UNIVERSITY OF CALIFORNIA
RIVERSIDE

Cache Management for Critical Sections on Multiprocessors

A Thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Electrical Engineering

by

Ankit Juneja

March 2018

Thesis Committee:

Dr. Hyoseung Kim, Chairperson
Dr. Nael Abu-Ghazaleh
Dr. Daniel Wong

The Thesis of Ankit Juneja is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

First and foremost I offer my sincerest gratitude to my supervisor, Dr. Hyoseung Kim, who has supported me throughout my thesis with his patience and knowledge while allowing me the space to work in my own way. I attribute the level of my Masters degree to his encouragement and effort and without him this thesis would not have been completed or written. One simply could not ask for a better and friendlier supervisor.

Besides my adviser, I would like to thank the rest of my thesis committee: Dr. Nael Abu-Ghazaleh and Dr. Daniel Wong, for their encouragement and insightful comments.

I would also like to thank my fellow lab mates especially Sujan Kumar Saha and Yecheng Xiang in Real-Time Embedded and Networked systems (RTEN) Lab for having the stimulating discussions and supporting me throughout.

To my parents and other family members for their continuous guidance, support and encouragement throughout.

ABSTRACT OF THE THESIS

Cache Management for Critical Sections on Multiprocessors

by

Ankit Juneja

Master of Science, Graduate Program in Electrical Engineering
University of California, Riverside, March 2018
Dr. Hyoseung Kim, Chairperson

Multi-core processors seek for a large last level cache to enhance the overall performance of the system. The problem however is that this last level cache is shared among all the cores which causes inter- and intra- core cache interference between the tasks running on these core. Page Coloring is a popular software cache partitioning approach to address the issue of inter-core cache interference. Also, to utilize multi-core processors better in real-time domain requires task allocation, scheduling and synchronization. Prior works have addressed the issues of scheduling penalties for multi-core processors and software cache partitioning scheme on multiprocessors independently. However, no prior work exist which considered both. Hence, in this thesis, we considered software cache partitioning scheme for the tasks having critical section accessing shared memory on multiprocessors. So for our approach which considers both, we propose a framework which eliminates the inter-core cache interference and bounds the intra-core cache interference for multi-core processors. For our approach we analyzed task schedulability and proposed a cache aware task allocation algorithm. We evaluated our proposed framework in Linux/RK environment on ODROID-XU4

board which is an ARM platform. We use the execution time as our performance metric. Also, we use the page coloring scheme of Linux/RK environment to assign colors to these tasks. As our proposed framework is the first work which considers both, so we can not compare it with the previous work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Background

Modern multi-core processors share resources among different cores to improve their performance and efficiency like sharing the last level cache among different cores. The main objective of last level cache is to reduce the average execution time of the tasks but it is possible that the performance gain achieved, may be offset by timing penalties introduced due to a phenomenon called cache interference in the last level cache.

To understand what cache interference is, consider running different tasks on different cores of a multi-core system as shown in Figure 1.1. These tasks access the large sized shared last level cache to improve system performance. Now, since this cache is shared, it is possible that one task running on one core replaces the cache data of other task running on different core. This is known as inter-core cache interference and can prove to be a major performance bottleneck in multi-core systems with shared last level cache.

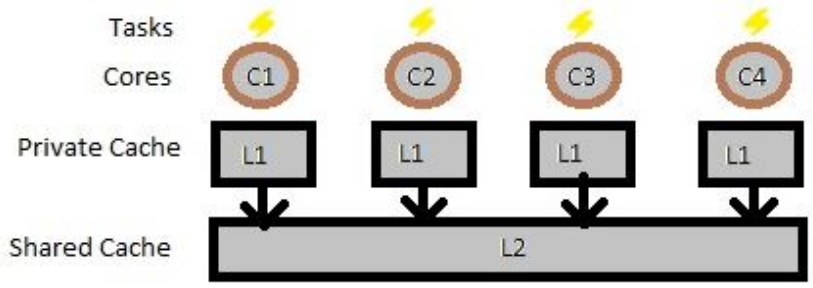

Figure 1.1: Inter-core cache Interference.

There is yet another type of cache interference called the intra-core cache interference shown in Figure 1.2 in which multiple tasks running on the same core can deteriorate the performance of the system not only at last level cache but also at private caches since these tasks have a shared cache memory system.

To overcome inter-core cache interference, hardware and software cache partitioning schemes are employed. A popular scheme known as page coloring is a software cache partitioning scheme which does not require any hardware support. Page coloring avoids cache interference by assigning exclusive cache partitions to each task. So, in Figure 1.1, one can think of task 1 running on core 1 while accessing only a specific portion of cache private to that core while task 2 running on core 2 while accessing its own but different portion of cache. In other words, the cache corruption by other tasks is eliminated by cache partitioning.

Page coloring is a software cache partitioning technique which does not require any hardware support and handles physically-indexed set-associative cache. Page coloring

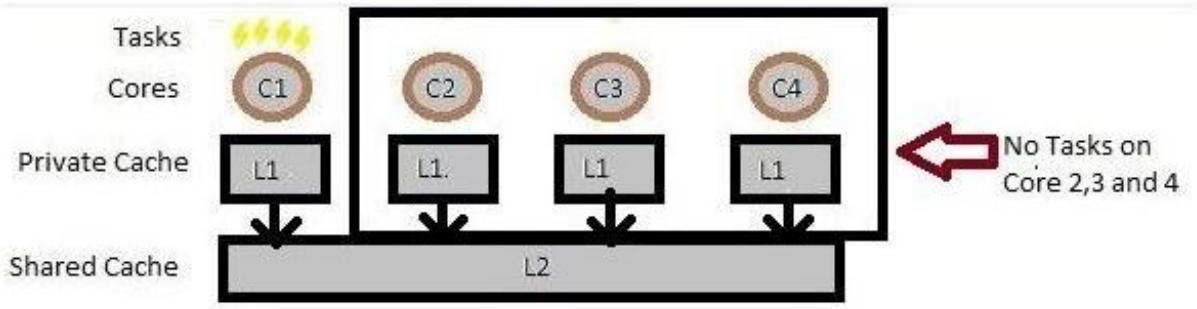

Figure 1.2: Intra-core cache Interference.

maps physical addresses and cache set indices. Figure 1.3 shows that we have overlapping bits between the physical page number and the cache set index. Page coloring uses these overlapping bits as color indices. Operating Systems directly controls the mapping between virtual and physical pages of a task, thus it can allocate specific cache colors to a task by providing it with physical pages that corresponds to the cache colors. The calculation of number of cache colors of a particular system is as follows:

$$n\_cache = s\_cache/(w \times p) \tag{1.1}$$

where, n_cache is the number of cache colors

s_cache is the size of cache

w is the number of ways of the cache and p is the size of a page

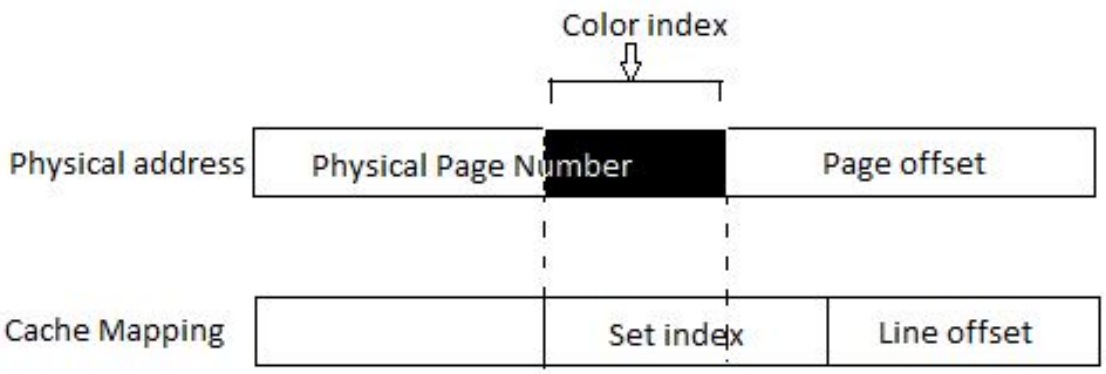

Figure 1.3: Page coloring

But, we assume that the number of cache sets is always a power of two in page coloring scheme. Let us calculate the number of cache colors of our platform ODROID-XU4. Size of cache is 2 MB and it is 16-way set associative cache and size of a page is 4 KB. So, number of cache colors will be: n_cache = (2 X 1024 X 1024)/(16 X 4 X 1024), which is equal to 32 cache colors. So, our platform has 32 cache colors.

Task allocation to processors is widely studied in real-time systems domain. Generally, tasks can either be statically or dynamically allocated to processors. Global scheduling (dynamic allocation) require job migration which has significant overheads on multiprocessor architectures. With multiple levels of a private cache for each processing core, the penalty of inter-core task migration is high. Therefore, partitioned scheduling algorithms (static allocation) is preferred.

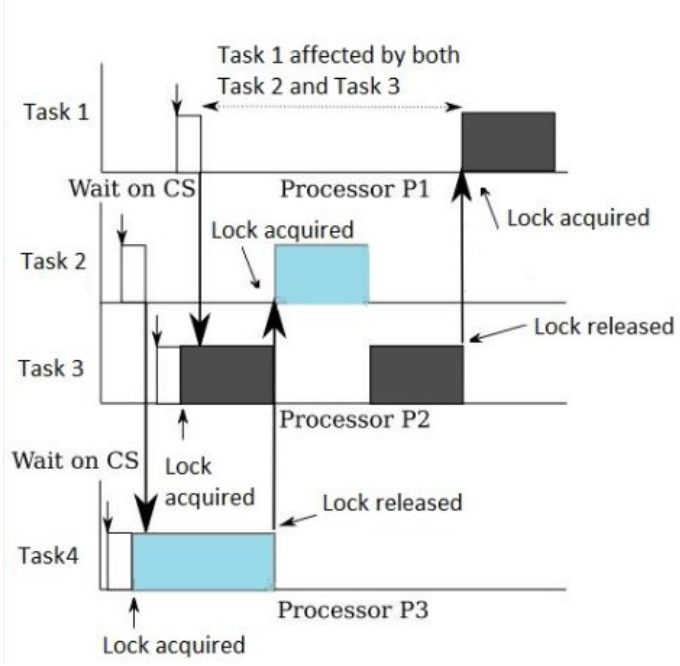

Figure 1.4: Transitive Interference due to remote blocking[17]

Multiprocessors encounters scheduling penalties due to task synchronization which are a) blocking delays on critical sections as task has to wait to acquire the lock b) transitive interference due to remote blocking c) back-to-back execution and d) multiple priority inversions

As we can see in Figure 1.4, task 1 on processor P1 is suspended on task 3 running on processor P2. Task 2 on processor P2 is suspended on task 4 running on processor P3. The priority ceiling of the mutex shared between task 2 and task 4 is higher than the priority-ceiling of the mutex shared between task 1 and task 3, as task 2 has higher priority than task 3. Now, the release of the lock by task 4 gives control to the task 2 and it acquires

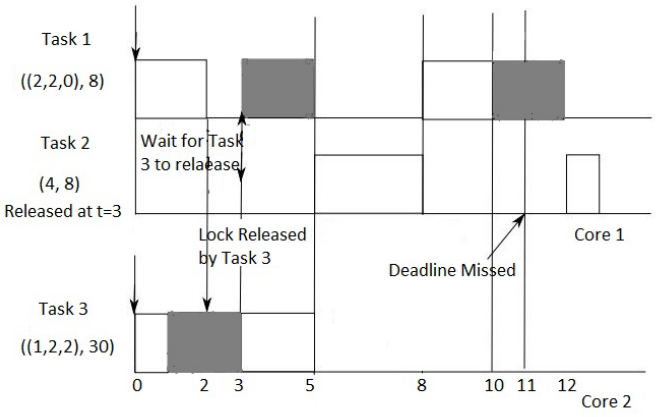

Figure 1.5: Back-to-Back execution[17]

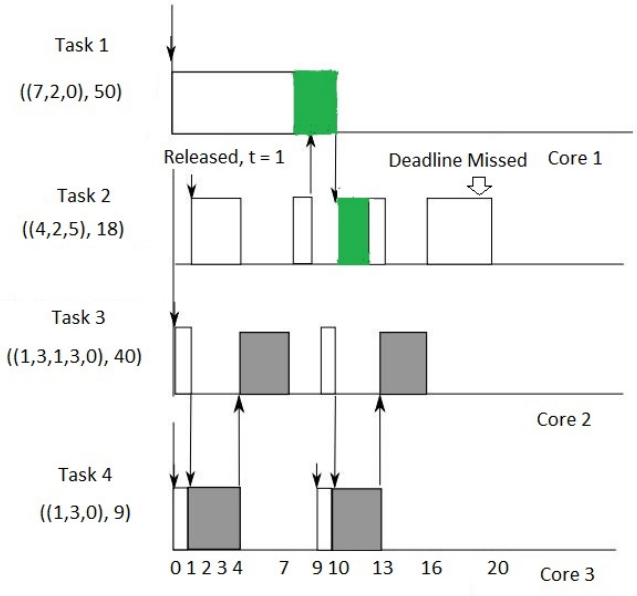

Figure 1.6: Multiple Priority Inversion[17]

the lock and executes its critical section, which preempts task 3 executing its critical section as it has lower priority than task 2. Therefore, task 1 that waits for task 3 to release the lock, suffers from an interference due to the release of a lock by task 4.

To understand back-to-back execution, let us take an example, refer Figure 1.5. From the figure, we can see that task 1 and task 3 are schedulable. However, task 2 misses its deadline. This happens because task 2 is released at t=3, and it faces back-to-back execution due to the remote synchronization effect of task 1.

To understand multiple priority inversion, let us consider an example, refer Figure 1.6. As soon as task 2 suspends, task 3 executes. Task 3 shares a critical section with task 4, so it requests a continuous lock on the shared global critical section until it acquires lock. As soon as task 4 releases the lock, task 3 acquires the lock preempting task 2 as it gets the higher priority because of the critical section. Task 3 interferes with the normal execution of task 2 and execute its critical section.

Previous work has addressed either the synchronization problem or the temporal interference problem. In this thesis, we proposed a framework which takes care of the cache interference as well as the task synchronization problems in multiprocessors by eliminating inter-core cache interference and bounding intra-core cache interference. We have used Linux/RK[3] environment on ODROID-XU4 board for the implementation and used Linux/RK APIs to create/destroy the resource environment allocated to a task:

1) *rk_resource_set_create*() creates a resource set of a given name and returns its descriptor to the user. For a task negotiation with resource kernel, the task have to create a resource environment first. The arguments for this API are the name of resource set, two flags; one for resource set inheritance and other for resource set cleaning up and policy for CPU reservations that is to be created in the resource set.

2) *rk_mem_reserve_create*() creates a memory reservation and attaches it to the resource set rd. The arguments for this API are resource set descriptor and attributes for memory reservation.

3) *rk_resource_set_destroy*() destroys the resource set that is passed as a parameter and the memory allocated to that resource set is freed. When a resource set is destroyed, the reserves contained in the resource set is automatically deleted. This API also removes the directory which corresponds to the resource set in the proc file system.

For shared memory and semaphores, Linux APIs used are:

1) *shmget*() returns the identifier of the shared memory segment associated with the value of the argument key. On success, a valid shared memory identifier is returned. On error, -1 is returned, and errno is set to indicate the error.

2) *shmat*() attaches the shared memory segment identified by shmid to the address space of the calling process. On success shmat() returns the address of the attached shared memory segment; on error -1 is returned, and errno is set to indicate the cause of the error.

3) *sem_open*() creates a new POSIX semaphore or opens an existing semaphore. The semaphore is identified by name.

4) *sem_wait*() decrements (locks) the semaphore pointed to by sem. If the semaphores value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either the semaphore value is greater than zero or a signal handler interrupts the call.

5) *sem_post*() increments (unlocks) the semaphore pointed to by sem. If the semaphores value consequently becomes greater than zero, then other process or thread blocked in sem_wait() call will be woken up and it proceeds to lock the semaphore.

6) *sem_close*() closes the named semaphore referred to by sem, allowing all the resources to be freed that was allocated to this semaphore.

Contributions: To summarize the work done in this thesis:

1) We proposed a framework which eliminates inter-core cache interference and bounds intra-core cache interference to addresses the synchronization penalties and cache interference for the critical sections accessing the shared memory region on multiprocessors.

2) We provide response time test to check task schedulability for our approach.

3) We proposed cache-aware task allocation algorithm for our approach

4) We evaluated our approach on Linux/RK platform on ODROID-XU4 board which is based on ARM architecture.

Chapter 2

Related Work

A lot of work has been done in both the areas of task synchronization and cache interference in multiprocessors and the research is still ongoing as both are widely studied. The task synchronization penalties in multiprocessors was addressed in [17] where authors characterized key synchronization penalties, developed and evaluated the synchronization-aware task allocation scheme which accommodated the synchronization penalties during task allocation. Multiprocessor synchronization challenges include both global and local synchronization. Priority Ceiling Protocol (PCP) is a uniprocessor real-time synchronization protocol i.e. local synchronization which has local mutex if many tasks on same processor accesses same memory location while Multiprocessor Priority Ceiling Protocol (MPCP) is a multiprocessor real-time synchronization protocol i.e. global synchronization which has global mutex if many tasks on different processors accesses same memory location.

Global synchronization was considered in [17] as the parameter for the schedulability analysis. In our work, we have considered both local and global synchronization for the response time test to check the task schedulability . Global and local synchronization was considered in virtual environment in [14].

To avoid the cache interference, software-based cache partitioning scheme *page coloring*, has been studied for multiprocessors in [7]. But two problems existed: (a) memory wastage and (b) limited number of cache partitions were available. Both of these problems were addressed in [11].

In prior work, synchronization penalties were addressed and cache interference were addressed separately. So, in this thesis, we have combined both things i.e task synchronization for multiprocessors taking into account page coloring with it. Our main focus is the tasks which have critical sections that accesses the shared memory region in multiprocessor systems. For that, we developed a framework which takes care of synchronization penalties as well as eliminates the inter-core cache interference and bounds the intra-core cache interference.

Chapter 3

System Model

For our work, we have taken into consideration a system that is equipped with a single-chip multi-core processor which has identical cores and are running at a fixed clock speed and has a unified last-level cache, which is shared among all the cores. We have considered page coloring scheme to manage the shared last level cache.

We have considered the systems which use partitioned fixed-priority preemptive task scheduling. We arrange tasks in strictly decreasing order of their priorities and assume each task to have a unique priority with n being the lowest priority of a task. For ex: if $a < b$, it implies that task τ_b has a lower priority than task τ_a.

We consider each task to be cache sensitive and an alternating sequence of normal execution segments and critical section execution segments. A task τ_i is characterized as follows:

$$\tau_i : ((C_{i,1}, C'_{i,1}, C_{i,2}, C'_{i,2}, ..., C'_{i,s_i} C_{i,s_i+1})^p, T_i, D_i) \tag{3.1}$$

where,

s_{i+1} : number of normal execution segments of τ_i

s_i : number of critical section execution segments of τ_i

$C_{i,j}$: WCET of the j^{th} normal execution of τ_i

$C'_{i,k}$: WCET of the k^{th} critical section of τ_i

T_i : period of task τ_i

D_i : deadline of task τ_i

$\tau_{i,j}$: j^{th} normal execution segment of task τ_i

$\tau'_{i,k}$: k^{th} critical section of task τ_i

C_i(for normal execution segment) and C'_i(for critical section execution segment) represent the worst-case execution time, when a task runs alone in system with 'p' cache partitions assigned to it.

$$C_i^p = \sum_{j=1}^{s_{i+1}} C_{i,j} + \sum_{k=1}^{s_i} C'_{i,k} \tag{3.2}$$

So, the worst case execution time of a task will be the sum of its both normal and critical execution segments.

Shared resources considered here are protected by suspension-based mutually-exclusive locks, i.e. the task suspends itself during remote blocking, thus enabling lower priority task to execute. There are two types of shared resources: 1) Global resources 2) Local resources. Global resources are those resources shared among tasks assigned to different CPUs whereas, local resources are resources shared among tasks assigned to same CPU. The critical sections corresponding to global and local resources are referred as global critical section (gcs) and local critical section (lcs) respectively.

$$s_i = s_i^{lcs} + s_i^{gcs} \tag{3.3}$$

where,

s_i : total number of critical section segments of τ_i

s_i^{lcs} : number of local critical section segments of τ_i

s_i^{gcs} : number of global critical section segments of τ_i

Chapter 4

Our Approach

For our framework, we are considering software cache partitioning scheme for the critical sections accessing the shared memory in multiprocessors. Our framework is taking care of synchronization penalties as well as eliminating inter-core and bounding intra-core cache interference. For this, we have proposed a cache-aware task allocation algorithm which is discussed later in the chapter. We will describe the schedulability analysis now.

4.1 Task Schedulability

Cache-related preemption delay (CRPD) is considered to bound intra-core cache interference penalties and it occurs when a task is preempted by a higher-priority task and is imposed on the the task which is preempted.

$$\gamma_{h,i} = \left| S(h) \cap \bigcup_{k \in mid(h,i)} S(k) \right| .\Delta \tag{4.1}$$

S(h): set of cache partitions assigned to τ_h

Δ : time to refill one cache partition

$\gamma_{h,i}$ is the CRPD caused by task τ_h and is imposed on the tasks that belongs to the set of tasks between the priorities of h and i i.e., whose priorities are lower than h and higher than or equal to i and share the cache partitions with task τ_h and is represented by *mid(h,i)* in equation 4.1

To determine the schedulability of task τ_i, we considered following factors:
(a) Local blocking Time (b) Remote blocking time (c) Back-to-back execution (d) Multiple priority inversions.

The Worst-case Response Time of task τ_i is represented as:

$$W_i^{n+1} = C_i^p + B_i^r + B_i^l + \sum_{h<i \;\&\; \tau_h \in P(\tau_i)} \left\lceil \frac{W_i^n + (W_h - C_h)}{T_h} \right\rceil .(C_h + \gamma_{h,i}) \tag{4.2}$$

and, $W_i^0 = C_i^p + B_i^r + B_i^l$

where, W_i^n is the worst case response time of task τ_i at nth iteration of the task. The schedulability test terminates when $W_i^{n+1} = W_i^n$. If the worst case response time is before the task's deadline, the task is schedulable. C_i^p is the summation of execution time of normal and critical segments of a task when p cache partitions are assigned to it, B_i^r is the remote blocking time of task τ_i and B_i^l is the local blocking time of task τ_i. P(τ_i) used in equation 4.2 returns the processor assigned to task τ_i.

Local Blocking Time

The lower-priority tasks can block the normal execution segment of a higher-priority task τ_i when the lower-priority task is executing its critical segment that can be either local or global. Local blocking time is represented as B_i^l:

$$B_i^l = (B_i^{l\_lcs} + B_i^{l\_gcs}).(s_i^{gcs} + 1) \tag{4.3}$$

where,

$B_i^{l\_lcs}$: Maximum per segment blocking time from local critical section of lower-priority tasks.

$B_i^{l\_gcs}$: Maximum per segment blocking time from global critical section of lower-priority tasks.

We multiply equation 4.3 with $(s_i^{gcs} + 1)$ because it might happen that before a task executes or whenever task self-suspends due to a global resource, the lower-priority tasks may issue requests for local or global resources.

$$B_i^{l\_lcs} = \max_{\substack{\tau_l \in P(\tau_i)\ \&\ l>i \\ \&\ S_l^{lcs}>0}} \left(\max_{\substack{1 \le u \le S_l\ \&\ type(\tau_l,u)=lcs \\ \&\ pc_{l,u}>\pi_i}} C'_{l,u} \right) + \gamma_{p,c_{l,u}} \tag{4.4}$$

where, π_i is the normal priority of τ_i

$$B_i^{l\_gcs} = \sum_{\substack{\tau_l \in P(\tau_i)\ \&\ l>i \\ \&\ S_l^{gcs}>0}} \left(\max_{1 \le u \le S_l \& type(\tau_l,u)=gcs} C'_{l,u} \right) \tag{4.5}$$

The function type(τ_i, j) returns lcs or gcs, which is the type of j^{th} critical section of task τ_i. We have used $pc_{i,j}$ in equation 4.4 as the priority ceiling of j^{th} critical section segment of task τ_i. In priority ceiling, each semaphore is assigned a priority ceiling equal to the highest priority of the tasks that can lock it. Then, a task τ_i is allowed to enter a critical section only if its priority is higher than all priority ceilings of the semaphores currently locked by tasks other than τ_i.

Remote Blocking Time

The remote blocking time B_i^r of a task is represented as below:

$$B_{i,j}^{r,n+1} = \max_{l \ge i\ \&\ (\tau'_{l,u}) \in R(\tau_i,j)} (C'_{l,u}) + \sum_{h<i\ \&\ (\tau'_{h,v}) \in R(\tau_i,j)} \left(\left\lceil \frac{B_{i,j}^{r,n}}{T_h} \right\rceil + 1 \right) . (C'_{h,v}) \tag{4.6}$$

where,

$$B_{i,j}^{r,0} = \max_{l \ge i\ \&\ (\tau'_{l,u}) \in R(\tau_i,j)} (C'_{l,u})$$

The function $R(\tau_i, j)$ returns the index of the resource used by j^{th} critical section of task τ_i or in other words this function is used to retrieve the locked resource corresponding to the j^{th} critical section segment of task τ_i.

The Worst-case Response Time of execution of global critical segment $C'_{i,k}$ is bounded by:

$$W'_{i,k} = C'_{i,k} + \sum_{\tau_u \in P(\tau_i)} \left(\max_{1 \leq v \leq S_u \ \& \ pc_{u,v} > pc_{i,k}} C'_{u,v} \right) \tag{4.7}$$

4.2 Algorithm Description

Previously, algorithm addressed the synchronization penalties on multiprocessors in [17] and assigned cache to tasks in [13].

When cache sharing is allowed, allocating cache colors to the tasks is a NP-hard problem, so we used a heuristic CachetoTaskAlloc[13]. Tasks having critical section and accessing the shared memory share a mutex (lock) are assigned 1 private cache partition i.e. that one partition is only accessed by the tasks that share a critical section. Similarly, rest of the shared critical sections are assigned a cache partition each. Then all the tasks are bundled together and then we try to allocate the bundle as a single task to a processor.

We considered enough processors that allocates the total utilization of all the tasks. All task bundles that fit are allocated without adding processors and the bundles which does not fit into the existing processor remains unallocated. The task bundles that do not fit the existing processor are arranged in increasing order of cost, and the bundle with the smallest cost is chosen to be broken by breakbundle algorithm [13] such that the utilization of the sub-bundle broken is less than or equal to 1. The cost here refers to the penalty of changing a local mutex to a global mutex. The bundle will be allocated according to the Best Fit Decreasing (BFD) heuristic [13]. BFD finds the best-fit processor that can schedule given bundle with 'k' additional cache colors assigned to it. If best-fit processor is found, the bundle is allocated to that processor and the number of colors of that processor and remaining cache colors are updated.

Algorithms:

Algorithm 1: Cache Aware Task Allocation($\Gamma, N_P, N_{Cache}, N_{rem}, N_{SV}$)**:**

Input: Γ: Taskset, N_P: Number of processors, N_{Cache}: Number of cache colors, N_{rem}: Number of remaining cache colors, N_{SV}: Number of shared variable

Output: Success or Fail

1: Check for N_{SV}.

2: Assign 1 private cache color starting from color 31 till 0 to each shared variable.

3: $N_{rem} \leftarrow N_{Cache} - N_{SV}$

4: /* Phase 1: Allocate task bundles */

5: $\varphi \leftarrow \Gamma; \Phi \leftarrow \emptyset$

6: **while** util(φ) > 1 **do**

7: $(\varphi^{'}, \varphi^{''}) \leftarrow BreakBundle(\varphi, 1, N_{rem})$

8: $\Phi \leftarrow \Phi \cup \{\varphi^{'}\}; \varphi \leftarrow \varphi^{'}$

9: $\Phi \leftarrow \Phi \cup \{\varphi\}$

10: **while** $\Phi \neq \emptyset$ **do**

11: /* Allocate bundles */

12: $\Phi_{rest} \leftarrow \emptyset$

13: **for all** $\varphi_i \in \Phi$ in decreasing order of utilization **do**

14: $(p_{BFD}, k) \leftarrow BestFitDecWithCache(\varphi_i, N_P, N_{rem})$

15: **if** $p_{BFD} \neq$ invalid **then**

16: $\Gamma_{BFD} \leftarrow \Gamma_{BFD} \cup \varphi_i$; $S^p_{BFD} \leftarrow S^p_{BFD} + k$; $N_{rem} \leftarrow N_{rem} - k$

17: **else**

18: $\Phi_{rest} \leftarrow \Phi_{rest} \cup \{\varphi_i\}$

19: /* Break unallocated bundles */

20: $\Phi \leftarrow \emptyset$; singletons $\leftarrow$ true

21: **for all** $\varphi_i \in \Phi_{rest}$ **do**

22: **if** $|\varphi_i| > 1$ **then**

23: singletons $\leftarrow$ false; size $\leftarrow$ 1-$min_{P_j \in P} util(\Gamma_j)$

24: $(\varphi^{'}, \varphi^{''}) \leftarrow BreakBundle(\varphi_i, size, N_{rem})$

25: $\Phi \leftarrow \Phi \cup \{\varphi^{'}, \varphi^{''}\}$

26: **else**

27: $\Phi \leftarrow \Phi \cup \{\varphi_i\}$

28: **if** singletons = true **then**

29: **return** Fail

Algorithm 2: BreakBundle($\varphi, size, N_{rem}$):

Input: φ: A bundle to be broken, size: Size constraint for the first sub-bundle, N_{rem}: Number of remaining cache colors

Output: $(\varphi^{'}, \varphi^{''})$: a tuple of sub-bundles

1: $\varphi^{'} \leftarrow \varphi$; $\varphi^{''} \leftarrow \emptyset$

2: **for all** $\tau \in \varphi$ in increasing order of number of a task sharing a variable **do**

3: $\varphi^{'} \leftarrow \varphi^{'} \backslash \tau_i$; $\varphi^{''} \leftarrow \varphi^{''} \cup \tau_i$

4: /* Get util($\varphi^{'}$) assuming each task uses one color */

5: **if** util($\varphi^{'}$) $\leq$ size **then**

6: **break**

7: **return**

Algorithm 3: BestFitDecWithCache(φ, P, N_{rem}):

Input: φ: A bundle of tasks to be allocated, P: A set of processors, N_{rem}: Number of cache colors

Output: (p_i, k): A tuple of the best-fit processor and the number of additional cache colors needed

1: **for** k $\leftarrow$ 0 to N_{rem} **do**

2: **for all** $p_i \in$ P in decreasing order of util(Γ_i) **do**

3: **if** CacheToTaskAlloc($\Gamma_i \cup \varphi, S_i^p$ + k) ≤ 1 **then**

4: **return** (p_i,k)

5: **return** (invalid,-1)

Algorithm 4: CacheToTaskAllocation(Γ, N_{rem}):

Input: Γ: Taskset, N_{rem}: Number of remaining cache colors

Output: Utilization of Γ if schedulable, and ∞ otherwise

1: **if** $N_{rem} = 0$ **then**

2: **return** ∞

3: cache_idx $\leftarrow 1$

4: **for all** $\tau_i \in \Gamma$ **do**

5: /* Find the number of cache colors for τ_i */

6: $S_i \leftarrow argmin_{1 \leq k \leq N_{rem}}(\frac{C_i(k)}{T_i} + \frac{\gamma_{i,n}}{T_i})$

7: /* Find cache-color indices for τ_i */

8: $M_i \leftarrow \emptyset$

9: **for** k $\leftarrow 1$ to S_i **do**

10: $M_i \leftarrow M_i \cup \{cache\_idx\}$

11: $\{cache\_idx\} \leftarrow (\{cache\_idx\} + 1) \bmod N_{rem}$

12: **if** schedulable(Γ) **then**

13: **return** util(Γ)

14: **else**

15: **return** ∞

Chapter 5

Evaluation

The Evaluation section consists of two case studies: 1) Case Study on Real Platform 2) Case Study of Schedulabilty Analysis.

5.1 Case Study on Real Platform

Experimental Setup

We used ODROID-XU4 board shown in Figure 5.1, an ARM platform for our experiments that has 2GB LPDDR memory (RAM) and a Exynos 5422 application processor. The System on a Chip (SoC) contains quad-core ARM Cortex-A7 and quad-core ARM Cortex-A15 CPUs. As the speed of Cortex-A15 cores were higher than the Cortex-A7 cores, we considered using Cortex-A15 cores for our experiments. The last level cache (L2) is of 2MB that is shared among four Cortex-A15 cores and provides 32 cache colors, with each cache color of size being 64KB. We also disabled dynamic clock frequency scaling

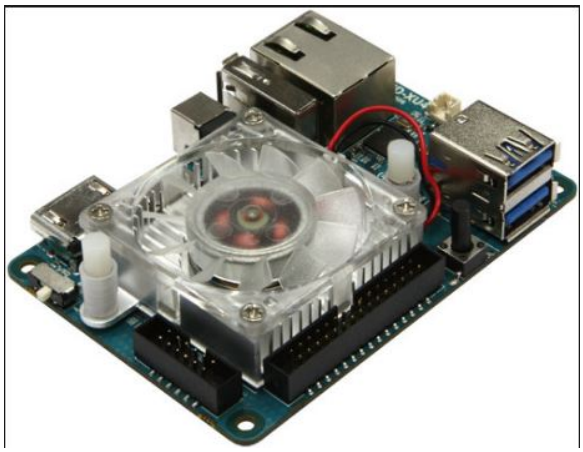

Figure 5.1: ODROID-XU4[4]

for our experiments. We used linux odroid 3.10.82-rk as guest OS on the board in our experimentation.

Experimental Results

For our experiment, we first checked whether the cache sensitive tasks which we created were correctly assigning colors or not. We used latency task [29] which traverses a randomly-ordered linked list. For latency task to be cache-sensitive, we considered the working set size (WSS) of the task to be 1MB i.e. half of the last level cache of our odroid platform. The execution time of the latency task is dependent on the memory access time because of the data dependency pointer-chasing operations in linked-list traversals. Figure 5.2 to Figure 5.5 shows the output of maximum observed execution times of the latency task when a latency task runs alone on a core with different numbers of cache colors assigned to it. We have shown the execution time of latency task for cache colors as 4, 8, 16 and 32 in this

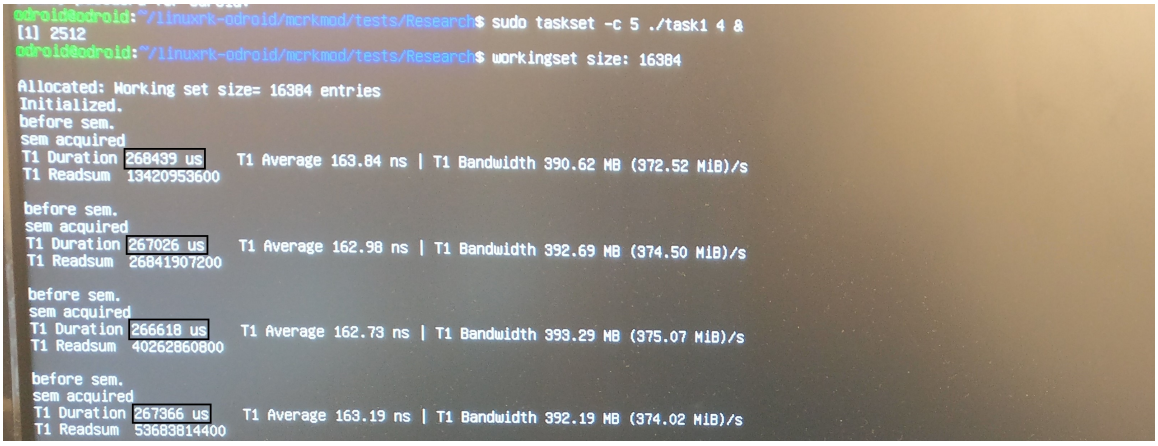

Figure 5.2: Execution Time when 4 cache color assigned

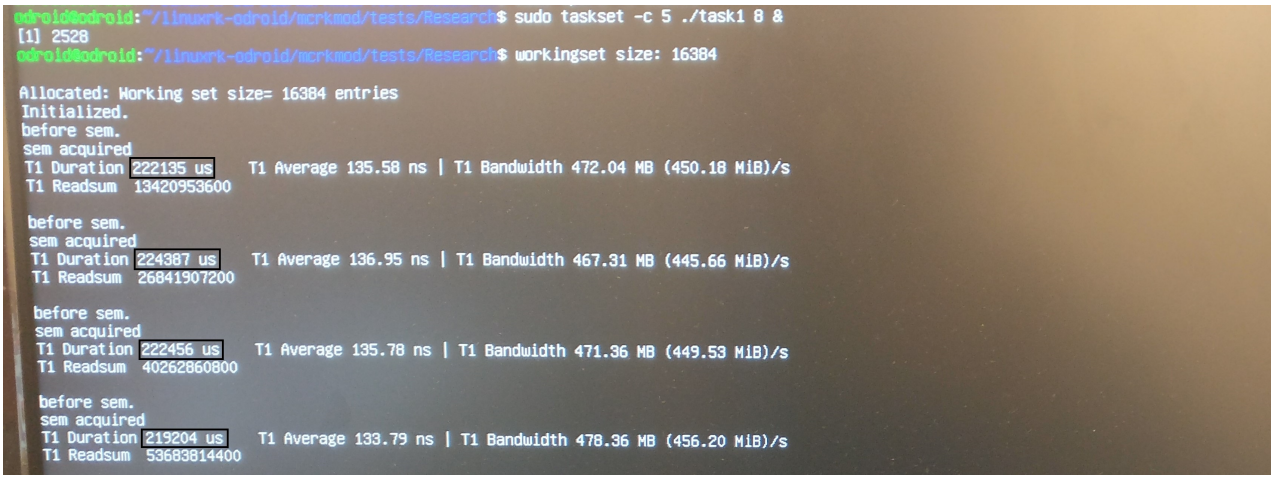

Figure 5.3: Execution Time when 8 cache color assigned

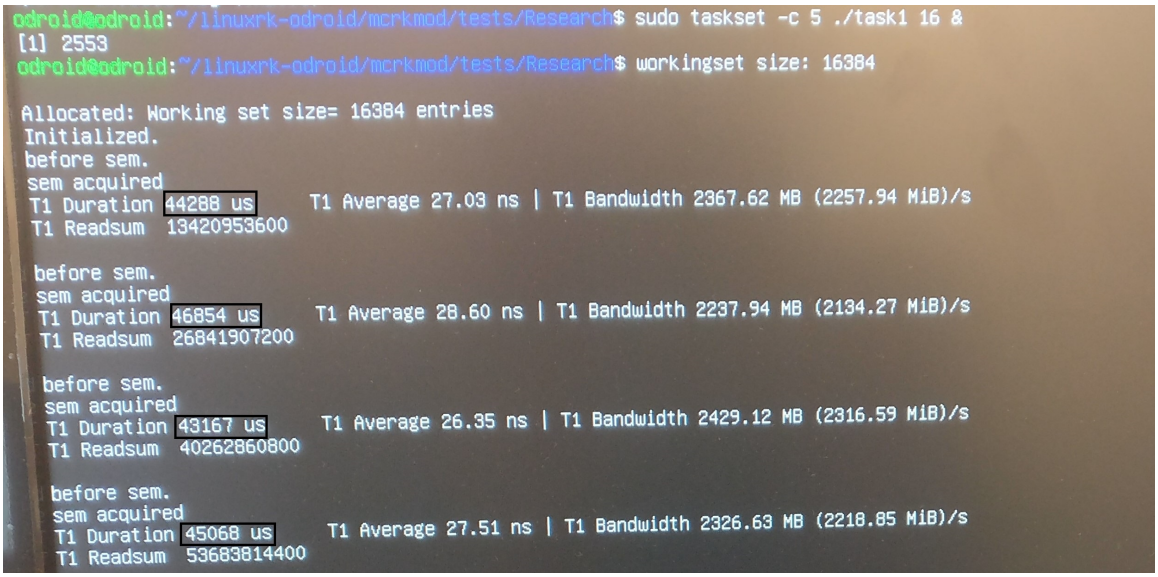

Figure 5.4: Execution Time when 16 cache color assigned

report. Figure 5.2 shows execution time when 4 cache colors are assigned; Figure 5.3 shows execution time when 8 cache colors are assigned; Figure 5.4 shows the execution time when 16 cache colors are assigned and Figure 5.5 shows execution time when 32 cache colors are assigned. We observed that the execution time of the task did not change much after 18 cache colors were assigned to it on ARM platform because the entire working set of the task was fitting in the last level cache (L2) after that point. Expected execution-time pattern is observed i.e. as we increase the number of cache colors, the execution time decrease and execution time becomes constant after a certain number of cache colors as shown in Figure 5.6.

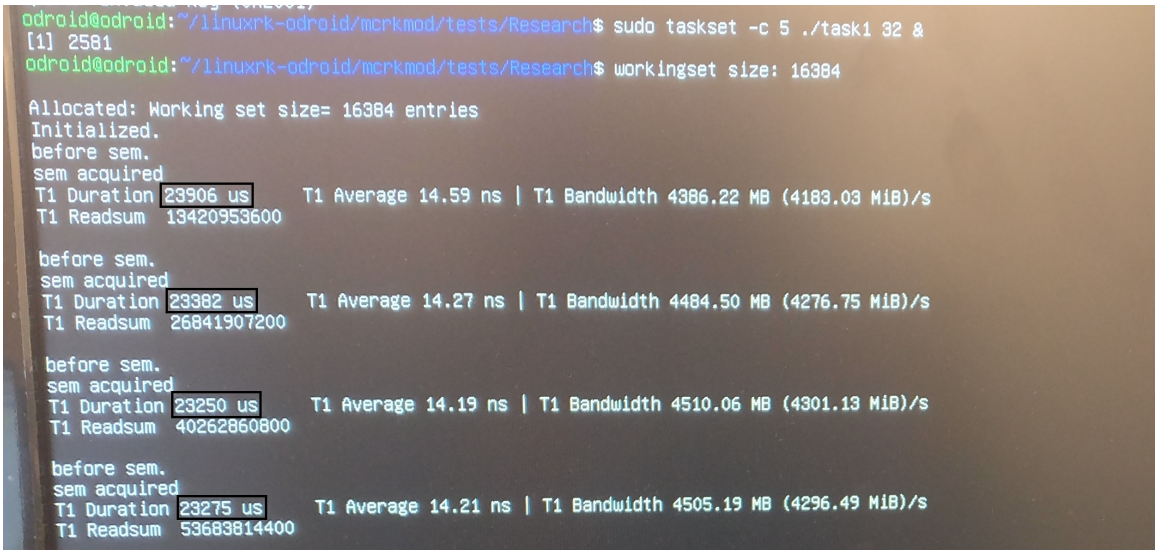

Figure 5.5: Execution Time when 32 cache color assigned

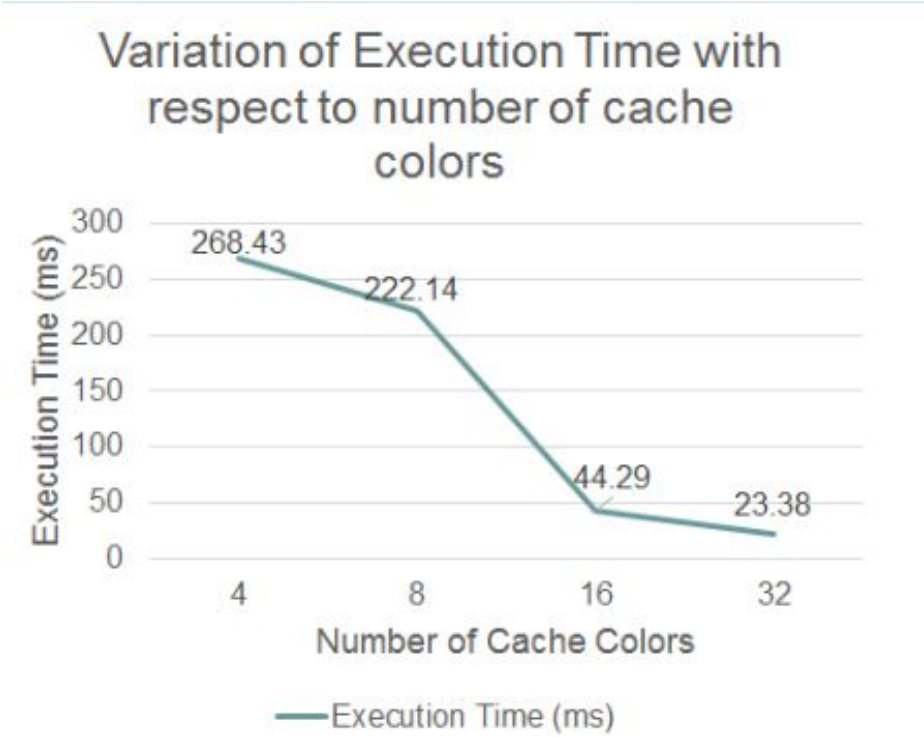

Figure 5.6: Variation of Execution Time with respect to number of cache color assigned

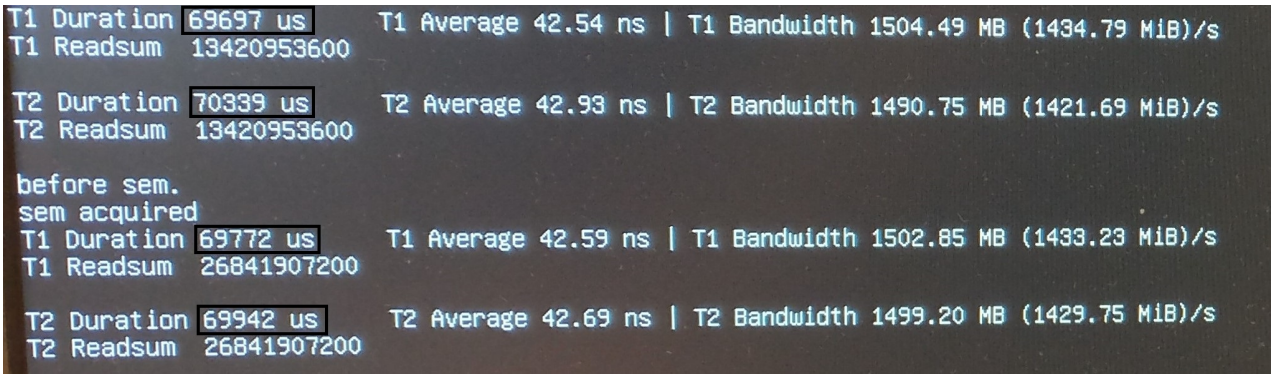

Figure 5.7: Tasks sharing memory region with interference

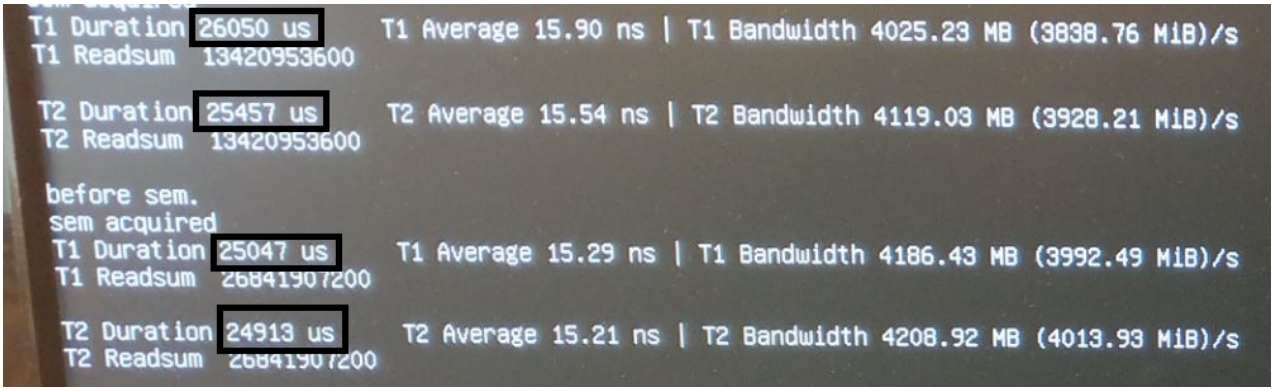

Figure 5.8: Tasks sharing memory region without interference

General Case: We have three tasks. First two tasks are cache sensitive tasks implemented by modifying the latency benchmark and third task is the latency benchmark[29] itself. First two tasks have a shared region which is protected by a lock, semaphore in our case. We are just taking the execution time of the critical section i.e we are not taking the waiting time to acquire the mutex into our consideration. In the shared region, both the tasks is traversing the linked list in random order and both the tasks are running on different A15 cores and we have assigned both the tasks 32 cache colors to share among themselves. We are running the third task in the background which accesses the same cache partitions assigned for sharing region of the two tasks which can happen in multiprocessors. As we can see from the Figure 5.7, that the execution time is almost thrice even if we run just one task assigned 32 cache colors on the third core (refer Figure 5.5 and Figure 5.7). This is because of the inter-core cache interference phenomenon.

Our Approach: To avoid the inter-core cache interference problem in the critical section accessing shared memory region described above, we came up with an approach to assign a cache partition to critical section of tasks accessing shared memory and use the remaining cache colors left after assigning cache color to all the shared region, to schedule the tasks. Figure 5.8 shows the two cache sensitive tasks implemented by modifying the latency benchmark same as general case but the difference is that the critical section is assigned private cache partition (no other tasks can access this region except the tasks sharing critical section). We are just taking the execution time of the critical section into our consideration. In the shared region, both the tasks is traversing the linked list in random order and both

the tasks are running on different A15 cores and we have assigned both the tasks 32 cache colors to share among themselves. As we can see from the figure that the execution time is almost the same if we run just one task assigned 32 cache colors on one core (refer Figure 5.5 and Figure 5.8). So, this shows that if we assign private cache partition to the critical section accessing shared region we can eliminate inter-core interference.

5.2 Case Study for Schedulability Analysis

We consider the following taskset example to compare the schedulability of our approach with one of the basic approaches i.e. fair-share scheduling.

In Table 5.1, we have considered that task τ_1 and task τ_3 share a critical section and task τ_2 and task τ_4 share a critical section. We have considered priority of the tasks from task τ_1 to task τ_4 i.e task τ_1 has the highest priority and task τ_4 has the lowest priority. 'p' in Table 5.1 represents number of cache partitions; the tasks are represented as sequence of normal and critical execution segment (2 normal segments and 1 critical segment); T_i and D_i represents the time period and the deadline of the task respectively. U is the utilization of the task.

According to our cache aware task allocation algorithm, task τ_1 and task τ_2 will be allocated to core 1 with 4 cache partitions, because the utilization of the core is less than 1 when we assign 4 cache partitions. Neither task τ_3 nor task τ_4 can be allocated to

Tasks	p	C_1^p	$C_1^{'p}$	C_2^p	$C_i = C_1^p + C_1^{'p} + C_2^p$	$T_i = D_i$	U
	1	30	42	28	100		0.454
	2	25	37	23	85		0.386
τ_1	4	10	21	9	40	220	0.181
	8	7	13	5	25		0.113
	16	5	10	3	18		0.08
	1	57	117	46	220		0.687
	2	52	107	41	200		0.625
τ_2	4	18	65	12	95	320	0.297
	8	15	35	10	60		0.188
	16	10	28	7	45		0.141
	1	53	113	34	200		0.80
	2	48	108	29	185		0.74
τ_3	4	33	93	14	140	250	0.56
	8	15	50	10	75		0.30
	16	12	40	8	60		0.24
	1	113	183	94	390		0.847
	2	101	171	83	355		0.771
τ_4	4	75	145	56	275	460	0.598
	8	25	60	10	95		0.206
	16	22	50	8	80		0.173

Table 5.1: Taskset Example for Schedulability Analysis

core 1 with 4 cache partitions as it will make utilization of the core greater than 1 which is not possible. Therefore, task τ_3 and task τ_4 are allocated to core 2 with 8 cache partitions assigned to it as the utilization will be greater than 1 if assigned lesser number of cache partitions than 8 on core 2.

All the four tasks are schedulable with our schedulability test with worst case response time as follows:

$\tau_1 \Rightarrow 220 \leq D_1$

$\tau_2 \Rightarrow 316 \leq D_2$

$\tau_3 \Rightarrow 237 \leq D_3$

$\tau_4 \Rightarrow 455 \leq D_4$

However, for fair-share scheduling, we find that out of four tasks only two tasks are schedulable, rest two tasks misses the deadline and are not schedulable.

Therefore, using our approach makes all the tasks schedulable while in fair-share scheduling all tasks are not schedulable. As our approach bounds the intra-core cache interference we get better results than the basic approach.

Chapter 6

Conclusions and Future Work

A. Conclusions

In this thesis, we present our proposed framework for cache management for critical sections accessing shared memory region in multiprocessors. Our framework has considered to address both synchronization penalties and cache interference by eliminating the inter-core cache interference and bounding intra-core cache interference. We analyzed the task schedulability and proposed cache-aware task allocation algorithm for our approach. Experimental results were taken on ARM platform which shows that our framework eliminates the inter-core cache interference and bounds intra-core cache interference. Also, we showed that for our approach, all the tasks were schedulable while it was not the case in fair-share scheduling. As our work is the first in this direction, so we can not compare our framework results with any existing framework.

B. Future Work

Currently, our experimental results are more like a case study, to check whether our framework eliminates inter-core and bounds intra-core cache interference for critical sections accessing shared memory region in multiprocessors. These experimental results can be further extended for randomly generated tasksets to check that our proposed framework acts in the expected way.

We have taken a very naive approach for cache allocation in this thesis. Suppose, it might happen that the critical section is too small, in that case assigning a cache partition would not be the best solution, as the small critical section would not cause significant interference. So, our approach can be further extended to include that as well.

Bibliography

[1] ARM Cortex-A15 technical reference manual. http://arm.com.

[2] Hardkernel ODROID-XU4 user manual. https://magazine.odroid.com/wp-content/uploads/odroid-xu4-user-manual.pdf.

[3] Linux/RK installation. http://rtml.ece.cmu.edu/redmine/projects/rk.

[4] ODROID-XU4 Image. https://www.cnx-software.com/2015/07/14/odroid-xu4-board-is-a-smaller-and-cheaper-version-of-odroid-xu3.

[5] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 101–110, Aug 2008.

[6] Giorgio C. Buttazzo. Hard Real-Time Computing Systems.

[7] S. Cho and L. Jin. Managing distributed, shared l2 caches through os-level page allocation. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 455–468, Dec 2006.

[8] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Cache-aware scheduling and analysis for multicores. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pages 245–254, New York, NY, USA, 2009. ACM.

[9] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[10] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154, April 2014.

[11] H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical os-level cache management in multi-core real-time systems. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 80–89, July 2013.

[12] H. Kim and R. Rajkumar. Shared-page management for improving the temporal isolation of memory reservations in resource kernels. In *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 310–319, Aug 2012.

[13] H. Kim and R. Rajkumar. Real-time cache management for multi-core virtualization. In *2016 International Conference on Embedded Software (EMSOFT)*, pages 1–10, Oct 2016.

[14] H. Kim, S. Wang, and R. Rajkumar. vmpcp: A synchronization framework for multi-core virtual machines. In *2014 IEEE Real-Time Systems Symposium*, pages 86–95, Dec 2014.

[15] Hyoseung Kim and Ragunathan (Raj) Rajkumar. Memory reservation and shared page management for real-time systems. *Journal of Systems Architecture*, 60(2):165 – 178, 2014.

[16] Hyoseung Kim and Ragunathan (Raj) Rajkumar. Predictable shared cache management for multi-core real-time virtualization. *ACM Trans. Embed. Comput. Syst.*, 17(1):22:1–22:27, December 2017.

[17] K. Lakshmanan, D. d. Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *2009 30th IEEE Real-Time Systems Symposium*, pages 469–478, Dec 2009.

[18] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *[1989] Proceedings. Real-Time Systems Symposium*, pages 166–171, Dec 1989.

[19] J. Liedtke, H. Hartig, and M. Hohmuth. Os-controlled cache predictability for real-time systems. In *Proceedings Third IEEE Real-Time Technology and Applications Symposium*, pages 213–224, Jun 1997.

[20] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 367–378, Feb 2008.

[21] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54, April 2013.

[22] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings.,10th International Conference on Distributed Computing Systems*, pages 116–123, May 1990.

[23] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sep 1990.

[24] L. Sha, R. Rajkumar, S. H. Son, and C. H. Chang. A real-time locking protocol. *IEEE Transactions on Computers*, 40(7):793–800, Jul 1991.

[25] Noriaki Suzuki, Hyoseung Kim, Dionisio de Niz, Bjorn Andersson, Lutz Wrage, Mark Klein, and Ragunathan (Raj) Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *Proceedings of the 2013 IEEE 16th International Conference on Computational Science and Engineering*, CSE '13, pages 685–692, Washington, DC, USA, 2013. IEEE Computer Society.

[26] D. Thiebaut, J. L. Wolf, and H. S. Stone. Synthetic traces for trace-driven simulation of cache memories. *IEEE Transactions on Computers*, 41(4):388–410, Apr 1992.

[27] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson. Outstanding paper award: Making shared caches more predictable on multicore platforms. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 157–167, July 2013.

[28] Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: A dynamic cache partitioning system using page coloring. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 381–392, Aug 2014.

[29] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, April 2014.