In *Proc. of 2nd Real-Time Technology and Application Symp.*, 1996.

[3] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems," *J. Real-Time Systems, Kluwer Academic Publishers*, vol. 17, no. 2–3, pp. 131–181, November 1999.

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 2nd edition, 1996.

[5] D. B. Kirk, "SMART (strategic memory allocation for real-time) cache design," In *Proc. of Real-Time Systems Symp.*, pp. 97–108, 1989.

[6] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim, "Enhanced analysis of cache-related preemption delay in fixed-priority preemptive scheduling," In *Proc. of 18th Real-Time Systems Symp.*, 1997.

[7] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *IEEE Trans. Computers*, vol. 47, no. 6, pp. 700–713, June 1998.

[8] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *IEEE Trans. CAD/ICAS*, vol. 16, no. 12, pp. 1477–1487, December 1997.

[9] Y.-T. S. Li, S. Malik, and A. Wolfe, "Cache modeling for real-time software: Beyond direct mapped instruction caches," In *Proc. of Real-Time Systems Symp.*, pp. 254–263, 1996.

[10] Y.-T. S. Li, S. Malik, and A. Wolfe, "Performance estimation of embedded software with instruction cache modeling," *ACM Trans. Design Automation of Electronic Systems*, vol. 4, no. 3, pp. 257–279, July 1999.

[11] P. R. Panda and N. Dutt, "1995 high level synthesis design repository," In *Proc. of 8th Int'l Symp. System Synthesis*, 1995.

[12] P. Puschner and Ch. Koza, "Calculating the maximum execution time of real-time programs," *Journal of Real-Time Systems*, vol. 1, no. 2, pp. 160–176, September 1989.

[13] SNU Real-Time Benchmarks, http://archi.snu.ac.kr/realtime/benchmark.

[14] SPARC International, Inc., *The SPARC Architecture Manual Version 8*, 1992.

[15] A. Wolfe, "Software-based cache partitioning for real-time applications," *Journal of Computer & Software Engineering*, vol. 1, no. 3, pp. 315–327, 1994.

[16] H. Yasuura, H. Tomiyama, A. Inoue, and F. N. Eko, "Embedded system design using soft-core processor and Valen-C," *IIS Journal of Information Science and Engineering*, vol. 14, no. 3, pp. 587–603, September 1998.