Limited Preemptible Scheduling to Embrace Cache Memory in Real-Time Systems

Sheayun Lee[1], Chang-Gun Lee[1], Minsuk Lee[2], Sang Lyul Min[1], and Chong Sang Kim[1]

[1] Dept. of Computer Engineering, Seoul National University, Seoul 151-742, Korea
{sylee,cglee,symin,cskim}@archi.snu.ac.kr
[2] Dept. of Computer Engineering, Hansung University, Seoul 136-792, Korea
mslee@ice.hansung.ac.kr

Abstract. In multi-tasking real-time systems, inter-task cache interference due to preemptions degrades system performance and predictability, complicating system design and analysis. To address this problem, we propose a novel scheduling scheme, called LPS (Limited Preemptible Scheduling), that limits preemptions to predetermined points with small cache-related preemption costs. We also give an accompanying analysis method that determines the schedulability of a given task set under LPS. By limiting preemption points, the proposed LPS scheme reduces preemption costs and thus increases the system throughput. Experimental results show that LPS can increase schedulable utilization by more than 10 % and save processor time by up to 44 % as compared with a traditional fully preemptible scheduling scheme.

1 Introduction

Cache memory is used in almost all computer systems today to bridge the ever increasing speed gap between the processor and main memory. However, if cache memory is to be used in real-time systems, special attention must be paid since cache memory introduces unpredictability to the system. For example, in a multi-tasking real-time system, when a task is preempted by a higher priority task, the preempted task's memory blocks in the cache are replaced by the preempting higher priority task's memory blocks. Later, when the preempted task resumes its execution, a considerable amount of delay occurs to reload the previously replaced memory blocks into the cache. When preemptions are frequent, the sum of such cache reloading delays takes a significant portion of task execution time. Moreover, the portion gets even larger as the speed gap between the processor and main memory increases.

The cache reloading costs due to preemptions have largely been ignored in real-time scheduling. Without a suitable analysis method, we have to conservatively assume that each preemption causes one cache miss for each cache block used by the preempted task. This will result in a severe overestimation of the cache reloading time.

In this paper, we propose a novel scheduling scheme, called LPS (Limited Preemptible Scheduling), that allows preemptions only at predetermined execution points with small cache reloading costs. The selection of preemptible points is based on the number of *useful cache blocks* [6]. A useful cache block at an execution point contains a memory block that *may* be re-referenced before being replaced by another memory block of the same task. The number of useful cache blocks at a given execution point in a task can be calculated by using a data-flow analysis technique explained in [6] and it gives an upper bound on the number of cache blocks that need to be reloaded when preemption occurs at that execution point. By limiting preemptible points to those with a small number of useful cache blocks, the cache-related preemption delay can be significantly reduced.

Although limiting preemptible points can significantly reduce the cache-related preemption delay, it increases the *blocking time* suffered by higher priority tasks. For example, if the processor is executing a nonpreemptible code section of a lower priority task when a higher priority task arrives, the higher priority task cannot begin its execution until the lower priority task exits the nonpreemptible code section. This blocks the execution of the higher priority task and can potentially make it miss its deadline. Fortunately, such blocking delay is bounded and we give a method for analyzing the worst case blocking delay based on the *extended timing schema* approach [8]. The method determines the worst case blocking time of a lower priority task by estimating the WCET (worst case execution time) of the longest nonpreemptible code section.

For the proposed LPS scheduling scheme, we give an analysis method that can determine whether a given task set is schedulable. The schedulability analysis compares each task's deadline with its WCRT (worst case response time) that is computed by augmenting the response time equation explained in [4, 10] to take into account both the cache-related preemption delay and the blocking delay.

The reduction of the cache-related preemption delay by LPS reduces the WCRTs of lower priority tasks. However, the accompanying increase of the blocking delay increases the WCRTs of higher priority tasks. Thus, LPS can improve or degrade the schedulability of a task set depending on the characteristics of the task set. However, since lower priority tasks constrain the schedulability of the task set in many cases, the improvement of lower priority tasks' WCRT by LPS usually enhances the schedulability. Furthermore, assuming that a given task set is schedulable, LPS significantly reduces the cache reloading time and the resulting savings in processor time can be used to perform other useful jobs.

We performed a set of experiments to assess the impact of LPS on schedulability and system utilization. The results show that the LPS can improve a task set's schedulability when lower priority tasks constrain the schedulability of the whole task set. The results also show that the LPS can save processor time by up to 44 % when compared with a traditional fully preemptible scheduling scheme.

The rest of this paper is organized as follows: In Section 2, we survey the related work. Section 3 details the proposed LPS scheme and the accompanying schedulability analysis method. Section 4 gives the results from our experiments. We conclude this paper in Section 5.

2 Related Work

Although caches are used in almost all computer systems today, they have not been widely used in real-time systems due to their unpredictable worst case performance. The unpredictable performance results from two sources: intra-task interference and inter-task interference. Intra-task interference, which occurs when more than one memory block of the same task are mapped to the same cache block, has been extensively studied in [1–3, 7, 8]. In this paper, we focus on inter-task interference of caches caused by task preemptions.

There have been two approaches to address the unpredictability resulting from the inter-task cache interference. The first is to eliminate the inter-task cache interference by using a cache partitioning technique where cache memory is divided into mutually disjoint partitions and one or more partitions are dedicated to each task [5, 11]. Although cache partitioning eliminates cache interferences due to preemptions, it has a number of drawbacks. One drawback is that the size of the cache seen by each task is significantly reduced, since a task can access only its own partitions. Another drawback is that the technique requires modification of existing hardware and/or software.

The second approach to address the unpredictability resulting from inter-task cache interference is to take into account the effect of cache interference in schedulability analysis. In [6], Lee *et al.* propose such an analysis technique based on the worst case response time equation [4, 10]. In this technique, the original response time equation is augmented to include the cache-related preemption delay as follows (assuming that task τ_i has a higher priority than task τ_j if $i < j$):

$$R_i = C_i + \sum_{j=1}^{i-1} \lceil \frac{R_i}{T_j} \rceil C_j + PC_i(R_i) \ , \tag{1}$$

where R_i, C_i, and T_i denote the response time, the WCET, and the period of task τ_i, respectively. The cache-related preemption delay is included in the equation in the additional term $PC_i(R_i)$. The term $PC_i(R_i)$ is computed by an integer linear programming technique that takes as its input the worst case cache reloading cost of each task. The calculation of the worst case cache reloading cost considers only the useful cache blocks, which significantly improves the accuracy of the WCRT prediction [6].

The cache-related preemption delay takes a significant portion of a task's response time. Its impact becomes more significant as the cache refill time increases. Simonson shows in his PhD thesis [9] that cache misses caused by preemptions can be significantly reduced by limiting preemptions to predetermined points called *preferred preemption points*. The preferred preemption points are determined by analyzing a given task's execution trace and selecting points that incur small preemption costs. One limitation of this approach is that the preferred preemption points cannot be determined from the program code since the approach is based on trace driven analysis. Thus, the approach cannot be applied during the design phase of a real-time system. Furthermore, the approach does not offer any schedulability analysis technique.

In this paper, we propose a technique that can determine preemption points with small cache reloading costs from the program code. We also give a schedulability analysis technique that considers not only the cache-related preemption delay but also the blocking delay resulting from limiting preemptible points.

3 Limited Preemptible Scheduling

In this section, we first explain how to determine preemptible points with small cache reloading costs. Then, we explain the technique to calculate the blocking time caused by limiting preemptible points. Finally, we explain the schedulability analysis that takes into account the cache-related preemption delay and the blocking delay.

3.1 Determining Preemptible Points

Since the number of useful cache blocks at an execution point gives an upper bound on the cache reloading cost at that point, a preemption at an execution point with a small number of useful cache blocks incurs a low preemption cost.

We divide the set of execution points of a given task into *preemptible execution points* and *nonpreemptible execution points* depending on the number of useful cache blocks. Specifically, if an execution point has more than M useful cache blocks, it is a nonpreemptible execution point; otherwise, it is a preemptible execution point. Here, M is a threshold value that controls the upper bound on the cache reloading cost.

When a task is executing within a nonpreemptible code section, preemption is not allowed even when there is a pending higher priority task. One possible way to implement the preemption control is to insert extra instructions to the execution points that correspond to entry points or exit points of nonpreemptible code sections. The inserted instructions modify a boolean variable that indicates whether the execution of the program is within a nonpreemptible code section or not. When the scheduler is invoked by an arrival of a task (in event-driven scheduling) or by a clock tick (in tick scheduling), the boolean variable is checked by the scheduler to determine whether it should perform a context switch.

3.2 Bounding Blocking Time

As we mentioned earlier, limiting preemptible code sections causes blocking of high priority tasks that arrive while a lower priority task is executing within a nonpreemptible code section. We call the amount of time that a lower priority task blocks a higher priority task's execution the *blocking time* of the lower priority task. To guarantee the worst case performance of the system, we need to bound the blocking time in the worst case, which we call the WCBT (worst case blocking time).

The tighter the WCBT bounds of tasks, the more accurate the prediction of the worst case performance of the system. In the following, we explain a technique that computes a tight bound of the WCBT based on the *extended timing schema* [8], which was originally proposed to compute the WCET of a program.

In the extended timing schema, the syntax tree of the given program is hierarchically analyzed, recursively applying a set of timing formulas to each program construct. The timing formulas are defined with two types of basic operations on the timing abstraction called PA (path abstraction); the concatenation ($\oplus$) operation models the sequential execution of two execution paths, and the set union ($\cup$) operation reflects the possibility of more than one execution path in a program construct.

To calculate the WCBT of a given task from its program code, we associate a data structure called WCBTA (worst case blocking time abstraction) with each program construct. The WCBTA maintains timing information of *blocking paths* that *might* have the largest execution time in the corresponding program construct. A blocking path is a partial execution path that consists only of nonpreemptible code sections, and thus preemption is not allowed when the execution of the program is on such a path. Since it is not possible to determine which blocking path in a program construct will give the largest execution time until the preceding/succeeding program constructs are analyzed, the WCBTA needs to maintain timing information for more than one blocking path. Thus, the WCBTA of a program construct has a set of abstractions, called BPAs (blocking path abstractions), for the blocking paths in the program construct. In addition to the timing information maintained in a PA of the extended timing schema, each BPA maintains information about whether the entry point and/or exit point of the corresponding path is preemptible or not. This information is needed when the concatenation operation is performed between two BPAs to determine whether the two paths lead to a longer blocking path.

When the hierarchical analysis reaches the top level, the WCETs of all the blocking paths in the program are calculated, among which the maximum value is chosen as the program's WCBT.

3.3 Schedulability Analysis

The schedulability analysis for the LPS scheme is based on the response time equation [4, 10]. To take into account the cache-related preemption delay and the blocking delay, the response time equation is augmented as follows:

$$R_i = C_i + \sum_{j=1}^{i-1} \lceil \frac{R_i}{T_j} \rceil C_j + PC_i(R_i) + B_i \ , \tag{2}$$

where R_i, C_i, and T_i denote the response time, the WCET, and the period of τ_i, respectively. The augmented response time equation includes both the cache-related preemption delay $PC_i(R_i)$ and the blocking delay B_i. The cache-related preemption delay is estimated using Lee *et al.*'s linear programming method. Since smaller preemption costs are used in LPS, the cache-related preemption delay is smaller than in a fully preemptible scheduling scheme.

The blocking delay B_i is the amount of time that task τ_i is blocked by a lower priority task in the worst case. The worst case is when task τ_i arrives immediately after the lower priority task with the largest WCBT begins executing its longest blocking path. Therefore, the worst case blocking delay B_i is equal to the WCBT of the task with the largest WCBT among the lower priority tasks. This is given by

$$B_i = \max_{j>i}(Z_j) \ , \tag{3}$$

where Z_j is the WCBT of task τ_j.

In estimating the WCRTs of the lower priority tasks, the cache-related preemption delay is significantly reduced since it includes the delay caused by preemptions of not only the task itself but also all the higher priority tasks. Therefore, in general, when the lower priority tasks in a given task set have relatively tight deadlines, LPS enhances the schedulability of the task set. On the other hand, LPS increases the blocking time of the higher priority tasks. In the case where the higher priority tasks have tight deadlines, the increased blocking delay may degrade the schedulability of the system. In other words, whether LPS leads to better schedulability depends on the characteristics of the task set. However, once LPS guarantees the schedulability of a given task set, a significant amount of processor time is saved due to reduced cache reloading costs. This saved processor time can be used for other useful jobs in the system, achieving a higher system throughput.

4 Experimental Results

To assess the effectiveness of the proposed LPS scheme, we performed three kinds of experiments. First, the WCBTs of several benchmark programs are analyzed, and the results are presented in Section 4.1. Second, experimental results showing the impact of the LPS scheme on schedulability are given in Sections 4.2 and 4.3. Finally, we present in Section 4.4 results that show how much processor time can be saved by LPS.

4.1 Per-Task Analysis for WCBTs

Our WCBT analysis assumes the MIPS R3000 RISC processor as the target machine. The processor has a direct mapped cache of 16 KB with a block size of 4 bytes. The cache refill time is assumed to be 16 machine cycles.

We set the M value (the maximum allowable number of useful cache blocks at preemptible execution points) of all the tasks to zero. In other words, preemption is allowed only at execution points with zero cache reloading costs.

Five simple benchmark programs were chosen: *matmul*, *jfdctint*, *fft*, *ludcmp*, and *fir*. The *matmul* benchmark performs multiplication on two 5×5 floating-point matrices. The *jfdctint* benchmark implements an integer discrete cosine transformation for the JPEG algorithm. The *fft* benchmark performs the FFT and inverse FFT operations on an array of 10 floating-point numbers, and *ludcmp* solves 10 simultaneous linear equations by the LU decomposition method. Finally, *fir* implements the FIR (Finite Impulse Response) filter.

Table 1. WCBTs of five benchmarks

Name	WCET	WCBT	$\frac{WCBT}{WCET}$
matmul	10795	10044	93.0 %
jfdctint	11932	3964	33.2 %
fft	24698	22647	91.7 %
ludcmp	37009	27133	73.3 %
fir	71298	71201	99.9 %

(unit: cycles)

The WCBTs and WCETs of the five benchmarks are shown in Table 1. The ratio WCBT/WCET varies depending on the characteristics of the benchmark program. It is affected mainly by the structure of loop statements in the program. Specifically, the *fir* benchmark consists of several nested loops and the outermost loop occupies most of the program code. In such a case, all the execution points inside the outermost loop have the same maximum number of useful cache blocks, and the whole loop becomes a single nonpreemptible code section. The *fir* program spends most of its execution time in this loop and, thus, its WCBT is very close to the WCET of the benchmark. On the other hand, in the *jfdctint* benchmark, there are three outermost loops that have nearly equal execution times. Thus, the WCBT is about 1/3 of the WCET.

4.2 Schedulability: A Good Case

We predicted the WCRTs of the tasks in a sample task set to see how LPS affects the schedulability of the task set as a whole. For this experiment, we used the task set given in Table 2. In the table, the frequency of a task is the number of invocations of the task within the system period (hyperperiod) of the task set.

The workload factor W of the task set is calculated by

$$W = \frac{\sum F_i \cdot C_i}{system\ period}\ , \tag{4}$$

Table 2. Task Set 1

Task	Name	Frequency
τ_1	matmul	32
τ_2	jfdctint	24
τ_3	fft	18
τ_4	ludcmp	12
τ_5	fir	9

where F_i denotes the number of invocations of τ_i in the system period and C_i the WCET of τ_i. The workload factor gives the system's pure workload that excludes the time for cache reloading due to preemptions. We performed schedulability analysis as we increased the workload factor by gradually decreasing the system period. The deadline of each task, which is assumed to be equal to the period of the task, was adjusted accordingly as we decreased the system period.

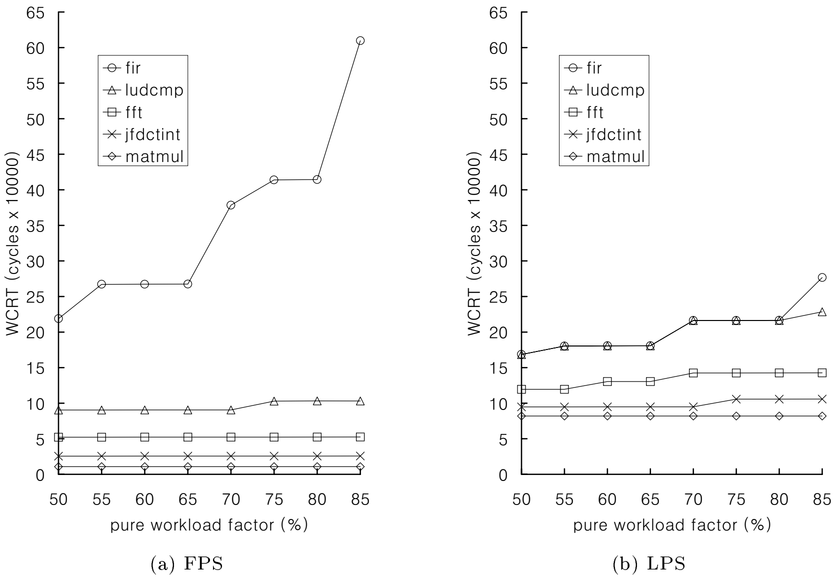


Fig. 1. WCRT predictions for Task Set 1

Figure 1 depicts the WCRT predictions for both the traditional fully preemptible scheduling (denoted by FPS) and LPS. The x-axis of the graphs is the system's workload factor W whereas the y-axis is the WCRTs of tasks (in machine cycles).

Figure 1(a) shows the results when FPS is used. In FPS, the WCRT of the highest priority task τ_1 (*matmul*) is equal to its WCET since it begins its execution immediately after release and is never preempted. The WCRTs of the intermediate priority tasks (i.e., τ_2, τ_3, and τ_4) do not increase much even when the workload factor is as high as 85 %. However, the WCRT of the lowest priority task τ_5 (*fir*) increases rapidly as the workload factor increases. This results from a rapid increase of the cache-related preemption delay that includes the cache reloading costs of not only itself but also all the higher priority tasks.

Figure 1(b) shows the results when LPS is used. As compared with FPS, the WCRTs of $\tau_1, \cdots, \tau_4$ become slightly larger because the WCRTs now include the blocking delay, which is equal to the WCBT of τ_5 (*fir*). However, the WCRT of the lowest priority task *fir* becomes much smaller than in the FPS case and does not increase as rapidly. This results from the fact that the cache-related preemption delay is zero in the LPS case since preemptions can occur only at execution points with no useful cache blocks.

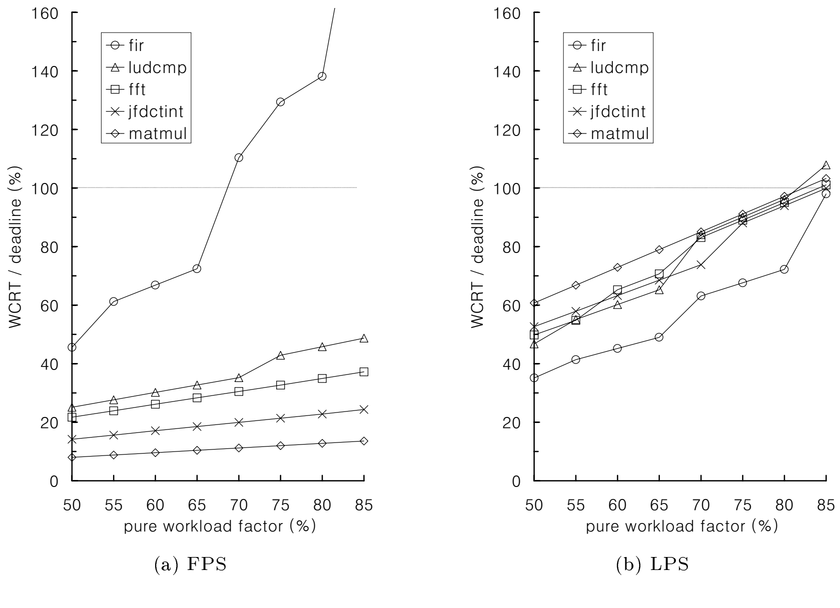


Fig. 2. WCRT/deadline for Task Set 1

Figure 2 depicts the ratio of the WCRT to the deadline assuming that each task's deadline is equal to its period. The results show that in the FPS case the lowest priority task *fir* is the one that limits the schedulability of the whole task set. Thus, the reduction of the WCRT of *fir* in the LPS case by reducing the cache-related preemption delay proves to be helpful in improving the schedulability of the given task set as we can see in Figure 2(b). The graphs show that the breakdown utilization of the task set is about 81 % for LPS, which is more than 10 % higher than the FPS's 68 %.

4.3 Schedulability: A Bad Case

We performed another experiment that was intended to show potential problems of LPS. Table 3 gives the sample task set used for this purpose. The task set consists of four tasks: *matmul*, *jfdctint*, *fft*, and *ludcmp*. For this task set, the ratio of the frequency of the highest priority task *matmul* to those of the other tasks is much higher than in the previous task set. This gives a much tighter deadline to the highest priority task as compared with the deadlines of the other tasks.

Table 3. Task Set 2

Task	Name	Frequency
τ_1	matmul	20
τ_2	jfdctint	5
τ_3	fft	4
τ_4	ludcmp	2

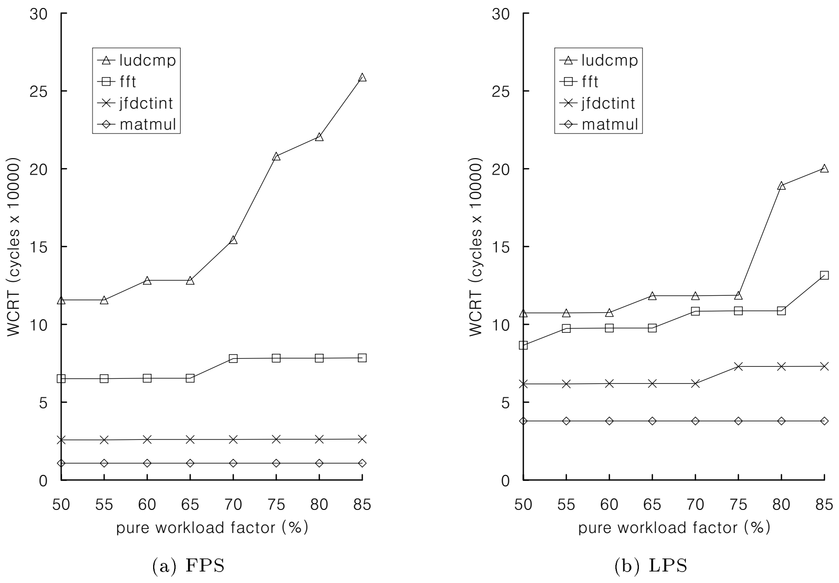
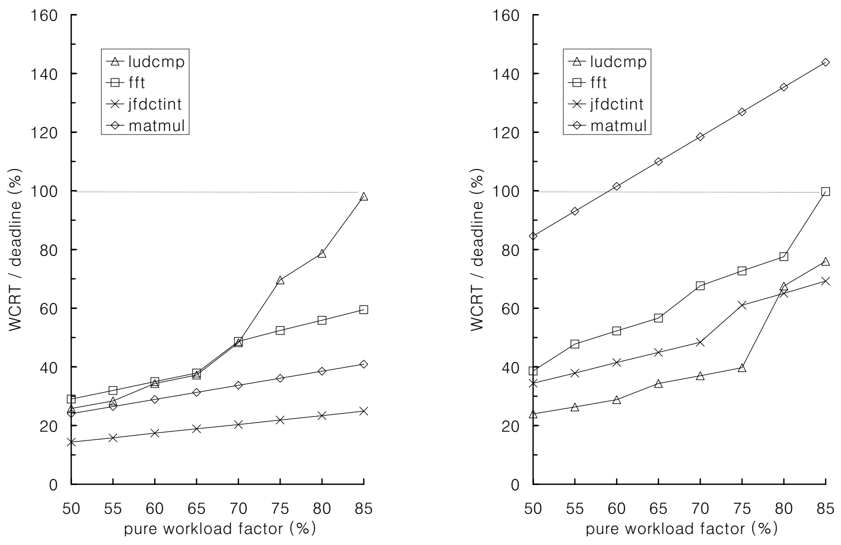


Fig. 3. WCRT predictions for Task Set 2



(a) FPS (b) LPS

Fig. 4. WCRT/deadline for Task Set 2

Figures 3 and 4 show the WCRT and the ratio of the WCRT to the deadline, respectively, for the new task set. LPS now fails to schedule the task set when the workload factor is about 59 % whereas FPS can still schedule the task set even when the workload factor is over 85 %. This results from the fact that the highest priority task has a much tighter deadline than the other tasks and, thus, even a small amount of blocking delay can make it miss the deadline.

The blocking delay can be reduced by transforming the source code of tasks. For example, a task's WCBT can be reduced by unrolling loops in the program of the task. In this case, a single loop is transformed into a number of equivalent loops. This helps reduce the WCBT of the task since most of the useful cache blocks are due to loop statements. This transformation places preemptible execution points between the resulting unrolled loops, thus reducing the WCBT. If a loop is transformed into n loops, the WCBT due to the loop becomes approximately $1/n$ of the WCBT of the loop before the loop unrolling.

However, the loop unrolling decreases temporal locality of instruction references and increases the code size. Thus, this technique trades increased WCET (due to increased cache misses) and code size for reduction in the WCBT.

Loop unrolling was applied to the tasks *fft* and *ludcmp*. After the loop unrolling, the ratios of the WCBT to the WCET of the two tasks are reduced from 91.7 % to 19.1 % and from 73.3 % to 6.3 %, respectively. Figures 5(a) and (b) give the WCRT and the ratio of the WCRT to the deadline of each task, respectively, for the LPS case after the loop unrolling is applied. As expected, the WCRT of higher priority tasks (most notably the highest priority task *matmul*)

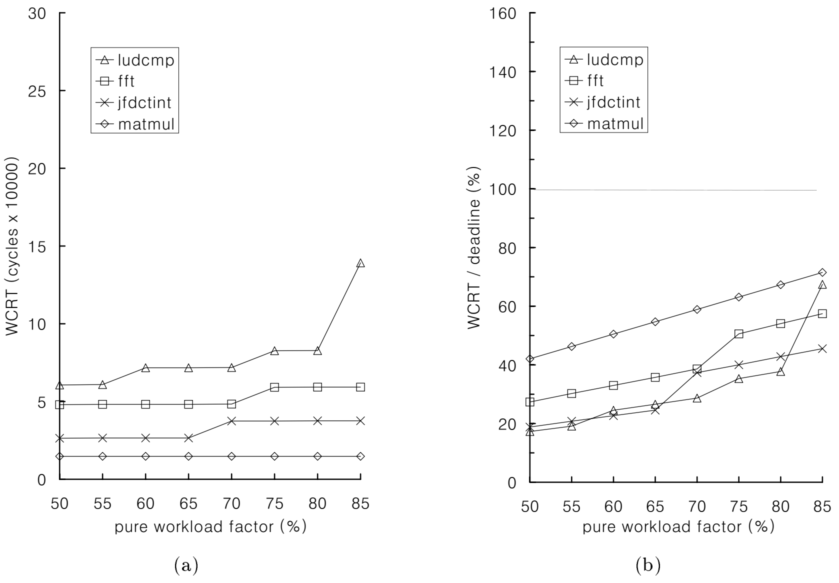


Fig. 5. Results of source code transformation

is reduced significantly due to the reduction of the WCBTs of the *fft* and *ludcmp* tasks. This improves the schedulability of the whole task set.

4.4 System Utilization

Figure 6 gives the processor time gain obtained by using LPS for the two previous task sets. The x-axis is the cache refill time, and the y-axis is the percentage of processor time saved by LPS. The results were obtained by averaging the processor time over 100 different simulation runs with random phasing between tasks. The results show that as the cache refill time increases, the percentage of the saved processor time increases too (up to 44 %). The percentage gain is greater for Task Set 1 than for Task Set 2 since the number of preemptions in Task Set 1 is larger than that in Task Set 2 and so are the savings.

5 Conclusion

We have proposed a novel scheduling scheme called LPS (Limited Preemptible Scheduling) that allows preemptions only at execution points with small cache reloading costs. We have also given a schedulability analysis technique that considers not only the reduction of the cache-related preemption delay by LPS but also the increase in the blocking delay resulting from nonpreemptible code sections. The worst case blocking delay resulting from nonpreemptible code sections

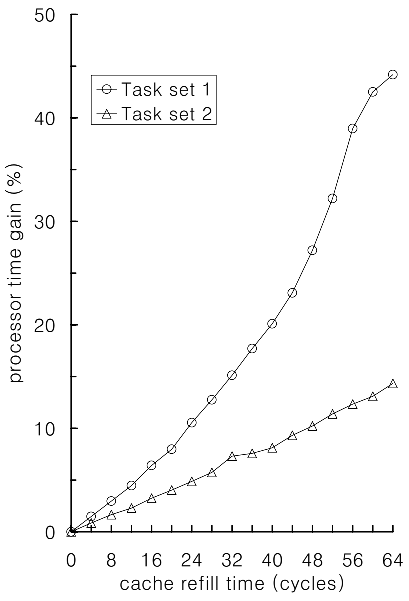


Fig. 6. Processor time gain

of a lower priority task was estimated by augmenting a method for estimating the worst case execution time explained in [8].

We assessed the impact of the LPS scheme on schedulability and system utilization through a set of experiments. The results showed that LPS can enhance the schedulability of a task set when the task set suffers from tight deadlines for lower priority tasks. The results also show that LPS saves a significant amount of processor time (by up to 44 %) and that the percentage of saved processor time increases as the cache refill time increases.

We are currently working on an optimization technique that assigns M values to tasks in such a way that maximizes the schedulability of the given task set. This technique uses information about how the WCBT of a task changes as the M value increases, which is obtained by a per-task analysis. The technique identifies the task that limits the schedulability of the task set and changes the M values of tasks accordingly.

References

1. R. Arnold, F. Mueller, D. B. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *Proceedings of the 15th Real-Time Systems Symposium*, pages 172–181, Dec. 1994.
2. C. A. Healy, D. B. Whalley, and M. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of the 16th Real-Time Systems Symposium*, pages 288–297, Dec. 1994.

3. Y. Hur, Y. H. Bae, S.-S. Lim, S.-K. Kim, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim. Worst case timing analysis of RISC processors: R3000/R3010 case study. In *Proceedings of the 16th Real-Time Systems Symposium*, pages 308–321, Dec. 1995.
4. M. Joseph and P. Pandya. Finding response times in a real-time system. *The BCS Computer Journal*, 29(5):390–395, Oct. 1986.
5. D. B. Kirk. SMART (strategic memory allocation for real-time) cache design. In *Proceedings of the 10th Real-Time Systems Symposium*, pages 229–237, Dec. 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. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *Proceedings of the Seventeenth Real-Time Systems Symposium*, pages 264–274, Dec. 1996.
7. Y. T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th Real-Time Systems Symposium*, pages 298–307, Dec. 1995.
8. S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C. S. Kim. An accurate worst case timing analysis for RISC processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, Jul. 1995.
9. J. Simonson. *Cache Memory Management in Real-Time Systems.* PhD thesis, University of Illinois at Urbana-Champaign, Sep. 1996.
10. K. Tindell, A. Burns, and A. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *The Journal of Real-Time Systems*, 6(2):133–151, Mar. 1994.
11. A. Wolfe. Software-based cache partitioning for real-time applications. In *Proceedings of the 3rd International Workshop on Responsive Computer Systems*, Sep. 1993.