Use of Preferred Preemption Points in Cache-Based Real-Time Systems

Jonathan Simonson and Janak H. Patel

Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801-2307
USA

Abstract

Time-critical applications require known worst-case execution times to ensure that system timing constraints are met. Traditional cache memory arrangements, however, significantly impede the determination of tight upper bounds on these worst-case execution times (WCET). The difficulty comes in adequately predicting the cache miss ratio for a task in a preemptable multi-tasking environment. Caches thus increase the complexity of calculating WCET. To resolve this, caches have simply been excluded from WCET calculations. Each task must then be provided greater time in which to execute leading to lower throughput and performance. In this paper we present a cache management scheme that allows WCET calculations to more easily reflect the timing effects of caching. This is done through the appropriate selection of preemption points within a task's execution. The scheme focuses on the WCET component that is due to preemption overhead. An added benefit is a reduction in execution time of up to 10% for some tasks over traditional cache management.

1: Introduction

Cache memory is essential to real-time systems in providing the high throughput and performance needed by time-critical applications. The critical nature of the applications, however, demands worst-case execution times (WCET) to ensure that all timing requirements are met. It is therefore important that such systems be able to use cache memory while having calculable WCETs that account for the caching effects.

WCETs are essential for determining the schedulability of a task set. Therefore, they must be guaranteed to ensure system reliability. These guarantees are frequently made by allocating much more time than ever actually used. This results in lower system utilization due to idle processor cycles. Some systems use these cycles for non-critical tasks. Alternatively greater attention can be given to the analysis of the system and the task set to obtain tighter bounds on the WCETs. Imprecise task results may be permissible when even shorter execution times are needed [1]. To obtain tighter execution bounds, however, a task must be examined to determine its longest path of execution. Also under examination must be the system resources required by the task and the time consumed by them. Here we look at the effects of the cache memory on these calculations and how the calculations can be made more tractable through the scheme that we present.

The difficulty that caches add to the calculation of WCET comes in determining the portion of memory references that are cache hits. The absolute upper bound on execution time when considering the cache is obtained using a hit ratio of zero, but this essentially negates the use of the cache. To get closer to the lowest upper bound, the system and the task set must be analyzed. One of the factors that contributes to the difficulty of this analysis is the variations in program flow within each task. Each execution path may have a very distinct memory access behavior resulting in very different cache hit rates. Preemptable multi-tasking environments also can add greatly to the variability in cache hit rates and thus to the variability of program execution times. Such environments result in significant reductions in tasks' cached state by the time the tasks resume execution. This is a consequence of task-state replacement by intervening processes. The degree of task-state reduction may be very difficult to determine due to variations in task execution and sequencing. All this can make the problem of determining cache hit rates intractable.

One of the first solutions that arose to resolve this problem was to simply exclude the cache from the timing calculations or even the actual system. As mentioned previously, this degrades performance with regard to throughput but does provide for more tractable timing calculations and thus better guarantees on WCETs. Kirk and Strosnider [2, 3, 4] approached the problem by partitioning the cache into segments that are then allocated to a set of tasks to

This research was supported in part by a grant from the National Aeronautics and Space Administration under contract NASA NAG-1-613.

0-8186-7059-2/95 $04.00

meet the system requirements. Hardware modifications to the cache design are necessary in their technique to maintain the ownership and activity of these segments. A drawback of their technique is that each task is limited in the amount of cache space it can use. This is mostly compensated for by their segment allocation scheme. A more software-oriented approach was taken by Niehaus, Nahum, and Stankovic [5]. Their work focuses on calculating WCETs, taking into account the effects of instruction caching. The tasks are divided up at large subtask boundaries to provide scheduling points at which task switching can take place. A similar approach was taken by Arnold *et al.* [6]. Here again large subtask are examined by careful program flow analysis. A method referred to as static cache simulation is used to determine when a particular assembly instruction is going to be hit or miss. Of the six applications that they ran using their method, none underestimated the WCET. For five of the applications, the estimated WCETs were within 12% of the actual. The sixth application was overestimated by 99%. Liu and Lee [7] also address WCET for uninterrupted cached programs. Instruction caching is accounted for at a basic block granularity. Their work focuses more on the complexity of determining the worst-case control flow path. To reduce the complexity cost they present various approaches that stop short of exact analysis. An exact analysis is also provided. They were able to estimate the WCET for a binary search programto within 10% of the actual. Basumalick and Nilsen [8] have provided a short overview of additional techniques to predicting cache hit rates in real-time systems.

This paper focuses on the component of WCET calculations due to task switching. Preemptions often result in cache state changes between periods of program execution. This makes it difficult to determine the degree to which the cache reload overhead affects execution time. We propose a scheme that both reduces the observed cache reload overhead and permits determination of the execution time cost of preemptions. This allows one to more accurately calculate WCET in a preemptable environment. This scheme is also known to reduce task execution time by up to 10% through selection of preemption points that result in low cache reload times.

In the remainder of the paper we present our cache management scheme. Its aim is to provide for more predictable WCETs and reduce task execution time in a preemptable multi-tasking environment. We begin by presenting our scheme and its proposed implementation in Section 2. Section 3 provides an explanation of how the scheme was evaluated. The results and their interpretation follow in Section 4. In Section 5 we conclude the paper with an overview and future directions of this work.

2: Guided preemption via cached state

In an environment where tasks vie for processor cycles and preempt one another in obtaining those cycles, a task's cached working-set can be significantly reduced by the time the task regains control of the processor. This results in a cache-reload overhead when the task resumes. The extent of the cache state reduction and what state is lost can be difficult to determine. This is affected by factors such as point of preemption occurrence, preemption frequency, and task sequencing. These factors are often loosely defined and vary from one execution of a task to the next. The state loss that occurs is actually a replacement of the preempted task's cached working-set by intervening tasks. The traditional cache is one that is a shared resource and one in which the currently executing task has free range over the use of the cache lines. The following scheme was devised with the above ideas in mind.

The technique that is proposed limits preemptions to prespecified points within a task's execution. Firstly, this allows for more accurate worst-case calculations of the cost of preemptions since the possible points of preemptions are well defined. Secondly it allows for the reduction of preemption costs through appropriate preemption point selections. The second point clearly benefits any system not just one requiring guarantees on WCETs.

2.1: Technique basis

The basis of our technique lies in calculating the cost of a preemption. When a task is preempted there is the cost of context switching as well as the additional cost of restoring to the cache any active task state that has been lost when the task resumes. It is this additional cost that is the focus of our scheme. The context switching time is often a direct function of the degree of processor state that is saved and restored.

At the time a task is preempted, there exists some portion of the task's cached state that would have been accessed in the future prior to replacement by a cache miss. The cache lines that fit this description will be referred to as *live* or *active* cache lines and those that don't as *dead* or *non-active* cache lines. The definitions for these terms are as follows:

Definition 1: A *Live* or *Active Cache Line* refers to one that contains a block of data that will be referenced in the future prior to its replacement. In other words, it is a cache line in which the next reference is a hit had the task been allowed to run to completion.

Definition 2: A *Dead or Non-Active Cache Line* refers to one that contains a block of data that will be replaced prior to any future reference or that will not be referenced

during the remainder of a task's execution. In other words, it is a cache line in which the next reference is a miss or for which there are no future references had the task been permitted to run to completion.

If the active cache lines are replaced due to the execution of other tasks, the data they contain must be fetched from main memory when referenced by the resuming task. This adds to the execution time of the task over what would have been necessary had the task not been preempted. This added execution time is equal to the actual number of active cache lines replaced times the cache miss penalty (ie. the additional time necessary to fetch the data from main memory). Since the determination of the number of active cache lines is a much more tractable problem than the determination of the number of active cache lines replaced, the remainder of this paper will use an upper bound on the cache reload overhead equal to the number of active cache lines times the cache miss penalty. How the number of active cache lines might be calculated or estimated is described in Section 2.2.

Once the number of active cache lines are determined at each instant within a task's execution, the points of preemption can be chosen to decrease preemption costs. These points will be referred to as *preferred preemption points* and are defined as follows and illustrated in Figure 1.

Definition 3: A *Preferred Preemption Point* for a given interval of task execution t_i to t_j is the instant within that interval having the minimum number of live cache lines for the interval.

Only one preemption point is needed per task switching if it is known when in each task's execution a higher priority task will be ready to run. On the other hand, preemption points are necessary at frequent intervals and possibly

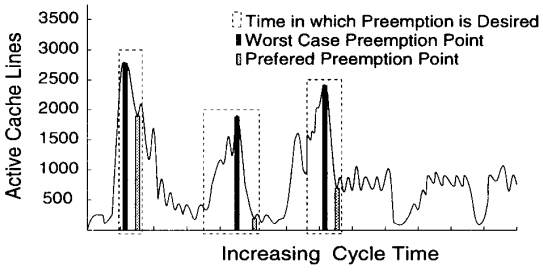


Figure 1: Preferred preemption points

throughout a task's execution in systems that exhibit a high degree of variability and where higher priority tasks may need to gain control of the processor quickly. In either case, by knowing the maximum number of preemptions for each individual task and the maximum cost of those preemptions, a upper bound on the added time due to the effects of preemptions on caching can be determined. This can be expressed as given below for the highly variable case. The equation does not account for other system effects of preemptions, such as a possible increase in paging and possible contention for system resources (eg. secondary storage devices). The overhead due to preferred preemptions above that without preemptions is then

$$E_P = (max(p_{l0}, \cdots, p_{ln-1})\ t_m + t_c)\ P \qquad (1)$$

where

E_P = *Time due to preferred preemption points above that necessary without preemptions*
p_{li} = *Number of live lines at preferred preemption point i*
n = *Number of prespecified preferred preemption points*
t_m = *Cache miss penalty*
t_c = *Context switch time*
P = *Maximum number of preemptions*

If the number of live lines per preemption point is known and the exact points of preemptions are known, E_p becomes

$$E_P = Pt_c + t_m \sum_R p_{li} \qquad (2)$$

where

R = *Actual points at which preemptions occur*

2.2: Determining number of active cache lines

In the previous section it was shown that the knowledge of the number of active cache lines during a task's execution permits better worst-case execution calculations. To make use of this, however, there must be some method of determining or predicting the number of active cache lines. This section presents a possible method of addressing this issue. The method assumes a data cache but can be similarly performed for a instruction cache.

Since the active cache line behavior is linked to the activity of program variables, it is natural to look here for a solution. As a starting point it will be assumed that the

task control flow paths are available in order of decreasing execution time for a non-preempting system with cache memory. From here the activity of the program variables are determined and correlated or translated to active cache lines. To determine this activity, the life times of the program variables need to be ascertained. This is similar to performing live variable analysis for register allocation but here cache lines are the resource allocated. Dynamically addressed variables may require profiling or programmer assistance to determine their life times. The life times of a variable are the times for which the variable is resident and active within the cache. For non-fully associative caches the secondary effects of mapping conflicts may need to be considered but some preliminary simulation results show that this effect should be small. The analysis of variable activity and their mapping to cache lines is performed on a per control path basis. The end result is the cache line activity along each path.

The above analysis is first applied to the worst-case execution path and the preferred preemption points are determined along this path based on cache line activity. The WCET with preferred preemption points is then calculated. The next worst-case execution path is then examined. At this point to reduce calculation costs, the cost due to all preemptions along this new path is first calculated using a cache line activity equal to the number of cache lines. If the execution time with this upper bound on preemption costs does not exceed the first worst-case execution path with preferred preemption points, further analysis of this path and others are unnecessary. If it does, then the same analysis is necessary on this path as on the worst-case execution path. If this path then exceeds the worst-case execution path, it becomes the case of comparison for the remaining paths. The analysis then proceeds to the next worst-case execution path and the above steps are repeated comparing this path to the worst-case execution path of comparison. It is likely that since execution paths share segments of code, portions of the analysis done for one path can be shared with that of other paths having overlapping segments. This leads to further reduction in calculation costs. To reduce the memory cost necessary for such compilations, it may be possible to use a smaller cache for the actual calculations and scale the results.

The steps necessary in determining the WCETs so that they account for the effects of caching are best incorporated into the compiler. This enables them to be transparent to the user. Further work is in progress in this area with respect to detail and implementation, but it is felt that a reasonable approach to the problem has been found.

2.3: Preferred preemption point selection

For best results preferred preemption points should be determined as indicated in the previous section. More relaxed techniques can be used to determine the selection of preferred preemption points for tasks where lower guarantees on deadlines are allowed or where tighter upper bounds on WCETs are unnecessary. This also applies to task that can afford to have longer execution times.

Perhaps one of the easiest techniques of selecting preemption points under these conditions is by the insertion of traps within the source code by the programer to check for ready to run tasks of higher priority. Since the programer will often have at least an intuitive feeling as to where in the code there are likely to be significant changes in the working set and consequently reductions in number of live caches lines, performance improvements will result. A slightly more complicated technique, along similar lines, is to select subroutine boundaries as the preemption points. Often at such boundaries the working-set is noted to go through significant changes. With either method if tighter bounds on WCETs are desired, further analysis would be necessary to determine the cost of the preemption points chosen. The frequency of these points would also need to be checked to determine if they met the needs of the system.

One technique, that provides perhaps the greatest performance improvements but which is probably the most computationally intensive, is to determine preferred preemption points for each control path through a task. This would be done almost identically to how preemption points were selected in the previous section, with the exception that no comparison of execution times would be necessary between paths unless tighter WCETs were desired.

The technique chosen depends on the system requirements. Take for instance a task which must use imprecise results because it can not meet its deadline but whose results are crucial to the system. Such a task would require a thorough analysis to find preferred preemption points along all program control flow paths. Systems are likely to use multiple techniques to obtain the greatest benefits.

2.4: Scheduling with preemption points

Once the preemption points are chosen the scheduling algorithm of choice must be adjusted to incorporate them. Task sets previously unschedulable may now through their reduced execution time meet the timing requirements. What follows is a modification to the rate monotone scheduling algorithm [9] that accounts for predetermined preemptions points.

The rate monotone scheduling algorithm has been modified to first allow tasks to preempt each other only at prespecified preemption points and secondly to provide the maximum time distance between preemptions such that all tasks will still meet their deadlines. The steps to the algorithm are as follows:

Step 1: Ensure the schedulability of tasks under the unmodified rate monotone scheduling algorithm without accounting for the cost of preemptions.

Step 2: Determine the maximum allowable preemption spacing for each task:

a: Form groups of tasks consisting of each task and the tasks of higher priority.

b: Beginning with the smallest group perform the following for each group. Apply the rate monotone schedule to the task group, but allow the lowest priority task to begin first at the critical instant (ie. a point when all tasks are ready to run). Determine the maximum time this task may execute such that all higher priority tasks in the group meet their deadlines. Already determine maximum preemption intervals should be used to define the preemption points of the higher priority tasks. Preemptions are allowed only at these points for this step of the proceedure.

Step 3: Find preferred preemption points for each task such that their maximum allowable preemption interval is not exceeded.

Step 4: Schedule tasks using the rate monotone scheduling algorithm but allow preemptions only at prespecified preemption points with the time cost for such points accounted for.

Step 1 can be determined using the work of Lehoczky, Sha, and Ding [10] for task set schedulability under the rate monotonic scheduling algorithm. The justification for Step 2 is as follows. It is possible for a lower priority task to begin execution one time unit before its higher priority tasks become ready to run. The higher priority tasks would then need to wait until this lower priority task reached a preemption point before they could begin execution. This preemption point must come in a sufficient amount of time such that all the higher priority tasks meet their deadlines. By allowing the lowest priority task to begin execution first when all higher priority tasks are ready to run, this situation is reproduced and the maximum time of execution without preemption can be determined. Step 2 was found to be very similar to work done in [11] for the schedulability of a task set using the rate-monotonic algorithm under the priority ceiling protocol. The blocking that occurs with preferred preemption points has a scheduling effect similar to the blocking that occurs with priority inheritance or semaphores. A theorem is provided in [11] for determining task schedulability under blocking.

An illustrative example of the modified rate monotone algorithm given here is depicted in Figures 2 through 5. In Figure 2, it is seen that all tasks within the task set meet their deadlines under the unmodified rate monotone scheduling algorithm. The execution times given are

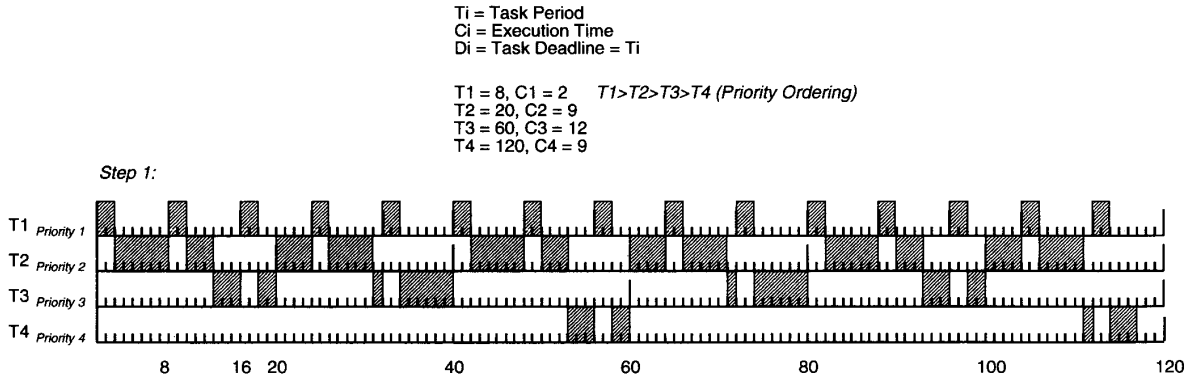


Figure 2: Unmodified rate monotone schedule w/o preemption costs

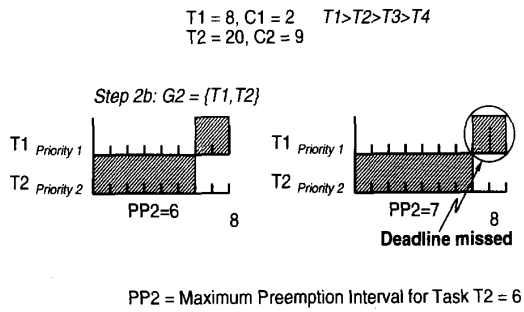


Figure 3: Step 2 of modified rate monotone scheduling algorithm for task T2

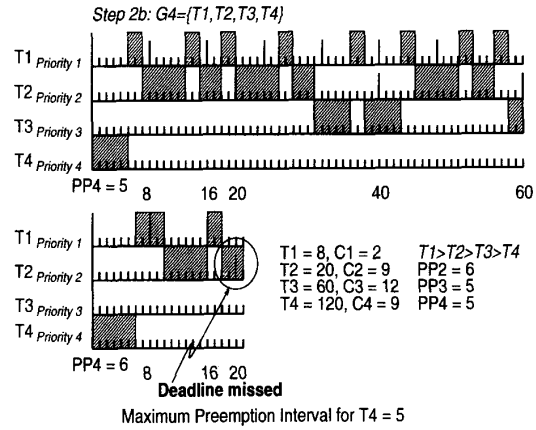


Figure 4: Step 2 of modified rate monotone scheduling algorithm for task T4

WCETs that account for caching but not for preemptions. This figure corresponds to Step 1. It can be seen that there are 15 preemptions and three remaining time units. The total cost of preemptions can not exceed three time units if the tasks are to meet their deadlines. By using preferred preemption points the chances of meeting the deadlines are increased.

Step 2 results in the following groups G1={T1}, G2={T1,T2}, G3={T1,T2,T3}, G4={T1,T2,T3,T4}. The maximum preemption interval for task T1 is always equal to its execution time. Figures 3 and 4 show the determination of the maximum preemption intervals for task T2 and T4 respectively. The maximum preemption interval for T3 can be done similarly. For task T2, it is seen from Figure 3 that if the preemption interval exceeds 6 time units, T1 misses its deadline. Therefore the maximum preemption interval for task T2 is 6. For task T4 in Figure 4, a preemption interval exceeding 5 time units causes T2 to miss its deadline. It should be noted that when T4 reaches one of its preemption points the ready to run task of highest priority above T4 replaces T4 as the executing process. This coincides with the unmodified rate monotone scheduling algorithm with the exception that tasks now preempt only at preemption points. In determining the maximum preemption intervals for lower priority tasks, the preemption points for higher priority tasks are set at a spacing equal to their respective maximum preemption intervals.

At this point the maximum preemption intervals have been determined for all four tasks and the preferred preemption points can be selected. For this illustration, the preferred preemption points will be set to the maximum preemption intervals. Task T1 is always non-preemptable. Task T2 has a preemption point at 6 time units into its execution. Task T3 has preemption points at 5 and 10 time units into its execution. Task T4 has a preemption point at 5 units into its execution. Figure 5 shows the tasks as they would be scheduled under Step 4 with the exception that preemption overhead is not taken into consideration in this figure. It should be noted that in this case the number of preemptions has been reduced to 11. In addition, the preemption cost will be less than the original schedule under the unmodified algorithm due to the appropriate selection of preemption points that reduce preemption costs.

Further work is in progress to refine this modification to the rate monotone scheduling algorithm. Other scheduling algorithms are also under examination to determine the modifications necessary for them.

3: Experimental evaluation

In order to verify that preferred preemption points exist and to confirm the effects that these preferred preemption points have on program execution and cache miss rate, a cache simulation was run with five programs from the PERFECT Benchmark Suite [12]. Program analysis as discussed in Sections 2.2 and 2.3 could have been used instead had it been in place at that time. The trace driven cache simulation is also intended to provide a point of comparison for the results of program analysis in future work. The simulation model was written in C++. The processor was modeled to stall on cache misses. Each miss resulted in a 10-cycle latency for accessing data from main memory. Cache hits had a 1-cycle cache access latency. A cache write-allocate scheme was used for accesses. The cache write-back scheme was setup such

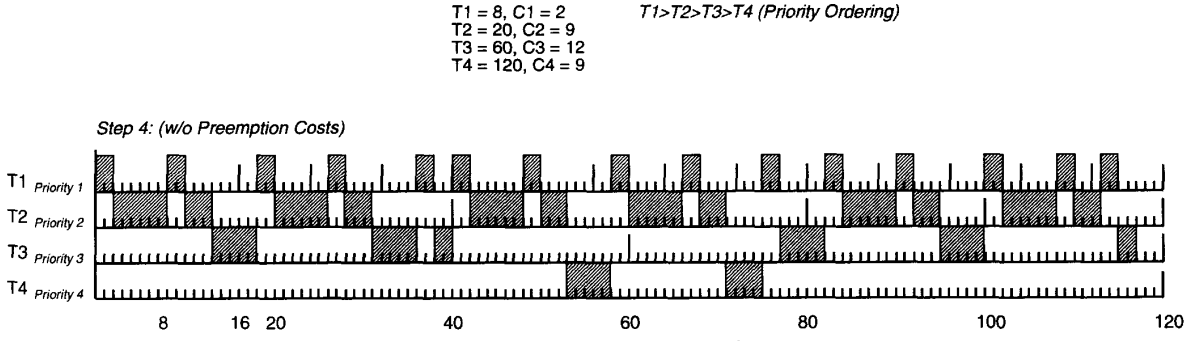


Figure 5: Modified rate monotone schedule w/o preemption costs

that it did not affect the execution time. The traces used in the simulation contained data references only and thus the simulation was restricted to that of a data cache. The cache size was set to 128K Bytes with 32-byte cache lines and direct mapping.

First simulations were done without preemptions to view the live cache line behavior of the task. Whether a cache line was active or not at any particular instant in time was not known until its replacement. For each cache line, a record was maintained of the time of the last load to the line and the last access. At the time of a cache line replacement, a period of activity for the cache line was recorded. The line was considered non-active or dead from the point of last access up until the point at which a new block was loaded. Therefore, in order to know the live line count during each cycle of execution, it was necessary to run the simulation through to completion. It was found, however, that a individual period of cache line activity was almost if not always determined within 200,000 cycles or less. This provided the plots of live cache lines versus cycles given in the next section.

Separate simulations were used to set the actual preemption points and observe the performance improvements. These simulations were run to determine the effects of preempting at preferred preemption points versus immediate and worst-case preemption points. The cache was purged at each preemption point chosen during the simulation. For a large task set, this is a reasonable approach since most of the cache lines have been replaced by intervening tasks. For each simulation run, a set interval was chosen to preempt the task. The other simulation variable was the amount of time allowed before the running task must relinquish the processor at the end of its interval of execution. It is during this time after the end of the interval that the preferred and worst-case preemption points are sought. Live cache lines are calculated for this period requiring the simulation to run until the end of the next interval where the cache is purged. At this point the active times of all cache lines within the search period can be determined. Purging the cache at the end of the next interval is reasonable because it is to be purged soon after that interval completes anyway. At this point the simulation is returned to the state at the end of the last interval, and simulation resumes, preempting the running task at the determined point. For each task preemption interval and search length, preferred preemption points were chosen and worst-case preemption points were chosen. In addition for each preemption interval, preemptions were made immediately at interval boundaries. This method of determining preferred preemption points was used purely as a way of determining the performance improvements that could be seen with our technique. Additionally, the execution paths taken were not necessarily worst-case but it is believed that this is unnecessary for seeing the performance gains achievable.

The PERFECT Benchmarks used were ADM, BDNA, TRACK, ARC2D, and DYFESM. These memory traces were obtained using an Alliant simulator on a Alliant FX/80 [13]. The traces from ADM, BDNA and TRACK were approximately 110 million continuous data references. ARC2D and DYFESM traces were composed of 40 sample traces each. These samples comprised a total of approximately 9 million data references for each application. In the cache simulations with ARC2D and DYFESM, the cache was flushed before the start of each sample. The Alliant simulator was set to emulate execution on an Alliant FX/8 single processor. TRACK is probably most representative as an application of a real-time process. TRACK contains 33770 lines of signal processing code that is used for tracking objects. ADM is an

application for air pollution analysis and contains 6142 lines of code. BDNA is a nucleic acid simulation with 3962 lines of code. ARC2D is computational fluid dynamics with 3605 lines of code and DYFESM is structural dynamics with 7599 lines of code.

4: Results

The simulation runs of ADM, BDNA, and TRACK without preemptions are shown in Figures 4, 5, and 6. The horizontal axis is the time line in millions of processor cycles. As can be seen from these plots, TRACK exhibits less frequent live cache line variance than does ADM or BDNA. This suggests that the effects of preferred preemption points will have less of a impact on the performance of TRACK than ADM and BDNA. This is supported by the results of simulations with preemptions to be shown below.

ADM, BDNA, and TRACK were run with preemption frequencies of 1/2500, 1/5000, and 1/10000 (in 1/processor-cycles). It was with this frequency that the actual preemption points where sought. Each search extended for a period of program execution equal to 10%, 50%, and 90% of the preemption interval. The preferred and worst-case preemption points were chosen from this search period. A worst-case preemption point exhibits the largest number of live cache lines within the search period. Simulations were also run with preemptions occurring immediately at the start of the search period. These preemptions are referred to as immediate preemptions. Table 1 shows percent decrease in execution time over worst case and

Table 1: Percent reduction in execution time via preferred preemption points relative to immediate (Ime) & worst-Case (Max) execution

Preemption Freq.†		1/2500			1/5000			1/10000		
Search Length ††		10%	50%	90%	10%	50%	90%	10%	50%	90%
ADM	Ime	1.8%	5.5%	8.2%	1.0%	4.6%	8.4%	0.7%	2.5%	6.2%
	Max	3.6%	9.8%	12.8%	1.8%	7.1%	11.0%	1.4%	4.1%	7.8%
BDNA	Ime	0.8%	4.6%	8.4%	0.6%	2.7%	6.9%	0.4%	1.9%	4.8%
	Max	1.6%	6.4%	10.4%	1.0%	3.9%	8.3%	0.7%	2.8%	5.8%
TRACK	Ime	0.5%	1.7%	2.6%	0.4%	1.1%	1.9%	0.3%	1.0%	1.7%
	Max	0.9%	2.7%	3.8%	0.7%	2.0%	2.9%	0.5%	1.6%	2.5%

† Preemption Frequency in 1/Processor-Cycles
†† Search Length as Percentage of Preemption Period

ime - preempt immediately upon request
max - assuming worst-case preemption point

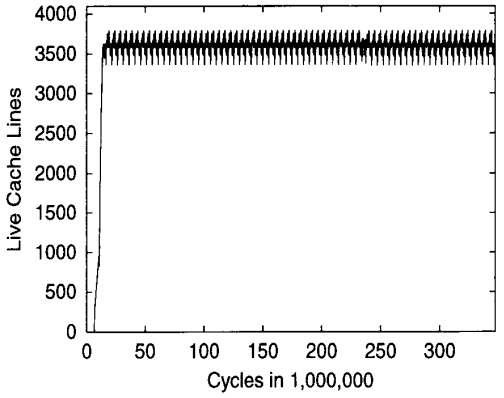


Figure 4: Simulation of ADM w/o preemptions

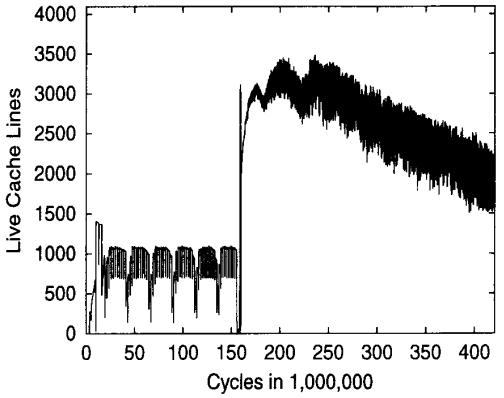


Figure 5: Simulation of BDNA w/o preemptions

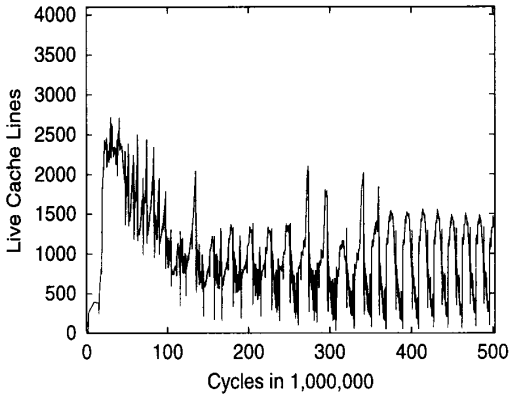


Figure 6: Simulation of TRACK w/o preemptions

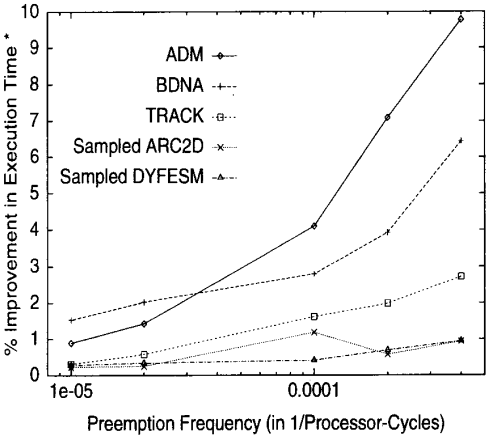
immediate preemption achieved via preferred preemption points. The miss ratio for worst-case preemption points, immediate preemption, and preferred preemption points are given in Table 2. Figure 7 presents the percent reduction in execution time for preferred preemption points relative to using worst-case preemptions points when a search length of 50% of the preemption interval is used. It also gives performance numbers for ARC2D and DYFESM not shown in Table 1.

These results show greater performance gains with preferred preemption points at higher preemption frequencies. This is expected due to the larger accumulated cost of poor preemption point selection that occurs at higher preemption frequencies. This, however, does not always hold true. For tasks that have a greater stability in their live cache line activity with occasional fluctuations, the performance improvement will be seen to be greater at the frequency of these fluctuations. It should also be noted that systems with greater cache miss penalties will enjoy even greater performance benefits from preferred preemption point usage. No matter what the performance improvement, however, predetermined preferred preemptions points provide a means of calculating tighter bounds on WCETs thus allowing for greater system throughput. It should also be noted here that even without predetermined preemption points, having the cost of the worst-case preemption point for the entire program execution permits tighter execution time bounds when there exists a bound on the number of preemptions.

Table 2: Miss ratio

Preemption Freq. †		1/2500			1/5000			1/10000		
Search Length ††		10%	50%	90%	10%	50%	90%	10%	50%	90%
ADM	Max	.25	.27	.27	.19	.20	.20	.13	.13	.13
	lme		.24			.18			.12	
	PPP	.23	.21	.20	.18	.16	.14	.12	.11	.10
BDNA	Max	.32	.32	.33	.23	.23	.24	.17	.17	.17
	lme		.31			.23			.17	
	PPP	.30	.28	.25	.22	.21	.18	.17	.16	.14
TRACK	Max	.11	.12	.12	.09	.09	.10	.07	.08	.08
	lme		.11			.09			.07	
	PPP	.11	.10	.10	.09	.08	.08	.07	.07	.06

† Preemption Frequency in 1/Processor-Cycles
†† Search Length as Percentage of Preemption Period
PPP - Preferred Preemption Point



* % Improvement in Execution Time with Preferred Preemption Points over Worst-Case Preemption Points

Figure 7: Comparison at search length of 50% preemption period

5: Conclusion and future work

Caches are instrumental in providing high system performance. At the same time they add greatly to the difficulty of obtaining tight bounds on worst-case execution times. WCETs are essential in real-time systems in order to guarantee that system timing constraints are met. Past real-time systems designs have not included the effects of caching in their timing calculations. This circumvents the added difficulty of calculating WCETs due to caching but unfortunately leads to under utilization of processor power and memory hierarchy. Since real-time systems are often designed for time critical applications requiring quick response times, it is important that caches be considered in WCETs so that these times can be reduced.

The unpredictable nature of cache memory accesses lies in the variability of the cache state in a multi-tasking preemptable environment. Whether a cache access is a hit or a miss at a particular point of program execution can change as a result of task sequencing, preemption frequency, and point of preemption occurrence. A task's working set can be partially or even completely replaced when preempted by the time it resumes execution, thus requiring it to restore part or all of the working set from

main memory. It is this cache-reload time that is difficult to determine. The significance of the changes in cache state due to preemptions can be minimized through the use preferred preemption points which we have outlined in this paper. Preferred preemption points are those points within a task's execution that exhibit large variances in the working set. Since the working set is in the process of changing at such times, the cached working set need not be restored in its entirety when the task resumes. By determining an upper bound on the number of active cache lines at these chosen preemption points, we can predict the time cost of these preemptions.

The use of preferred preemption points, in comparison to worst-case preemptions points, resulted in improvements in execution time as high as 12%. It was also found, however, that with preemption frequencies lower than 1/100,000 processor-cycles, the percent improvement with preferred preemption points over worst-case preemption points is minimal. Even so, WCETs with preemptions can still be predicted taking into account the cache memory, thus reducing the WCET used for a task.

Currently the direction of this work focuses on developing methods of selecting preferred preemptions points both dynamically and statically, as well as looking at adjusting various scheduling algorithms to account for this technique. There are also plans to examine this technique more closely with regards to instruction caching. As the work proceeds other approaches to this problem will be sought. These approaches may make use of multi-level caches, code reorganization, and other preexisting architecture performance improvement techniques. The objective is to make a more predictable system while maintaining the same performance level or even achieving greater performance.

References

[1] Wei-Kuan Shih, Jane W. S. Liu, and Jen-Yao Chung, "Algorithms for Scheduling Imprecise Computations with Timing Constraints," *SIAM J. Computing*, vol. 20, no. 3, pp. 537-552, June 1991.

[2] David B. Kirk, "SMART (Strategic Memory Allocation for Real-Time) Cache Design," *Proceedings of the Real-Time Systems Symposium*, pp. 229-237, IEEE, December 1989.

[3] David B. Kirk and Jay K. Strosnider, "SMART (Strategic Memory Allocation for Real-Time) Cache Design Using the MIPS R3000," *Proceedings of the Real-Time Systems Symposium*, pp. 322-330, IEEE, December 1990.

[4] David B. Kirk, Jay K. Strosnider, and John E. Sasinowski, "Allocating SMART Cache Segments for Schedulability," *Proceedings of the Euromicro'91 Workshop on Real-Time Systems*, pp. 41-50, IEEE, June 1991.

[5] D. Niehaus, E. Nahum, and J.A. Stankovic, "Predictable Real-Time Caching in the Spring System," *Proceedings of the IFAC Workshop on Real Time Programming*, pp. 79-83, May 1991.

[6] Robert Arnold, Frank Mueller, David Whally, and Marion Harmon, "Bounding Worst-Case Instruction Cache Performance," *Proceedings of the Real-Time Systems Symposium*, pp. 172-181, IEEE, December 1994.

[7] Jyh-Charn Liu and Hung-Ju Lee, "Deterministic Upperbounds of the Worst-Case Execution Times of Cached Programs," *Proceedings of the Real-Time Systems Symposium*, pp. 182-191, IEEE, December 1994.

[8] Swagato Basumalick and Kelvin Nilsen, "Incorporating Caches in Real-Time Systems," *Proceedings of the Workshop on Architectures for Real-Time Applications.*, IEEE, April 1994.

[9] C.L. Liu and J.W. Layland,, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the Association of Computing Machinery*, vol. 20, no. 1, pp. 46-61, January 1973.

[10] John Lehoczky, Lui Sha, and Ye DIng, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," *Proceedings of the Real-Time Systems Symposium*, pp. 166-171, IEEE, December 1989.

[11] Lui Sha, Ragunathan Rajkumar, and John Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computer*, vol. 39, no. 9, pp. 1175-1185, IEEE, September 1990.

[12] M. Berry *et al., "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers," The International Journal of Supercomputer Applications*, vol. 3, no. 3, pp. 5-40, MIT Press, 1989.

[13] John W. C. Fu and Janak H. Patel, "Trace Driven Simulation Using Sampled Traces," *Proceedings of the 27th Hawaii International Conference on System Sciences*, pp. 211-220, 1994.

Performance Recovery in Direct - Mapped Faulty Caches via the Use of a Very Small Fully Associative Spare Cache

H. T. Vergos & D. Nikolos

Computer Technology Institute, Kolokotroni 3 , Patras, Greece
&
Computer Engineering and Informatics Department,
University of Patras, 26500 Rio, Patras, Greece.

Abstract

Single chip VLSI processors use on-chip cache memories to satisfy the memory bandwidth demands of CPU. By tolerating cache defects without a noticeable performance degradation, the yield of VLSI processors can be enhanced considerably.

In this paper we investigate how much of the lost hit ratio due to faulty block disabling in direct-mapped caches can be recovered by the incorporation of a very small fully associative spare cache. The recovery percentage that can be achieved as a function of the primary cache's parameters (cache size, block size), the number of faulty blocks and the size of the spare cache is derived by trace driven simulation. The results show that when the number of the faulty blocks is small the use of a spare cache with only one block offers a hit ratio recovery of more than 70%, which increases further with cache size. A spare cache with two blocks is justified only in the case of a large number of faulty blocks.

1. Introduction

Single-chip VLSI processors use on-chip cache memory to provide adequate memory bandwidth and reduced memory latency for the CPU [4 - 10]. The area devoted to some on-chip caches is already a large fraction of the chip area and is expected to be larger in the near future. For example, in the MIPS-X processor [6] more than half of the chip area is devoted to an on-chip instruction cache.

Since in the near future a large fraction of the chip area will be devoted to on-chip caches, we expect that in a large fraction of VLSI processor chips the manufacturing defects will be present in the cache memory portion of the chip. Application of yield improvement models [11] suggests that, by tolerating cache defects without a substantial performance degradation the yield of VLSI processors can be enhanced considerably.

A technique for tolerating defects is the use of redundancy [25]. The use of redundancy to tolerate defects in cache memories was discussed in [1, 2]. Redundancy can have the form of spare cache blocks where if a block is defective it can, after the production testing, be switched out and substituted by a spare block using electrical or laser fuses. Instead of spare cache blocks, spare word lines and/or bit lines may exist that are selected instead of faulty ones. The overhead of these techniques includes the chip area for the spare blocks or word lines/bit lines and logic needed to implement the reconfiguration. Another form of redundancy is the use of extra bits per word to store an error correcting code [26]. Sohi [1] investigated the application of a Single Error Correcting and Double Error Detecting (SEC-DED) Hamming code in an on-chip cache memory and found out that it degrades the overall memory access time significantly. Therefore the classical application of a SEC-DED code in the on-chip cache for yield enhancement does not seem to be an attractive option for high-performance VLSI processors. In [27] it was shown that the defects in the tag store of a cache memory may cause significantly more serious consequences on the integrity and performance of the system than similar defects in the data store of the cache. To this reason a new way of the SEC-DED code exploitation well suited to cache tag memories was proposed. During fault free operation this technique does not add any delay on the critical path of the cache, while in the case of a single error the delay is so small that the cache access time is increased by at most one CPU cycle. Unfortunately, this technique is effective only in the case that the defects cause single errors per word as for example in the case of a bit line defect.

Another technique to tolerate defects in cache memories is the disabling of the faulty cache blocks that was investigated in [1, 2]. It has been shown in [1, 2] that the

0-8186-7059-2/95 $04.00