A Prioritized Cache for Multi-tasking Real-Time Systems

Yudong Tan

School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia, USA 30332
Tel : 404-894-0966
Fax : 404-894-9959
e-mail : ydtan@ece.gatech.edu

Vincent Mooney

School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia, USA 30332
Tel : 404-385-0437
Fax : 404-894-9959
e-mail : mooney@ece.gatech.edu

Abstract: In this paper, we present a new prioritized cache which can reduce the Worst Case Cache Miss Rate (WCCMR) of an application in a multi-tasking environment. The Worst Case Execution Time (WCET) can be estimated more precisely when the cache miss rate has a stable and low upper bound. An experiment in which a MPEG decoder and a Mobile Robot (MR) control program are executed alternatively shows that we can achieve a nearly constant cache miss rate with our method. The cache miss rate of the time critical application (MR) varies from 45% to 15% with a normal cache, while with a prioritized cache the cache miss rate is always less than 13%, a 3X reduction. With this lower WCCMR, we achieve a much tighter WCET estimate, 41% lower in fact, for MR in this example.

I. Introduction

In a real-time system, it is important to estimate the Worst Case Execution Time (WCET) of each task in order to schedule tasks so that timing constraints are not violated. Unfortunately, WCET is hard to estimate in a processor with a cache. While a cache can dramatically speed up processor performance, a cache also complicates processor performance analysis, especially in a multi-tasking environment where one task may evict cache lines used by another task. Cache line evictions cause unpredictability in cache behavior, resulting in pessimistic WCET estimates. In some real-time systems, caching is disabled in order to reduce unpredictability. However, disabling caches degrades processor performance. A better way to solve this problem is to design a new cache management policy that can reduce or eliminate the interference among tasks.

In this paper, we propose a cache management policy which makes the cache behavior more predictable for high priority tasks, resulting in significantly reduced Worst Case Cache Miss Rate (WCCMR) for these tasks. It is helpful to analyze WCET when the cache is more predictable.

This paper is organized as follows. Section 2 investigates some related work. Section 3 elaborates the details of design and implementation of our cache management policy. Section 4 gives the experimental results. Section 5 concludes the paper.

II. Related Work

Interference among tasks in the cache complicates WCET analysis. This problem motivates us to improve cache management policies so that cache behavior is more predictable. Dropso et al. [1] investigate some existing cache management techniques that intend to make caching predictable. They divide these techniques into two categories, Spatial Only Sharing (SOS) and Temporal Only Sharing (TOS). Dropso compares these two methods and concludes that neither policy is superior to the other when considering a large variety of general purpose computing scenarios such as scientific computing, real-time control applications and so on. Effectiveness of each policy depends on the target applications. In [2], a data-replace-controlled cache is presented. Users can control every cache line by setting a cache line in a status of "lock" or "release". Only when a cache line is in the "release" status can it be replaced. Users can "lock" cache lines in order to prevent replacement. Additional instructions are used to release and lock cache lines. Users have to take care of all cache lines that need to be locked/released, which increases users' work. Chiou et al. [3,4,5] proposes a column cache model. In this scheme, caches are partitioned at the granularity of columns. Columns in a set-associative cache can be assigned to a task exclusively so that the cache lines in these columns are not going to be kicked out. This technique requires users to partition the cache explicitly. Approach [6] gives another way to eliminate interference in the cache using specific load/store instructions; also, the compiler is modified to partition the data and instructions. Approaches [7,8] present another cache partitioning method which focuses on memory mapping without considering the tasks properties. Approaches [9,10] present cache partitioning schemes in which timing requirements of tasks are taken into account. However, only the utilization of tasks are considered in the their cache allocation algorithms, no matter which scheduling algorithm is actually used by the system outside of the cache, while in our approach the cache allocation algorithm can be consistent with the task scheduling algorithm in the Real-Time Operating System (RTOS).

We distinguish our contribution from the previous

work in that while the previous work requires the user to map code/data to specific memory locations, use specific instructions, or partition the cache explicitly, we only require the user to choose a priority for each task. Cache allocation is dependent on the task scheduling algorithm used in the real-time system so that the cache allocation in our scheme is flexible. In short, we provide a new method with a simple interface to the user and with transparent hardware details.

III. Prioritized Cache

As mentioned above, the difficulties in estimating WCET of real-time tasks in a multi-tasking environment with caches lie in interference among tasks. Cache lines used by one task may be evicted by another task when the former is suspended. One way to help solve this problem is to divide the cache into several partitions. Each task is assigned one or more partitions exclusively so that interference among tasks is eliminated. In this method, the algorithms of partitioning the cache and assigning partitions to tasks are important. Here, we propose a assignment strategy by borrowing some ideas from real-time scheduling.

A. Assignment Strategy

We target set associative caches in this paper. In a multi-way set associative cache, one "way" is called a column [3]. For example, a 4-way set associative cache has four columns. The cache is partitioned at the granularity of columns, that is, all cache lines in one column are always in the same partition. When a column in the cache is assigned to a task, that task is called the owner of the column and the column is owned by the task. Not all columns need to be assigned to tasks. We can also set a column to the status of "shared" so that the column can be shared by tasks.

We want to assign cache partitions to tasks according to their priorities. Priorities are widely used in task scheduling of real-time systems. Depending on the scheduling algorithm chosen, priorities of tasks may be fixed (e.g., Rate Monotonic Scheduling) or dynamic (e.g., Earliest Deadline First or the Priority Ceiling Protocol). Usually, those tasks with strict timing constraints have higher priorities in using CPU resources. Note that these existing scheduling algorithms (e.g., RMS, EDF and PCP as mentioned above) are used to allocate CPU resources. The priorities of tasks are not taken into account in conventional cache allocation. However, tasks with strict timing constraints should have higher priorities not only for using the CPU but also in using other resources such as caches. With this intuition, we divide a cache into partitions and assign partitions to each task according to its priority. In this section, we do not address the problem of choosing task priorities, but assume that each task has been assigned a unique priority (not assigned to any other task) with an existing priority-based scheduling algorithm such as RMS or EDF. (At the end of this section we will extend our approach to handle multiple tasks with the **same** priority.) We focus instead on how to assign cache partitions to tasks according to their priorities.

Now, we assume that priorities of tasks range from 0 to N. 0 is the highest priority and N is the lowest priority. We also give each column a priority. At the beginning, the priority of every column in the cache is the lowest one (N). When a task needs to use the cache, the cache controller compares the priority of the task with the priority of each column. To accomplish this, we added a special register to the cache controller as described in the next subsection. Only when the priority of a task is higher than or equal to the priority of a column can the task use the column. In other words, a task with a higher priority can use all columns owned by tasks with lower priorities. When a column is used by a task, the priority of the column is upgraded to be equal to the priority of the task. After a task completes, it notifies the cache controller to release all columns the task owns. The cache controller does this by setting priorities to those columns to the lowest priority again. Let us consider an example as below.

Example 1. Suppose we have two tasks, an MPEG decoder (MPEG for short) and a Mobile Robot Control program (MR for short). The MR application is derived from Missionlab, which is mobile robot control software developed by the Georgia Tech Mobile Robot Lab [13]. The Control Flow Graph (CFG) of MR is shown in Figure 1.

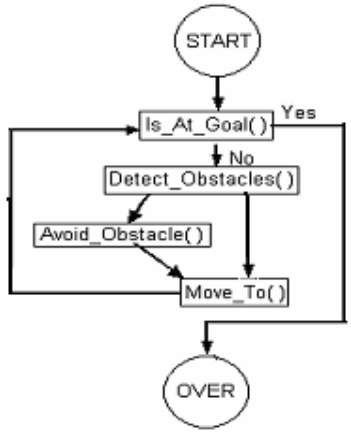


Figure 1. Control Flow Graph of MR

MPEG and MR are two different kinds of applications. MPEG is a data-processing application with soft real-time constraints. MR depends on a fixed size of data, which mainly includes the coordinates of the robot and the coordinates of obstacles. However, MR has a more strict timing requirement than MPEG. Thus, a tight WCET analysis for MR is needed. According to the CFG shown in Figure 1, we can see that the worst case control path in MR is the path from *Is_At_Goal( )* to *Move_To( )* via *Detect_Obstacles( )* and *Avoid_Obstacles( ).*

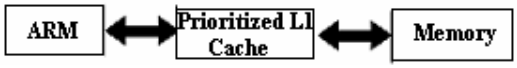


Figure 2. Architecture for Executing MPEG and MR

We assume that MR has a higher priority than MPEG. In this example, there are 4 priorities, where 3 is the lowest and 0 is the highest priority. MR is given a priority of 1 and MPEG is given a priority of 2. We use a 4-way 16KB set associative L1 cache for all instructions and data. Each column has 256 lines. The processor used in this example is ARM9TDMI. Figure 2 shows the architecture for the example. At the very beginning, all columns in the cache are empty, thus, with the lowest priority of 3. When MPEG runs, it uses all four columns. The priorities of these four columns are upgraded to the priority of MPEG, i.e., to 2 as shown in Figure 3(b). Then, MPEG is suspended and MR begins to run. When there is a cache miss, a cache line is chosen to be replaced. If the priority of the column in which the cache line locates is lower than the priority of the task, the priority of this column is upgraded to the priority of the task. In this example, two columns used by MPEG are replaced by MR and the priorities of these two columns are upgraded to 1 as shown in Figure 3(c). So, next time when MPEG is executed, MPEG can only use the other two columns that still have priority 2. From this example, we can see that if there is no other task with an equal or higher priority than MR, MR can use the first two columns exclusively. In this manner, we can guarantee the usage of the cache by high priority tasks at a cost, however, of degrading the performance of lower priority tasks.

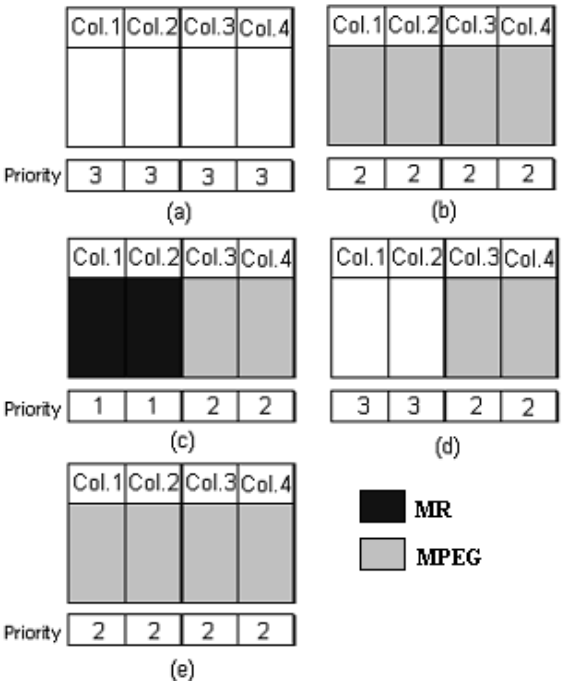


Figure 3. Assignment Strategy in the Prioritized Cache

When a task is completely over, it releases all columns it owns. The cache controller sets the priorities of these columns to the lowest priority. In the example above, we assume that MR is completed earlier than MPEG. When MR is over, it releases the first and the second column and sets the priorities of these two columns to 3, which is shown in Figure 3 (d). So, when MPEG is executed next time, it can use all four columns again, as shown in Figure 3 (e). □

B. Hardware Implementation

The assignment strategy is implemented in the cache controller. We add two tables, the Column Priority Table (CPT) and the Column Owner Table (COT), and three registers, Current Task Register (CTR), Current Task Priority Register (CTPR) and Column Status Register (CSR), to the cache controller. Each column has an entry in CPT and COT. An entry records the priority and the owner of the column. CTR and CTPR are used to save the ID and the priority of the task which is currently running on the processor. CSR indicates if a column is shared. A column is shared if all tasks can use this column. Note that this shared column differs from a column with the lowest priority in that a shared column always has the lowest priority and this priority is never upgraded even if the column is used by a high priority task. Each column has one bit in the CSR. If the bit is set, the corresponding column is shared. An example of such an extended cache controller is shown in Figure 4.

Obviously, the prioritized cache only uses the task ID and the task priorities to allocate columns. The prioritized cache does not limit the number of tasks and priorities directly, except for the limitation imposed by the length of the CPT and COT registers. For example, if the CPT and COT entries each have 16 bits, up to 2^{16} tasks and 2^{16} different priorities can be supported, which is sufficient for many real-time systems. Suppose we have a *m*-way set associative cache, a maximum of 2^n tasks and a maximum of 2^k different priorities. Clearly, then, we have m entries each in the COT and CPT tables (i.e., m columns or ways). Each COT entry needs *n* bits, for a total usage of $m \times n$ bits. Each CPT entry needs *k* bits, for a total usage of $m \times k$ bits. The CTR register has *n* bits, while CTPR has k bits. Additionally, the CSR register needs *m* bits. Therefore, in total, we need $m \times n + m \times k + n + k + m = (m+1)(k+n) + m$ bits for the CPT and COT tables and the CTR, CTPR and CSR registers. Example 2 shows these extra tables and registers in a 4-way set-associative prioritized cache. The prioritized cache does not require much more area than a normal cache. For a prioritized 16KB 4-way cache which supports 64 tasks and 64 priorities, the area increases by less than 1% if compared with a same size normal cache (i.e., non-prioritized).

Example 2. Suppose we have a 16KB cache with 4 columns as shown in Figure 4. The lengths of the specialized registers in the cache – CPT, COT, CSR and CTR – are 16 bits. Since the CPT and COT registers are each 16 bits long, this cache supports up to 2^{16} tasks and 2^{16} priorities. In this example, *k*=the number of bits in each CPT register=16, *n*=the number of bits in each COT register=16, *m*=the number of columns=4; thus, we need

$(m+1)(k+n)+m = (4+1)\times(16+16)+4=164$ extra bits for the prioritized 16KB cache.

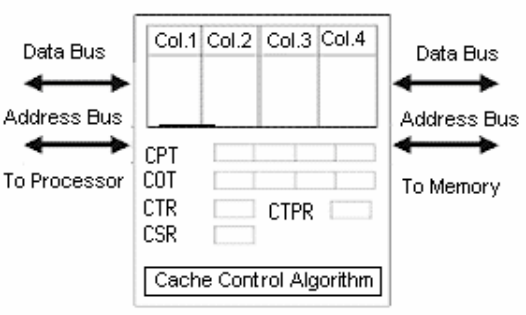


Figure 4. Extended Cache Controller

In Example 1, the prioritized cache needs to be initialized before it is used (as shown in Figure 3(a)). We notice that most Operating Systems have an IDLE task which controls the CPU when there is no other task running. The IDLE task has the lowest priority. Therefore, we can assign the ID and the priority of the IDLE task to the corresponding registers in the prioritized cache for initialization. We assume that the IDLE task has an ID of 0 and a priority of *0xFFFF* which is the lowest priority. Therefore, all of the CPT entries in Table 1 are *0xFFFF*, and all of the COT entries are zero. Also, in order to allow low priority tasks to use the cache, we set the fourth column as shared by default. The initial settings of the registers are listed in Table 1. □

CPT	0xFFFF	0xFFFF	0xFFFF	0xFFFF
COT	0x0000	0x0000	0x0000	0x0000

CTR	0x0000

CTR	0xFFFF

CSR	0001

Table 1 Initialization of Registers in the Prioritized Cache

We give memory mapped addresses to the tables and the registers so that the values in the tables and registers can be set as needed. Alternatively, specialized assembly instructions for accessing these tables and registers can be defined if the target instruction set has a sufficient number of undefined assembly instructions and if the processor can be redesigned to support the new specialized assembly instructions. Usually, we only need to set the value of CTR and CTPR. Every time when the task is switched, we write the task ID and the priority of the task to be executed to the CTR and CTPR.

When the cache hits, the prioritized cache works the same as the normal cache. When there is a cache miss, a cache line is searched for replacement. The columns owned by the current task, the columns with a lower priority than the current task and the shared columns are searched. If the cache line to be replaced is located in a column that is not owned by the current task, the priority and the owner of the column are updated, that is, the values in CTR and CTPR are copied to the corresponding entries in COT and CPT. When a cache line is replaced, its status needs to be updated. The entries in COT and CPT can be updated concurrently. When a column is released, the priority of this column is set to the lowest priority.

Example 3. In Example 1, suppose MPEG has an ID of 1 and MR has an ID of 2. Suppose further that the priority of MPEG is 2 while the priority of MR is 1. Column 3 is set as shared by default in order to allow low priority tasks to always be able to use at least one column in the cache. After MPEG runs, all four columns are owned by MPEG (as shown in Figure 3(b)). The status of registers in the prioritized cache is shown in Line 1 of Table 2. Then, MR begins to run for the first time. MR writes its ID and priority to the appropriate registers in the cache, which is shown in Line 2 of Table 2. MR needs to load its instructions and data to the cache. The cache searches for a cache line to be replaced. Since MPEG has a lower priority than MR, the first column, which is owned by MPEG, is selected and assigned to MR. For the same reason, the second column is also assigned to MR (as shown in Figure 3(c)). The values in registers are changed as shown in Line 3 of Table 1. □

Line	CTR	CTPR	CSR	Priority of Columns (CPT)				Owner of Columns (COT)			
				0	1	2	3	0	1	2	3
1	1	2	0001	2	2	2	3	1	1	1	0
2	2	1	0001	2	2	2	3	1	1	1	0
3	2	1	0001	1	1	2	3	2	2	1	0

Table 2 An Example of Cache Replacement

```
1. Set_column_shared(3);
2. while(!MPEG_over()&&!MR_over()){
3.    Set_tid_pri(MPEG_ID,2);
4.    MPEG_decode_one_slice();
5.    Set_tid_pri(MR_ID,1);
6.    MR_move_one_step();
7.    if(MPEG_over())
8.       Release_column(MPEG_ID);
9.    if(MR_over())
10.      Release_column(MR_ID); }
```

Figure 5. Code Using APIs to Control the Prioritized Cache

C. *Software Interface*

The prioritized cache is software controllable. We provide APIs for users or the RTOS to configure the cache. API functions can change the values in COT, CPT, CTR, CSR and CTPR to assign or release the columns. We provide four APIs. *Set_tid_pri(tid,pri)* writes the priority and ID of the current task into CTR and CPTR, respectively. *Set_column_pri(col,pri)* sets the priority of a column. *Release_column(tid)* releases all columns owned by the task with an ID of *tid*. *Set_ column_shared(col)* sets a column to a status of shared. These APIs can be implemented as system calls in an RTOS or a general purpose OS. We give an example below to show how we use these APIs in the MPEG and MR applications.

Line	CTR	CTPR	CSR	Priority of Columns				Owner of Columns			
				0	1	2	3	0	1	2	3
Initial	0	0	0000	3	3	3	3	0	0	0	0
1	0	0	0001	3	3	3	3	0	0	0	0
3	1	2	0001	3	3	3	3	0	0	0	0
4	1	2	0001	2	2	2	3	1	1	1	0
5	2	1	0001	2	2	2	3	1	1	1	0
6	2	1	0001	1	1	2	3	2	2	1	0
8	2	1	0001	1	1	3	3	2	2	0	0
10	2	1	0001	3	3	3	3	0	0	0	0

Table 3. Changes of Values in Registers and Tables

Example 4. Consider the example shown in Figure 3. If we set Column 3 to be shared and execute MPEG and MR alternatively, we can implement this example with the C code shown in Figure 5. Table 3 shows how the CTR, CTPR, CSR, the priority and the owner of each column changes after each line of code in Figure 5 is executed. We give MPEG an ID of 1 and MR an ID of 2. After line 1 of Figure 5 is executed, Column 3 is set to be shared: thus, the value in CSR is changed to 0001 as can be seen in the second row of Table 3. *Set_tid_pri( )* is called in line 5 of Figure 5 in order to write the ID and priority of MR to CTR and CTPR. Thus, the values in CTR and CTPR are changed to 2 and 1 respectively as can be seen in the fifth row of Table 3. Then, MR starts to run. As described in Example 3, the first two columns of MPEG are assigned to MR. Thus, the register values are changed as shown in the sixth row of Table 3. In Line 8 of Figure 5, MPEG releases all of its columns if MPEG is over, which causes the change of register values indicating column priorities and owners, as shown in the seventh row of Table 3. When MR is over, it also releases all of its columns. The last row of Table 3 gives the status of registers after MR is over. □

D. Embedding Prioritized Cache APIs in an OS Kernel

We provide APIs for users to configure the prioritized cache. However, users do not need to call these APIs directly; instead, the APIs can be embedded into the OS system calls. For example, we can insert *Set_tid_pri()* into the context switch function so that every time the task is switched, the priority and the ID of the current task is written into CTR and CTPR in the prioritized cache respectively. We can also embed *Release_Column()* into the task destruction function so that when one task is completed, all the columns it owns are released. Obviously, the changes needed to be made in the OS are minor. By embedding prioritized cache control APIs into the OS kernel, the details of the prioritized cache are transparent to users. Thus, users can focus on application development at a higher level of abstraction.

E. Multiple Tasks with the Same Priority

In the descriptions above, we assume that each task has a unique priority. This assumption may not be true in real-time systems because different tasks may have the same priorities. In this case, we need to decide how to allocate caches among tasks with the same priority. In the current implementation of our prioritized cache, the columns used by a task can be shared by other tasks with the same priority. The LRU algorithm is used for cache line replacement.

IV. Experiment

In order to verify the effectiveness of the prioritized cache model, we ran two applications, an MPEG decoder (MPEG for short) and a Mobile Robot control program (MR for short), on a ARM9TDMI utilizing a prioritized L1 cache: a 16KB 4-way set associative cache. We simulated the cache hardware using Verilog and the software using Seamless CVE, a hardware/software co-verification environment provided by Mentor Graphics [11, 12]. Specially, we use the ARM9TDMI Processor Support Package (PSP) in this experiment.

We design two cases for experiments, which are explained in detail in the remaining sections.

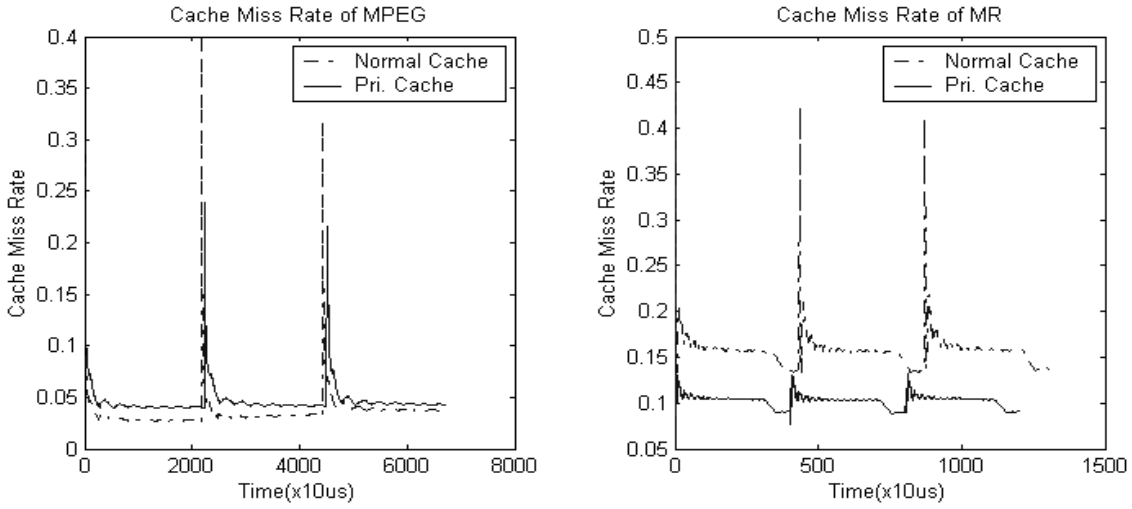
A. Case 1

We have two tasks, MPEG and MR, which execute alternatively. After a slice of data is decoded by MPEG, MPEG is suspended and MR is executed. After MR moves one step forward, MR is suspended and MPEG is executed again. In this experiment, we do not use any scheduler but rather have the two tasks execute in a co-routine-like fashion as described above.

As we mentioned before, our goal is to obtain a more precise estimate of WCET for high priority real-time tasks. We achieve this goal by dramatically reducing the WCCMR of high priority tasks. We assume that MR is more time critical than MPEG. Thus, we aim to make the cache miss rate of MR more predictable with our approach so that we can more precisely estimate WCET of MR, which helps MR to not miss its deadline. In order to reach this goal, we give MR a higher priority than MPEG. The priority of MR is 1, while MPEG has priority 2.

We first use a normal cache with an LRU cache line replacement algorithm to run this example. Then, we use a prioritized cache also with an LRU cache line replacement algorithm. In the two cases, we run both MPEG and MR three times. We sample the cache miss rate of MR every 10us. The result is shown in Figure 6, about which we note several observations.

In the normal cache, the cache miss rate of MR varies between 45% and 15% in each run, as shown in Figure 6(b). The reason for this variance is that the cache lines used by MR are evicted by MPEG when MR is suspended and MPEG begins to run. The uncertainty in cache miss rate makes the prediction of WCET of MR artificially high (see Table 4).



(a) (b)

Figure 6. Comparison of Results. (a) Cache Miss Rate of MPEG. (b) Cache miss rate of MR. (Note: MR and MPEG are executed alternatively three times. Thus, the execution times of three runs are not continuous actually. But for convenience of comparison, we concatenate three runs of MR and MPEG respectively)

With the prioritized cache, on the other hand, the MR cache miss rate is less than 13%, except in the very first run when the cache is cold. We need to point out that, in this experiment, we create a scenario for MR in which the robot has to avoid obstacles in every step. The code has no branches other than the branches used to implement the control flow of Figure 1. All branches are executed along the longest code path accessing all memory locations in any other code path, plus some more. Thus, MR is always executed along the worst case path. On the other hand, the MR program is also not data-dependent. The amount of data (i.e., coordinates of the robot, the obstacle and the goal) it needs to process is always the same in each run. No data-dependent loops, memory stores, or other such memory operations affected by data dependencies exist in the MR code. Furthermore, because MR has the highest priority, no other tasks can replace the cache lines used by MR after they are loaded the first time. Therefore, our simulation captures the worst case cache behavior for this simple MR code. Thus, we can use the cache miss rate in this experiment which is 13% as the WCCMR for WCET analysis of MR. Compared with the normal cache, the WCCMR of MR is reduced by a factor of 3.5 when using the prioritized cache.

One might wonder how a reduction in WCCMR can impact WCET estimation? In our specific example, the MR code is fairly straightforward, this allowing us to use a simple WCET analysis method that is not broadly applicable.

We have pointed out that our simulation of MR captures the worst case execution path. We also assume there are no exceptions. Thus, if we only consider the effect of the cache, we can use a simple method sufficient to accurately estimate the WCET for the MR code [14]. On the basis of the WCCMR for this worst case code path, we show how the WCCMR can affect the estimate of WCET as follows.

Suppose we know the number of instructions M, CPI with no cache miss CPI_{ideal}, the penalty of a cache miss P (cycles) and the WCCMR r. We can derive CPI_{ideal} by running applications with a large enough cache so that the cache miss rate is nearly zero. (In this experiment, we run MR using a 1MB cache in order to obtain CPI_{ideal}. The average cache miss rate is zero after the second run.) We can use the following formula to estimate WCET:

$$\text{WCET}=M\times CPI_{ideal}+r\times M\times P$$

In our experiment, we want to estimate the WCET of MR, where M=238959, CPI_{ideal}=1.26 and P=4. As shown in the experimental result, the WCCMR using the normal cache is 45%, while the WCCMR using the prioritized cache is 13%. We use the WCCMRs 45% and 13% to estimate WCET of MR respectively, then compare the estimates with the actual execution time. In our simulation, the actual execution time of MR using a prioritized cache was 409286 cycles. Table 4 shows the result, a 41% reduction in WCET estimate.

r	WCET estimates
0.45	731214
0.13	425347

Table 4 Comparison of WCET Estimates

Note that although in this specific example, only a simple analytical WCET calculation method was used, it sufficed. We hypothesize that similar reductions in WCET would be observed in more complicated software

requiring more sophisticated WCET analysis techniques. The example here gives us an insight that we can estimate the WCET with a tighter bound by reducing WCCMR.

In our method, the performance of lower priority tasks is sacrificed. If we check the cache miss rate of MPEG in this example, we find that the average miss rate of MPEG increases only by 2%, thus increasing overall MPEG execution time by only 3%. In the normal cache, MPEG can always use all the columns. However, only two columns can be used by MPEG in the prioritized cache. That is the reason why the cache miss rate of MPEG is increased.

B. Case 2

In order to evaluate the performance of the prioritized cache when multiple tasks have the same priority, we design a case in which there are one instance of the MPEG application and two instances of the MR application. They are still executed one by one in a co-routine-like fashion. The WCCMRs are compared in Figure 7.

From the experimental results, we can see that the WCCMR of MR is increased if compared with the result in Case 1. However, the WCCMR of MR in Case 2 is still much less than in the normal cache case. On the other hand, if the number of tasks which have the same priority is large, it is not hard to see that the performance of the prioritized cache will deteriorate. The reason is that the partition of cache shared by the tasks with the same priorities works the same as a normal cache with LRU replacement strategy. Thus, our current implementation of the prioritized cache needs to be improved in order to handle the case in which a large number of tasks have the same priority. One possible way is to partition the cache at a finer granularity than columns. This will be considered in our future research. Compared with other cache partitioning schemes [2-10], the prioritized cache model has several clear advantages. First, timing constraints of applications are considered in the assignment policy so that highly time critical tasks are given high priorities in using the cache.

Although priority-based scheduling algorithms are widely used in real-time systems, these algorithms are mainly used to allocate CPU resources. Our method provides a way to possibly extend these algorithms to apply to cache allocation so that the performance of high priority tasks are guaranteed with more confidence in systems with caches. Even more, the lower level details of our method are transparent to users. Users only need to call a few APIs in order to use the prioritized cache. In a system with an RTOS, we can even embed the APIs into the OS kernel such as the context switch function and the task destruction function so that the prioritized cache is totally transparent to users.

V. Conclusion and Future Work

In this paper, we present a prioritized cache model. The experiment shows that we can achieve a much lower cache miss rate for high priority tasks with this cache model. With a lower constant cache miss rate, we can estimate WCET of a task more precisely, which is critical is real-time task scheduling.

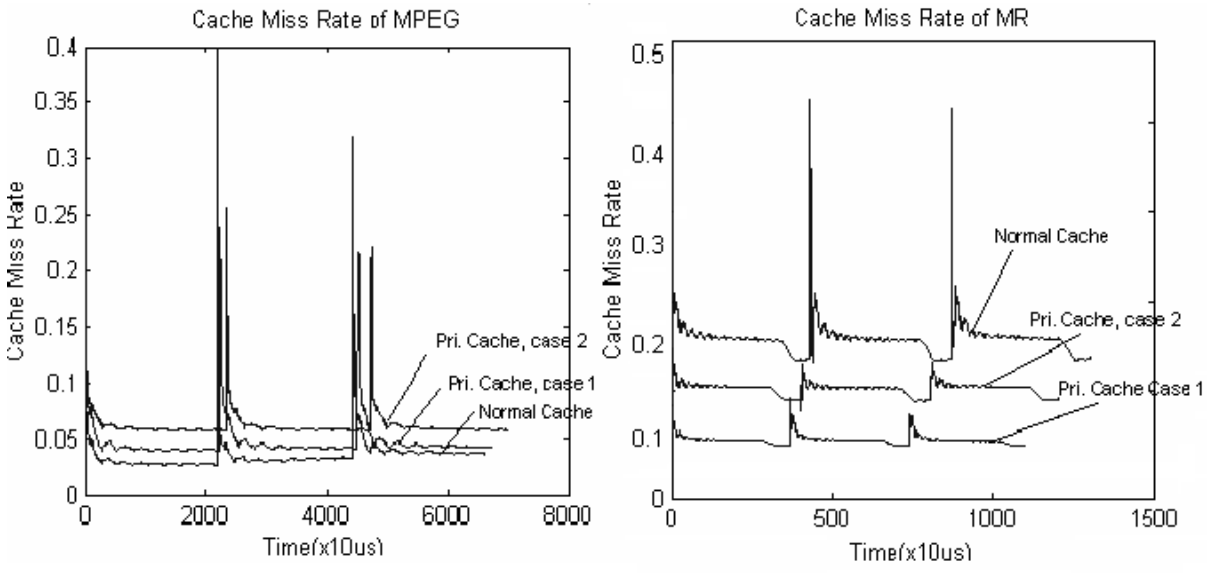


Figure 7 Compare of results (a) Cache Miss Rate of MPEG. (b) Cache miss rate of MR.

For our future work, we plan to improve our approach in several aspects. First, the cache is partitioned at the granularity of columns, which may lower the utilization of the cache. We plan to support partitioning the cache at a lower level of granularity in order to solve this problem. Second, the experiment shows a reduction in WCCMR. Also, we intuitively believe that WCET can be tightly bounded with a significantly reduced WCCMR. However, we still need to build an analytical model to estimate the WCET of applications with the prioritized cache formally. Third, we need to analyze the performance further when there are multiple tasks with the same priorities.

In conclusion, this paper is the first time a connection is made between task priorities and cache column priorities. For real-time applications, this connection can result in a 3X or more reduction in cache miss rates and a corresponding reduction in WCET for critical tasks, which is, we believe, a very important and new result.

VI. Acknowledgement

This research is funded by NSF under INT-9973120, CCR-9984808 and CCR-0082164. We acknowledge donations received from Denali, Hewlett-Packard, Intel, LEDA, Mentor Graphics, Sun and Synopsys.

References

[1] S. Dropso, "Comparing caching techniques for multitasking real-time systems," Technical Report, Computer Science Department, University of Massachusetts, Amherst, UM-CS-1997-065, November, 1997.

[2] N. Maki, K. Hoson and A. Ishida, "A Data-Replace-Controlled Cache Memory System and its Performance Evaluations," *Proceedings of the IEEE Region 10 Conference*, pp. 471-474, April 1999.

[3] D. Chiou, P. Jain, L. Rudolph and S. Devadas, "Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches," *Proceedings of the 37th Design Automation Conference (DAC'00)*, pp. 416-420, June 2000.

[4] G. Suh, L. Rudolph and S. Devadas, "Dynamic Cache Partitioning for Simultaneous Multithreading Systems," *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, August, pp.116-127, September 2001.

[5] P. Jain, S. Devadas, D. Engels and L. Rudolph, "Software-Assisted Cache Replacement Mechanisms for Embedded Systems m," *Proceedings of the Int'l Conference on Computer-Aided Design*, pp. 119-126, November 2001.

[6] D. May, J. Irwin and H. Muller, "Effective Caching for Multithreaded Processors," in P. H. Welch and A. W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pp. 145-154, IOS Press, September 2000.

[7] J. Liedtke, H. Härtig and M. Hohmuth, "OS-Controlled Cache Predictability for Real-Time Systems," *Proceedings of the Third IEEE Real-time Technology and Applications Symposium (RTAS'97)*, pp. 213-227, June, 1997.

[8] J. Löser and H. Härtig, "Cache Influence on Worst Case Execution Time of Network Stacks," Technische Universität Dresden Technical Report TUD-FI02-07, July 2002.

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

[10] S. Shahrier and J. Liu, "On the Design of Multiprogrammed Caches for Hard Real-Time systems," *International Performance, Computers and Communications Computer (IPCCC'97)*, pp. 17-25, February 1997.

[11] "Getting Started With Seamless Co-Verification Environment (Software Version 3.0-1.0)," Seamless Documentation, Mentor Graphics.

[12] Mentor Graphics, Seamless Hardware/Software Co-Verification, http://www.mentor.com/seamless/.

[13] Research Projects: MissionLab, http://www.cc.gatech.edu/ai/robot-lab/research/MissionLab/.

[14] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach* (3rd edition), Morgan Kaufmann, Menlo Park, CA, 2002.