# Accurate Estimation of Cache-Related Preemption Delay

Hemendra Singh Negi     Tulika Mitra     Abhik Roychoudhury

School of Computing
National University of Singapore
Republic of Singapore 117543.
[hemendra,tulika,abhik]@comp.nus.edu.sg

## ABSTRACT

Multitasked real-time systems often employ caches to boost performance. However the unpredictable dynamic behavior of caches makes schedulability analysis of such systems difficult. In particular, the effect of caches needs to be considered for estimating the inter-task interference. As the memory blocks of different tasks can map to the same cache blocks, preemption of a task may introduce additional cache misses. The time penalty introduced by these misses is called the Cache-Related Preemption Delay (CRPD).

In this paper, we provide a program path analysis technique to estimate CRPD. Our technique performs path analysis of both the preempted and the preempting tasks. Furthermore, we improve the accuracy of the analysis by estimating the possible states of the entire cache at each possible preemption point rather than estimating the states of each cache block independently. To avoid incurring high space requirements, the cache states can be maintained symbolically as a Binary Decision Diagram. Experimental results indicate that we obtain tight CRPD estimates for realistic benchmarks.

## Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and Embedded Systems*

## General Terms

Measurement, Performance, Reliability.

## Keywords

Caches, Multitasking, Preemption, Formal Analysis.

## 1. INTRODUCTION

Real-time embedded systems involve tasks operating under deadlines which need to be met. Consequently schedu-

lability analysis of such tasks has been a topic of active research. For each task, schedulability analysis needs to know the worst case execution time (WCET), that is, an upper bound on the task execution time. For preemptive task scheduling, we also need to estimate the inter-task interference (the additional execution time introduced when a high priority task preempts a low priority task). However, performance enhancing micro-architectural features of modern processors make such estimation difficult.

Ideally, the WCET of a program should be the cost of its longest path. However, this assumes that the execution time of each instruction is constant. In reality, the execution time of a memory access instruction depends on the past instructions executed (which decides whether it is a cache hit/miss). Thus, path analysis of the program cannot be done at the level of assembly code; it also needs to take into account the effect of micro-architectural features such as caches [5, 9, 11, 14]. This explains the importance of cache behavior modeling for WCET analysis.

Caches have a role to play in inter-task interference estimation as well. Consider a low priority task $\tau$ which is preempted by a higher priority task $\tau'$. Let the set of cache blocks to which the memory blocks of $\tau$ ($\tau'$) get mapped be $C_\tau$ ($C_{\tau'}$). If $C_\tau \cap C_{\tau'} \neq \emptyset$ then $\tau'$ may displace some memory blocks of $\tau$ from the cache, resulting in additional cache misses when $\tau$ resumes. This effect of task preemption on cache performance is known in the literature as *Cache-Related Preemption Delay* (CRPD).

CRPD estimation is important from the point of view of schedulability analysis. In particular, [7, 8] reports how CRPD can be used to derive accurate response times of multiple periodic tasks running on a single processor for fixed-priority preemptive scheduling. We can avoid CRPD estimation by ignoring arbitrary preemption (as in [10] which only studies the interference at the beginning of a periodic task's execution). Alternatively, we can employ cache partitioning [6], where each task is allowed to use only a portion of the cache. However, this leads to severe degradation of performance. Thus, for multitasked preemptive real-time systems with caches, CRPD estimation is necessary.

When a task $\tau$ is preempted by a higher priority task $\tau'$ what are the additional cache misses introduced? To answer this question, we need to consider the following factors. First, all the memory blocks of $\tau$ which are in the cache when $\tau$ is preempted may not be re-referenced after resumption. Secondly, all the memory blocks of $\tau$ which are in the cache when $\tau$ is preempted may not be replaced by $\tau'$. Thirdly, there are several possible cache contents when

$\tau$ is preempted (resulting from the different paths of $\tau$), and when $\tau$ resumes execution (resulting from the different paths of $\tau'$.) Last but not the least, there are several possible memory reference patterns after $\tau$ resumes execution (due to different paths of $\tau$).

The importance of some of these factors on CRPD is mentioned in [4]. However, *no estimation technique is given* and only simulation results are presented. The work of Lee et. al. [7] performs set based analysis of the cache blocks used by the preempted task $\tau$ before and after preemption. This is extended in [8] to also consider the set of cache blocks used by the high-priority task $\tau'$. Tomiyama and Dutt perform implicit path analysis of the preempting task $\tau'$ but ignore the effects of the preempted task $\tau$ [13].

In this paper, we propose methods to improve the accuracy of CRPD estimation. As per existing works [7, 8, 13], we only consider the effects of the instruction cache (and not data cache). Our technique performs path analysis of both $\tau$ and $\tau'$. Furthermore, we compute the possible states of the cache when the lower priority task $\tau$ is preempted and when the higher priority task $\tau'$ is completed. This is more accurate than existing set-based analysis techniques which estimate the cache states by inferring the set of memory blocks which may exist in each cache block. Suppose that the cache has two blocks and the possible states of the cache when $\tau$ is preempted are as follows

$$\{\langle m_0, m_1 \rangle, \langle m_2, m_3 \rangle\}$$

In the above $m_0, m_1, m_2, m_3$ are memory blocks with $m_0, m_2$ mapping to the first cache block and $m_1, m_3$ mapping to the second cache block. Existing path analysis techniques such as [7, 8] will report the possible states as follows.

Content of first cache block $\in \{m_0, m_2\}$
Content of second cache block $\in \{m_1, m_3\}$

This actually denotes the four cache states

$$\{\langle m_0, m_1 \rangle, \langle m_0, m_3 \rangle, \langle m_2, m_1 \rangle, \langle m_2, m_3 \rangle\}$$

clearly leading to an over-approximation.

Note that we represent the cache states at a program point in a task as a set of tuples (where each tuple denotes an assignment of memory blocks to the cache blocks). This is more expensive than representing the possible contents of each cache block as a set of memory blocks. To avoid an exponential blow-up in the space consumption we can represent the possible cache states at any program point implicitly as a Binary Decision Diagram (BDD) [1]. A BDD is an efficient data structure for representing a propositional logic formula.

We now describe the core CRPD estimation technique. Section 3 explains the technique with a concrete example. Section 4 gives experimental results regarding accuracy of our estimation technique as well as its time and space consumption. Concluding remarks and discussions on space reduction appear in Section 5.

## 2. CRPD ESTIMATION

Recall from Section 1 that when a task $\tau$ is preempted by a higher priority task $\tau'$, the additional cache misses introduced depend on (a) cache content when $\tau$ is preempted, (b) cache content after $\tau'$ completes execution, and (c) cache reference pattern after $\tau$ resumes execution. Clearly, (a) and

(c) require analysis of $\tau$ – the low priority task that gets preempted and (b) requires analysis of $\tau'$ – the high priority preempting task. Our analysis considers both $\tau$ and $\tau'$.

*Cache States.* We use the terminology "cache state" to denote the contents of all the cache blocks. For simplicity of exposition, we assume a direct-mapped cache. However, the technique can be easily generalized to set-associative caches. For a direct mapped cache with $n$ blocks, a **cache state** is a vector of $n$ elements $c[0 \ldots n-1]$ where $c[i] = m$ if cache block $i$ holds memory block $m$. Otherwise, if the $i$th cache block does not hold any memory block we denote this as $c[i] = \perp$. Thus, a cache state is a vector of length $n$ where each element of the vector belongs to $M \cup \{\perp\}$ ($M$ is the set of all memory blocks). In an abuse of notation, we assume that any operation $\odot$ over $M \cup \{\perp\}$ can be applied to cache states (by applying the operation pointwise to its elements). For example, if $\odot$ is a binary operation over $M \cup \{\perp\}$ and $c, c'$ are cache states then $c \odot c' = c''$ where $c''[i] = c[i] \odot c'[i]$.

DEFINITION 1 (RCS). *The Reaching Cache States at a basic block $B$ of a program, denoted as $RCS_B$, is the set of possible cache states when $B$ is reached via any incoming program path.*

This notion helps us capture the cache content when the low-priority task is preempted.

DEFINITION 2 (LCS). *Given a program, we define the Live Cache States at a basic block $B$, denoted as $LCS_B$, as the possible first memory references to cache blocks via any outgoing program path from $B$.*

This notion helps us capture the cache reference pattern when the low-priority task resumes after preemption.

### 2.1 Computing Cache States

We consider the end of any basic block as a possible preemption point. [7] shows why it is sufficient to consider only these program points as candidate preemption points for CRPD estimation. The same reasoning can be extended to our analysis technique as well.

We now show how $RCS_B$ and $LCS_B$ are computed for each basic block $B$. This involves a modification of the work in [7, 8] since our notion of cache states is more elaborate. To compute $RCS_B$, we compute the quantities $RCS_B^{IN}$ and $RCS_B^{OUT}$ as a least fixed point. Once the fixed point is reached, we set $RCS_B = RCS_B^{OUT}$. Initially,

$$RCS_B^{IN} = \emptyset \qquad\qquad RCS_B^{OUT} = gen_B$$

For a basic block $B$, we define $gen_B = [m_0, \ldots, m_{n-1}]$ where $m_i = m$ if $m$ is the *last* memory block in $B$ that maps to cache block $i$ and $\perp$ if no memory block in $B$ maps to cache block $i$. Thus $gen_B$ records all the memory blocks introduced into the cache by basic block $B$. The iterative equations are as follows:

$$RCS_B^{IN} = \bigcup_{p \in predecessor(B)} RCS_p^{OUT}$$

$$RCS_B^{OUT} = \{r \odot gen_B \mid r \in RCS_B^{IN}\}$$

The operation $\odot$ is defined over memory blocks as

$$m \odot m' = \begin{cases} m' & \text{if } m' \neq \perp \\ m & \text{otherwise} \end{cases}$$

It is extended to cache states by applying the operation to its individual elements in the usual way.

The computation of $LCS_B$ is similar. Again we compute $LCS_B^{OUT}$ and $LCS_B^{IN}$ as a least fixed point and we set $LCS_B = LCS_B^{OUT}$ once fixed point is reached. Initially:

$$LCS_B^{OUT} = \emptyset \qquad\qquad LCS_B^{IN} = gen_B$$

Here for a basic block $B$, we define $gen_B = [m_0, \ldots, m_{n-1}]$ where $m_i = m$ if $m$ is the *first* memory block in $B$ that maps to cache block $i$ and $\perp$ if no memory block in $B$ maps to cache block $i$. The iterative equations are as follows:

$$LCS_B^{OUT} = \bigcup_{s \in successor(B)} LCS_s^{IN}$$

$$LCS_B^{IN} = \{l \odot gen_B \mid l \in LCS_B^{OUT}\}$$

The operation $\odot$ is as defined in the computation of $RCS_B$.

## 2.2 Computing Utility Vectors

Suppose the low-priority task $\tau$ is preempted at basic block $B$. Then $RCS_B$ captures the possible cache states when $\tau$ is preempted and $LCS_B$ captures the possible cache usages when $\tau$ resumes execution. This brings us to the notion of "useful cache blocks". These are cache blocks containing memory blocks at the point $B$ which are referenced after $B$ (before being replaced from the cache). To formally capture this notion in our representation of cache states we define the **Cache Utility Vector** $CUV_B$ at basic block $B$. The following section discusses why this notion is more fine-grained than the "set of useful cache blocks" defined in [7].

$$CUV_B = \{l \approx r \mid l \in LCS_B, r \in RCS_B\}$$

The operator $\approx$ is the equality predicate over memory blocks, that is $m \approx m' = 1$ iff $m = m'$. It is extended to cache states by applying $\approx$ pointwise to its elements. For a cache with $n$ blocks, $CUV_B$ is a set of boolean vectors of length $n$. Each member of $CUV_B$ records the useful cache blocks for some possible incoming and outgoing path of $B$.

Determining the useful cache blocks at the preemption point of the low-priority task $\tau$ is not enough. We also need to compute the possible cache states after the completion of the high priority task $\tau'$. This will provide the usages of the $n$ cache blocks along the different program paths of $\tau'$. To formally capture this notion, we define the **Final Usage Vector** $FUV_{\tau'}$ of the high-priority task $\tau'$. Let $end$ be the last basic block of task $\tau'$ (the one that is executed just before completion of the task). Then

$$FUV_{\tau'} = \{used(r) \mid r \in RCS_{end}\}$$

Note that $RCS_{end}$ is computed as a fixed point as described before. The function $used$ is defined over cache states as $used(r) = r'$ where $r'$ is the following bit-vector:

$$r'[i] = \begin{cases} 1 & \text{if } r[i] \neq \perp \\ 0 & \text{otherwise} \end{cases}$$

## 2.3 Computing CRPD

Given $CUV_B$ at each basic block $B$ of the preempted task $\tau$ and $FUV_{\tau'}$ of the preempting task $\tau'$, we compute the CRPD as follows. First, we define the intersection of $FUV_{\tau'}$ with $CUV_B$ of task $\tau$ at all possible preemption points.

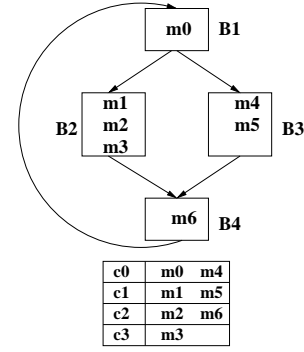$$Delay(\tau, \tau') = \bigcup_{B \in BasicBlks(\tau)} \{c \wedge f \mid c \in CUV_B, f \in FUV_{\tau'}\}$$



**Figure 1: An example control flow graph.**

where $\wedge$ is the AND operation applied to boolean vectors and $BasicBlks(\tau)$ is the set of all basic blocks of $\tau$. Thus, if a cache block contains a useful memory block of $\tau$ and it is used by the memory blocks of $\tau'$ then it causes an additional cache miss. Finally

$$CRPD(\tau, \tau') = max\{cnt\_ones(v) \mid v \in Delay(\tau, \tau')\}$$

where $cnt\_ones(v)$ returns the number of 1's in bit vector $v$. Of course, to calculate the actual delay in terms of number of processor clock cycles $CRPD(\tau, \tau')$ should be multiplied by the cache miss penalty.

## 3. AN EXAMPLE

In this section, we illustrate our technique with a simple control flow graph (CFG) shown in Figure 1. The CFG consists of four basic blocks (B1 - B4) and seven memory blocks ($m0 - m6$) within a loop with a single if-then-else statement. We assume a direct mapped instruction cache with four cache blocks ($c0 - c3$). The mapping of memory blocks to cache blocks is shown in the figure.

The fixed point iterations to compute $RCS_B$ are shown in detail in Table 1. Note that we set $RCS_B := RCS_B^{OUT}$ after fixed point is reached. We set $gen_B$ as follows.

$$\begin{aligned} gen_{B1} &= [m0, \perp, \perp, \perp] \\ gen_{B2} &= [\perp, m1, m2, m3] \\ gen_{B3} &= [m4, m5, \perp, \perp] \\ gen_{B4} &= [\perp, \perp, m6, \perp] \end{aligned}$$

We should also mention that if there are two cache states $c$ and $c'$ in a set of cache states $C$ such that $c'$ is *subsumed* by $c$, then $c'$ is removed from $C$. A cache state $c'$ is subsumed by another cache state $c$ if $\forall i \ \ c'[i] = c[i] \ \ or \ \ c'[i] = \perp$. Thus, in iteration 4 for basic block B3,

$$RCS_{B3}^{IN} = \{[m0, m1, m6, m3], [m0, m5, m6, \perp]\}$$

and $gen_{B3} = [m4, m5, \perp, \perp]$. Therefore,

$$RCS_{B3}^{OUT} = \{[m4, m5, m6, m3], [m4, m5, m6, \perp]\}$$

However, $[m4, m5, m6, \perp]$ is subsumed by $[m4, m5, m6, m3]$ and hence $RCS_{B3}^{OUT} = \{[m4, m5, m6, m3]\}$.

| Iteration | Basic Block | $RCS^{IN}$ | $RCS^{OUT}$ |
|---|---|---|---|
| 1 | B1 | $\emptyset$ | $[m0, \bot, \bot, \bot]$ |
|   | B2 | $\emptyset$ | $[\bot, m1, m2, m3]$ |
|   | B3 | $\emptyset$ | $[m4, m5, \bot, \bot]$ |
|   | B4 | $\emptyset$ | $[\bot, \bot, m6, \bot]$ |
| 2 | B1 | $[\bot, \bot, m6, \bot]$ | $[m0, \bot, m6, \bot]$ |
|   | B2 | $[m0, \bot, \bot, \bot]$ | $[m0, m1, m2, m3]$ |
|   | B3 | $[m0, \bot, \bot, \bot]$ | $[m4, m5, \bot, \bot]$ |
|   | B4 | $[\bot, m1, m2, m3], [m4, m5, \bot, \bot]$ | $[\bot, m1, m6, m3], [m4, m5, m6, \bot]$ |
| 3 | B1 | $[\bot, m1, m6, m3], [m4, m5, m6, \bot]$ | $[m0, m1, m6, m3], [m0, m5, m6, \bot]$ |
|   | B2 | $[m0, \bot, m6, \bot]$ | $[m0, m1, m2, m3]$ |
|   | B3 | $[m0, \bot, m6, \bot]$ | $[m4, m5, m6, \bot]$ |
|   | B4 | $[m0, m1, m2, m3], [m4, m5, \bot, \bot]$ | $[m0, m1, m6, m3], [m4, m5, m6, \bot]$ |
| 4 | B1 | $[m0, m1, m6, m3], [m4, m5, m6, \bot]$ | $[m0, m1, m6, m3], [m0, m5, m6, \bot]$ |
|   | B2 | $[m0, m1, m6, m3], [m0, m5, m6, \bot]$ | $[m0, m1, m2, m3]$ |
|   | B3 | $[m0, m1, m6, m3], [m0, m5, m6, \bot]$ | $[m4, m5, m6, m3]$ |
|   | B4 | $[m0, m1, m2, m3], [m4, m5, m6, \bot]$ | $[m0, m1, m6, m3], [m4, m5, m6, \bot]$ |
| 7 | B1 | $[m0, m1, m6, m3], [m4, m5, m6, m3]$ | $[m0, m1, m6, m3], [m0, m5, m6, m3]$ |
|   | B2 | $[m0, m1, m6, m3], [m0, m5, m6, m3]$ | $[m0, m1, m2, m3]$ |
|   | B3 | $[m0, m1, m6, m3], [m0, m5, m6, m3]$ | $[m4, m5, m6, m3]$ |
|   | B4 | $[m0, m1, m2, m3], [m4, m5, m6, m3]$ | $[m0, m1, m6, m3], [m4, m5, m6, m3]$ |

**Table 1: Computation of $RCS_B$ for the CFG in Figure 1. Iteration steps 5 and 6 are not shown.**

We can also compute $LCS_B$ in a similar fashion

$$LCS_{B1} = \{[m0, m1, m2, m3], [m4, m5, m6, m3]\}$$
$$LCS_{B2} = \{[m0, m1, m6, m3], [m0, m5, m6, m3]\}$$
$$LCS_{B3} = \{[m0, m1, m6, m3], [m0, m5, m6, m3]\}$$
$$LCS_{B4} = \{[m0, m1, m2, m3], [m0, m5, m6, m3]\}$$

Given LCS and RCS for each basic block, we can compute

$$CUV_{B1} = \{[1, 1, 0, 1], [0, 0, 1, 1], [0, 1, 1, 1], [1, 0, 0, 1]\}$$
$$CUV_{B2} = \{[1, 1, 0, 1], [1, 0, 0, 1]\}$$
$$CUV_{B3} = \{[0, 1, 1, 1], [0, 0, 1, 1]\}$$
$$CUV_{B4} = \{[1, 1, 0, 1], [1, 0, 1, 1], [0, 1, 1, 1], [0, 0, 0, 1]\}$$

There are at most 3 useful cache blocks for B1 and B4.

Now let us illustrate the advantage of our technique over separate analysis of each cache block [7, 8]. In that case, $RCS_B$ and $LCS_B$ have a set of reaching memory blocks for each cache block as shown in the following.

$$RCS_{B1} = [\{m0\}, \{m1, m5\}, \{m6\}, \{m3\}]$$
$$RCS_{B2} = [\{m0\}, \{m1\}, \{m2\}, \{m3\}]$$
$$RCS_{B3} = [\{m4\}, \{m5\}, \{m6\}, \{m3\}]$$
$$RCS_{B4} = [\{m0, m4\}, \{m1, m5\}, \{m6\}, \{m3\}]$$

$$LCS_{B1} = [\{m0, m4\}, \{m1, m5\}, \{m2, m6\}, \{m3\}]$$
$$LCS_{B2} = [\{m0\}, \{m1, m5\}, \{m6\}, \{m3\}]$$
$$LCS_{B3} = [\{m0\}, \{m1, m5\}, \{m6\}, \{m3\}]$$
$$LCS_{B4} = [\{m0\}, \{m1, m5\}, \{m2, m6\}, \{m3\}]$$

Let us consider $RCS_{B4}$. From separate analysis of cache blocks, we infer that $RCS_{B4}$ has four possible cache states: $[m0, m1, m6, m3]$, $[m0, m5, m6, m3]$, $[m4, m1, m6, m3]$, and $[m4, m5, m6, m3]$. However, our combined analysis of cache blocks infers that only two of these cache states are feasible. The identification of these infeasible cache states leads to decrease in the number of useful cache blocks (computed via intersection of $RCS_B$ and $LCS_B$) at each program point. For example, our analysis infers at most 3 useful cache blocks for both B1 and B4 (Even though each of the cache blocks is useful along some path, all 4 of them are not useful along

| Program | Description |
|---|---|
| matsum | Summation of two $100 \times 100$ matrices |
| qsort | Non-recursive quick sort algorithm |
| crc | Cyclic redundancy check program |
| sqrt | Square root calculation |
| eqntott | Drawn from SPEC'92 integer benchmarks |
| des | Data Encryption Standard |
| whet | Whetstone benchmark |
| ssearch | Pratt-Boyer-Moore string search |
| math | Basic math within nested loop |

**Table 2: Description of benchmark programs.**

any path). Whereas, with separate analysis of cache blocks, we get 4 useful cache blocks for B1 and B4.

Note that we also maintain the Final Usage Vector (FUV) of the high priority task as a set of boolean vectors. This leads to further accuracy in CRPD analysis. For example, suppose we compute $FUV = \{[1, 0, 1, 0], [1, 1, 0, 0]\}$. This will allow our analysis to estimate the number of useful cache blocks to be 2 leading to even tighter CRPD estimation. In case of separate analysis of each cache block, the above $FUV$ will be approximated as $FUV = [1, 1, 1, 0]$. This only allows the separated analysis to tighten the CRPD estimate to 3 cache misses (as compared to the 4 cache misses estimated earlier based on the low-priority task).

## 4. EXPERIMENTAL RESULTS

In this section, we present the accuracy and performance of our CRPD estimation technique.

### 4.1 Methodology

We select nine different benchmarks for our experiments (refer Table 2). Most of these benchmarks are from [12] and [11]. We use the Simplescalar architectural simulation platform [2] in the experiments. All the benchmarks are compiled to Simplescalar assembly language with modified gcc. A prototype analyzer written by us accepts assembly language code, disassembles it, identifies the basic blocks and constructs the control flow graph (CFG). Given the CFG of the low-priority and the high-priority task as well as the instruction cache configuration, our analyzer computes

| LP Task | HP Task | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | matsum | | | eqntott | | | sqrt | | |
| | A | C | S | A | C | S | A | C | S |
| qsort | 19 | 20 | 24 | 16 | 22 | 28 | 18 | 19 | 26 |
| crc | 17 | 17 | 18 | 17 | 21 | 22 | 18 | 19 | 20 |
| ssearch | 19 | 22 | 23 | 19 | 25 | 27 | 21 | 22 | 25 |
| des | 22 | 23 | 24 | 21 | 24 | 26 | 22 | 22 | 25 |
| whet | 20 | 21 | 22 | 20 | 25 | 25 | 23 | 23 | 24 |
| math | 18 | 22 | 23 | 20 | 25 | 27 | 20 | 22 | 25 |

**Table 3: Accuracy of CRPD analysis for a 32-block cache. A stands for actual number of misses in CRPD (found by exhaustive simulation), C stands for our combined analysis of all cache blocks and S stands for separate analysis of each cache block.**

| Task | Combined | Separate |
|---|---|---|
| matsum | 23 | 24 |
| eqntott | 26 | 28 |
| sqrt | 23 | 26 |

**Table 4: Maximum number of cache blocks used by high priority task for a 32-block cache.**

| Task | # of Cache Blocks | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 8 | | 16 | | 32 | | 64 | |
| | C | S | C | S | C | S | C | S |
| qsort | 1 | 1 | 14 | 14 | 28 | 32 | 62 | 63 |
| crc | 2 | 2 | 12 | 12 | 26 | 26 | 48 | 48 |
| ssearch | 6 | 6 | 15 | 16 | 31 | 31 | 59 | 59 |
| des | 0 | 0 | 12 | 12 | 30 | 30 | 60 | 64 |
| whet | 1 | 2 | 11 | 14 | 29 | 29 | 59 | 59 |
| math | 4 | 5 | 14 | 16 | 30 | 31 | 63 | 64 |

**Table 5: Maximum number of useful cache blocks of the low-priority task at any program point. C stands for combined analysis of all cache blocks and S stands for separate analysis of each cache block.**

| Task | # of Cache Blocks | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 8 | | 16 | | 32 | | 64 | |
| | #B | D | #B | D | #B | D | #B | D |
| qsort | 0 | 0 | 1 | 1 | 2 | 4 | 1 | 1 |
| crc | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| ssearch | 0 | 0 | 2 | 2 | 0 | 0 | 0 | 0 |
| des | 0 | 0 | 0 | 0 | 9 | 2 | 47 | 15 |
| whet | 7 | 1 | 11 | 4 | 0 | 0 | 0 | 0 |
| math | 2 | 1 | 4 | 5 | 2 | 3 | 1 | 1 |

**Table 6: Comparison of combined and separate analysis for low-priority task. #B denotes the number of basic blocks at which useful cache block count differs and D denotes the maximum of these differences.**

the CRPD of the tasks. Our analyzer is parameterizable with respect to cache block size and number of cache blocks.

## 4.2 Accuracy

First we present the accuracy of our analysis method in computing CRPD. We choose matsum, eqntott, and sqrt as high priority tasks and others as low priority tasks. For each pair of low-priority and high-priority tasks, Table 3 presents CPRD in terms of number of useful cache blocks that may get replaced by the high-priority task. We assume a direct mapped instruction cache with 32 cache blocks and block size of 8 bytes. The results show that our analysis yields estimates which is are very close to the actual CRPD. Furthermore, as compared to the separated analysis of [8], our analysis obtains much tighter bound on CRPD with improvements as high as 37% for some benchmarks. Let us now analyze the factors working behind this improvement.

Table 4 shows the maximum number of cache blocks used by the high-priority task for a direct mapped cache with 32 blocks. For combined analysis, this is $max\{cnt\_ones(v) \mid v \in FUV_{\tau'}\}$ where $\tau'$ is the high-priority task; $cnt\_ones(v)$ returns the number of 1s in the bit vector $v$. Separate analysis per cache block results in higher value as it marks a cache block as useful if that block is used along any path in the task. However, cache block $c$ and $c'$ may never be used together along any program path, even though they may be used in different program paths. Our method captures that scenario and therefore results in lesser cache block usage by high priority task. Note that [13] also made the above observation. However, their analysis is based on integer linear programming (ILP) and is limited to only the high-priority task. Similarly, Table 5 shows the maximum number of useful cache blocks of the low-priority task at any program point. For combined analysis, this is equal to $max\{cnt\_ones(v) \mid v \in CUV_B, B \in BasicBlks(\tau)\}$ where $\tau$ is the low-priority task. Again separate analysis per cache block results in higher value as was explained earlier.

Table 5 shows that the maximum number of useful cache blocks in the low-priority task are quite close in our combined analysis and the separated per cache-block analysis.

However, this does not indicate that the CRPD values produced by the two techniques are equally close (Table 3). Recall that the CRPD of tasks $\tau, \tau'$ is defined as:

$$max\{cnt\_ones(c \wedge f) \mid c \in CUV_B, B \in BasicBlks(\tau),$$
$$f \in FUV_{\tau'}\}$$

Thus, apart from the effects of the high-priority task, the CRPD depends on the estimated number of useful cache blocks at each preemption point of $\tau$. In other words, even though the maximum number of useful cache blocks over all preemption points of $\tau$ (shown in Table 5) might be the same in both analysis techniques, the estimated number of useful cache blocks in individual preemption points of $\tau$ may be different. Table 6 shows the number of preemption points (basic blocks) at which useful cache block count differs in the two analysis as well as the maximum of these differences. Such differences in the useful cache block estimates for individual preemption points of $\tau$ produces a difference in CRPD when these estimates are combined with the cache block usage estimates from $\tau'$.

## 4.3 Time and Space Overheads

Table 7 shows the time taken by our analyzer to compute $CUV_B$ for all basic blocks in the low-priority task. The time taken to compute $FUV$ of high-priority task is negligible and is not shown here. For all the benchmarks, it takes less than 1.5 minute on a Pentium IV 1.7GHz CPU with 1GB memory. One concern with our approach is that the number of cache states may increase exponentially while computing $RCS_B$ and $LCS_B$. Assuming a task with $B$ basic blocks, $M$ memory blocks and a cache with $C$ blocks, $M/C$ memory blocks map to one cache block. Therefore, in the worst case, we can have total of $(M/C)^C \times B$ cache states. However, our experimental results indicate that the number of cache states is much smaller in practice. For example, for qsort

| Task | # of Cache Blocks | | | |
|---|---|---|---|---|
| | **8** | **16** | **32** | **64** |
| qsort | 0.003 | 0.004 | 0.022 | 0.082 |
| crc | 0.016 | 0.015 | 0.164 | 65.663 |
| ssearch | 0.007 | 0.025 | 1.613 | 16.168 |
| des | 0.010 | 0.022 | 0.525 | 55.329 |
| whet | 0.005 | 0.023 | 4.189 | 89.858 |
| math | 0.013 | 0.059 | 1.061 | 8.414 |

**Table 7: Time to compute useful cache blocks for low-priority task in sec.**

with $B = 40$, $M = 490$ and $C = 8$, we only get 68 cache states for $LCS_B$ and 69 for $RCS_B$. When we set $C = 64$, number of cache states in $LCS_B$ and $RCS_B$ increase to only 425 and 600 respectively.

## 5. DISCUSSION

In this paper, we have studied the problem of estimating *Cache-Related Preemption Delay* (CRPD) in preemptive multitasked real-time systems. CRPD is the delay introduced in a low-priority task owing to additional cache misses caused via preemption (by a higher priority task). To enable accurate analysis, our approach maintains possible cache contents (1) at each preemption point of the low-priority task and (2) at the end of high-priority task. Our experimental results show that our approach leads to tighter estimates than existing approaches which estimate the possible contents of each cache block separately (for the low-priority as well as the high-priority task). Note that in our analysis, we have throughout assumed only two tasks. Extending the technique to $m$ tasks is straightforward: we need to consider all possible preemption sequences.

One possible concern regarding our analysis technique is a blow-up in space consumption. As observed in the previous section, none of our benchmarks suffered from an exponential space blow-up due to our decision to represent cache states (instead of the content of each cache block separately).

Furthermore, we can handle possible blow-up in the representation of reaching/live cache states by adopting a symbolic representation of $LCS_B$ and $RCS_B$. In particular, we can represent $LCS_B$ and $RCS_B$ (for any basic block $B$) as a Binary Decision Diagram (BDD). A BDD is a compact representation for a boolean function [1]. Consider a direct mapped cache with $n$ cache blocks and at most $k$ memory blocks mapping to each cache block. The content of any cache block in a cache state can then be captured with $\lceil log_2(k+1) \rceil$ boolean variables. For example if two memory blocks $m$ and $m'$ can map to a cache block, then the cache block can have three possible contents: $m$, $m'$ and $\perp$ (that is, it does not hold any memory block). These three possibilities can be captured with $\lceil log_2(3) \rceil = 2$ boolean variables. A cache state simply denotes the content of all the $n$ cache blocks. Therefore, a cache state can be represented via truth assignment of $n * \lceil log_2(k + 1) \rceil$ boolean variables. A set of cache states $S$ simply denotes a set of truth assignments of these $n * \lceil log_2(k + 1) \rceil$ boolean variables, call it $V_S$. Thus, a set of cache states $S$ can be represented as a propositional logic formula (or a BDD) $f_S$ over the $n * \lceil log_2(k+1) \rceil$ boolean variables (s.t. $f_S$ is true for only the truth assignments in $V_S$). We use this BDD representation for $LCS_B$ and $RCS_B$ (for any basic block $B$) since they are simply sets of cache

states. The variable ordering in the BDD representation is obtained from a total ordering of the cache blocks. Given the BDD representation for $LCS_B$ and $RCS_B$, the fixed point equations for computing these quantities (given in Section 2.1) are modified in a standard fashion.

We have used the Colorado University Decision Diagrams (CUDD) package [3] for computing the quantities $LCS_B$, $RCS_B$ as BDDs. However, none of our benchmarks suffered from an exponential space blow-up in the number of cache states appearing in $LCS$ or $RCS$. Therefore, the space gains obtained from the symbolic BDD representation (as compared to our enumerative set based representation) were modest in all our benchmarks. In future, we will conduct further experiments to evaluate the need for symbolic representation of cache states in our analysis technique.

## 6. ACKNOWLEDGMENTS

## 7. REFERENCES

[1] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[2] D. Burger, T. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Toolset. Technical Report CS-TR96-1308, Univ. of Wisconsin - Madison, 1996.

[3] CUDD. Colorado University Decision Diagram Package Version 2.3.1, 2001. Free software, http://vlsi.colorado.edu/~fabio/CUDD/.

[4] H. Dwyer and J. Fernando. Establishing a tight bound on task interference in embedded system instruction caches. In *International Conference on Compilers, Architecture and Synthesis of Embedded Systems (CASES)*, 2001.

[5] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *ACM Intl. Workshop on Languages, Compilers and Tools for Real-Time Sys. (LCTRTS)*, 1997.

[6] D.B. Kirk. SMART (strategic memory allocation for real-time) cache design. In *IEEE Real-Time Systems Symposium (RTSS)*, 1990.

[7] C-G. Lee et al. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6), 1998.

[8] C-G. Lee et al. Bounding cache-related preemption delay for real-time systems. *IEEE Transactions on Software Engineering*, 27(9), 2001.

[9] X. Li, T. Mitra, and A. Roychoudhury. Accurate timing analysis by modeling caches, speculation and their interaction. In *ACM Design Automation Conf. (DAC)*, 2003.

[10] Y. Li and W. Wolf. Hardware/Software Co-Synthesis with Memory Hierarchies. *IEEE Transaction on Computer-aided Design of Integrated Circuit and Systems*, 1999.

[11] Y-T. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems*, 4(3), 1999.

[12] Seoul National University Real-Time Research Groups. SNU real-time benchmarks. http://archi.snu.ac.kr/realtime/benchmark/.

[13] H. Tomiyama and N. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *ACM International Symposium on Hardware Software Codesign (CODES)*, 2000.

[14] F. Wolf, J. Staschulat, and R. Ernst. Associative caches in formal software timing analysis. In *ACM Design Automation Conf. (DAC)*, 2002.