Memory Aware Scheduling in Mixed Criticality Systems

Ankita Samaddar

To my family and my supervisor

CERTIFICATE

This is to certify that the dissertation titled **"Memory Aware Scheduling in Mixed Criticality Systems"** submitted by **Ankita Samaddar** to Indian Statistical Institute, Kolkata, in partial fulfillment for the award of the degree of **Master of Technology in Computer Science** is a bonafide record of work carried out by her under my supervision and guidance. The dissertation has fulfilled all the requirements as per the regulations of this institute and, in my opinion, has reached the standard needed for submission.

Ansuman Banerjee
Associate Professor,
Advanced Computing and Microelectronics Unit,
Indian Statistical Institute,
Kolkata-700108, INDIA.

Acknowledgments

I would like to show my highest gratitude to my advisor, *Ansuman Banerjee*, Advanced Computing and Microelectronics Unit, Indian Statistical Institute, Kolkata, for his guidance and continuous support and encouragement. He has literally taught me how to do good research, and motivated me with great insights and innovative ideas.

My deepest thanks to all the teachers of Indian Statistical Institute, for their valuable suggestions and discussions which added an important dimension to my research work.

Finally, I am very much thankful to my parents and family for their everlasting support.

Last but not the least, I would like to thank all of my friends for their help and support. I thank all those, whom I have missed out from the above list.

Ankita Samaddar
Indian Statistical Institute
Kolkata - 700108 , India.

Abstract

Mixed criticality systems integrate tasks of different criticality levels on the same platform. In order to ensure safety of such systems, we need to guarantee that all critical tasks must complete their execution prior to their deadlines. In normal architectural platforms, DRAM controllers generally serve the memory requests on an open page policy. So DRAM controllers that are used in normal architecture cannot serve all the memory requests and hence all the tasks often cannot meet their deadlines. In this work, we propose a novel approach for bank aware memory allocation of tasks which can significantly improve the performance of these mixed criticality systems. Experimental results on different benchmarks show the efficacy of our proposed scheme.

Keywords: *Mixed criticality system, schedulabity analysis, graph partitioning, constrained optimization.*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A real time system is any information processing system which has to respond to an externally generated input stimuli within a finite and specified period. Correctness of a real time system depends not only on the logical result of input but also on the time at which the result is produced. Different real time embedded systems include automobile systems, avionics systems and many others. Real time embedded systems which integrate tasks of different criticality levels on the same hardware platform are more commonly known as mixed criticality systems. In these systems, applications belonging to different criticality levels are engineered to different levels of assurance where high criticality applications are the costliest to design. Criticality level denotes the assurance against failure needed for a system component [3].

A task $\tau_i = (A_i, D_i, C_i, E_i^j)$ in a mixed criticality system is defined as follows:

- $A_i \in N$ denotes the arrival time,
- $D_i \in N$ and $D_i > A_i$ denotes the deadline,
- C_i denotes the set of criticality levels,
- $E_i^j \in R$ denotes its execution time / memory budget at the j^{th} criticality level.

A mixed-criticality system (MCS) consists of tasks of two or more distinct levels of criticality. The main objective of mixed-criticality systems is to guarantee the safety of the system by increasing the number of execution of critical tasks. A mixed critical task has many execution modes and each mode is associated with different budgets for execution. Execution budget means resource requirement of a task in a particular execution mode. A task can switch from one mode to another mode if required. For example, a critical task generally executes in its lowest execution mode with minimum budget of execution in that mode. But it may switch to higher modes during execution with larger number of resources

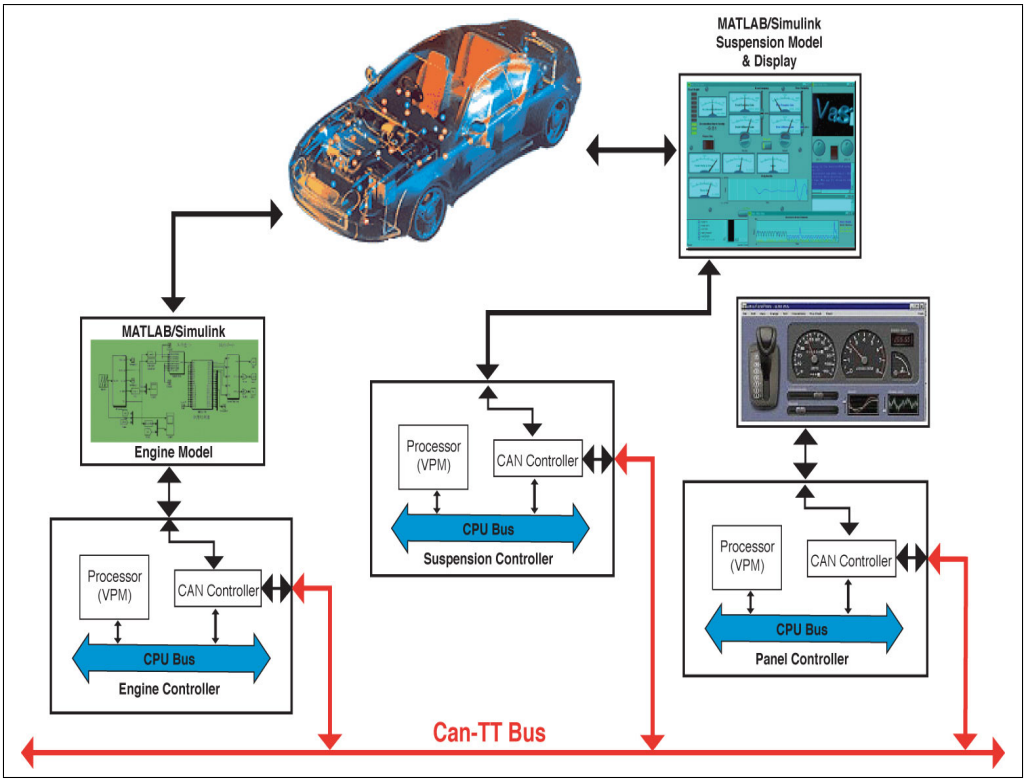

Figure 1.1: Overview of an automotive system

so that it may not miss its deadline. Scheduling critical tasks on a mixed criticality platform is quite challenging. We need to guarantee that all the critical tasks are to be scheduled and executed within their deadlines.

A popular example of a mixed criticality system is an automotive system. Automotive embedded systems contain a mix of safety critical control tasks (needed for system stability and safety) and time critical tasks with deadline constraints (needed for ensuring system performance and behavior). Safety critical tasks include tasks for controlling vehicle dynamics, air-bag control which are crucial for ensuring the safety of the vehicle. On the other hand, tasks associated with stringent timing constraints like tasks for driver-assistance, help improve usability and driving comfort. Some tasks may also exhibit both timing and safety critical behavior. With the consolidation of functionality on the same hardware, applications with both classes of tasks may co-exist, interact and share common hardware platform resources like Electronic Control Units (ECUs), and buses. Scheduling and platform design to ensure such a mix of timing, control performance and stability constraints is a challenging problem. Fig1.1 gives an overview of an automotive system.

In this type of mixed criticality systems, each task generally constitutes multiple instructions, some of which are compute intensive instructions, while others are memory intensive.

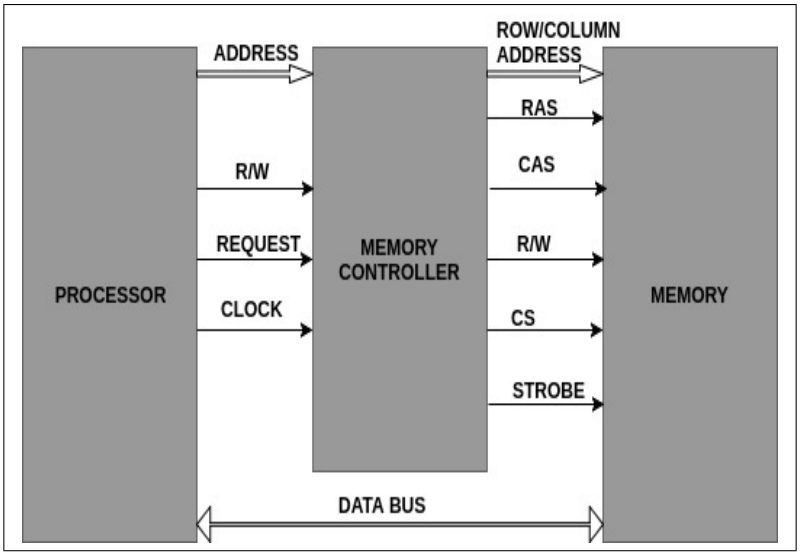

Figure 1.2: Schematic diagram of memory architecture

Thus, memory intensive tasks require frequent memory accesses. But in our existing architecture, instructions cannot be served instantaneously as and when they arrive. A schematic diagram of a contemporary memory architecture is shown in Fig1.2. In existing computer architecture, we can schedule requests either at the processor or at the memory or more specifically the DRAM controller. Several scheduling algorithms have been proposed to schedule the tasks at the processor level. Uniprocessor schedulers mostly use Earliest Deadline First (EDF) [20] or Rate Monotonic Scheduling (RMS) [22] to schedule tasks at the processor. Scheduling of tasks in multi-core systems has also been proposed in [7]. Current DRAM controllers take the advantage of row buffer locality. They follow an open row policy to schedule requests to access memory. According to the open row policy, the instruction which results in row hit is served first. Then the older requests are served in the order of their arrival. In this dissertation, we focus on the scheduling of memory requests at the DRAM so that maximum number of high critical tasks get executed.

1.1 Motivation of this dissertation

In this dissertation, we study the problem of memory scheduling for mixed criticality systems. We first take up the problem of deciding the required number of banks that can ensure a given set of mixed criticality tasks can meet their deadlines. We study both the decision and optimization problems for the same. Following this, we take up the issue of bank scheduling. The major problem with the open row policy at the DRAM is that most of the high criticality tasks fail to meet their deadlines while waiting for memory access if not scheduled and executed within their deadlines. Due to this open row policy, some

low criticality tasks get prioritized to be executed first whereas, some high criticality tasks may miss their deadlines while waiting in the buffer for memory access. Thus, selection of tasks is a very important criteria for scheduling tasks of different criticality levels executing on such a platform in order to ensure safety of the system. In this dissertation, we have proposed a novel method to schedule tasks around memory which increases the number of execution of high criticality tasks significantly. A number of proposals have been made for scheduling tasks of different criticality levels around memory. The fundamental difference of these methods with our proposed method is that in our proposed method, we ensure that no high criticality task will miss their deadlines at the cost of any low criticality task. This is the main motivation of this dissertation. The main contribution of this dissertation is highlighted below.

1.2 Contribution of this dissertation

Our method proposes a bank aware memory scheduling policy to schedule tasks of different criticality levels across memory banks. We have partitioned the tasks across banks according to their address mapping and then we have refined our existing partitions by a heuristic partitioning algorithm which uses a cost function based on some task parameters and some system parameters as a metric for partitioning. Our proposed method has been compared with existing state-of-the-art memory controllers on different benchmarks of Malardalen Worst Case Execution Time Benchmark [9]. Results on different benchmarks show the efficiency of our proposed method.

1.3 Organization of this dissertation

The rest of the dissertation is organized into 5 chapters. A summary of the contents of the chapters is as follows:

Chapter 2: A detailed study of relevant research is presented here.

Chapter 3: This chapter deals with design specifications to support this type of mixed criticality system.

Chapter 4: This chapter addresses the inefficiency of existing DRAMs to support this type of mixed criticality system and our contribution to overcome this inefficiency.

Chapter 5: We summarise with conclusions on the contribution of our dissertation.

Chapter 2

Background and Related Works

In this chapter, we first present a few background concepts needed for developing the foundation of our proposed work. We also present an overview of different methods proposed in the literature of scheduling around DRAM.

2.1 Background

In this section, we discuss a few background concepts.

2.1.1 Organization of DRAM

Main memory is stored in DRAM cells that have much higher storage density. DRAM chips are large, rectangular arrays of memory cells with support logic that is used for reading and writing data in the arrays, and refresh circuitry to maintain the integrity of stored data [19]. According to storage oraganization, memory is hierarchically organized into DIMM, rank, bank and array. DRAM devices are composed of basic blocks of DRAM memory. Multiple DRAM devices are accessed in parallel which together forms a DRAM rank [18]. DRAM devices in the same rank share same bus for address and command and another bus for data communication. Each DRAM device is made up of multiple DRAM arrays which store the data. Each of these arrays can be accessed independently. A group of DRAM arrays in different DRAM devices that are accessed together forms a DRAM bank. A DRAM bank is a 2D array of cells: rows x columns. A DRAM row is also called a DRAM page. Memory arrays are arranged in rows and columns of memory cells called wordlines and bitlines, respectively. Each memory cell has a unique location or address defined by the intersection of a row and a column. A DRAM cell stores a bit in a capacitor and hence they are needed to be charged periodically to prevent loss of data. Fig.2.1 shows the organisation of DRAM.

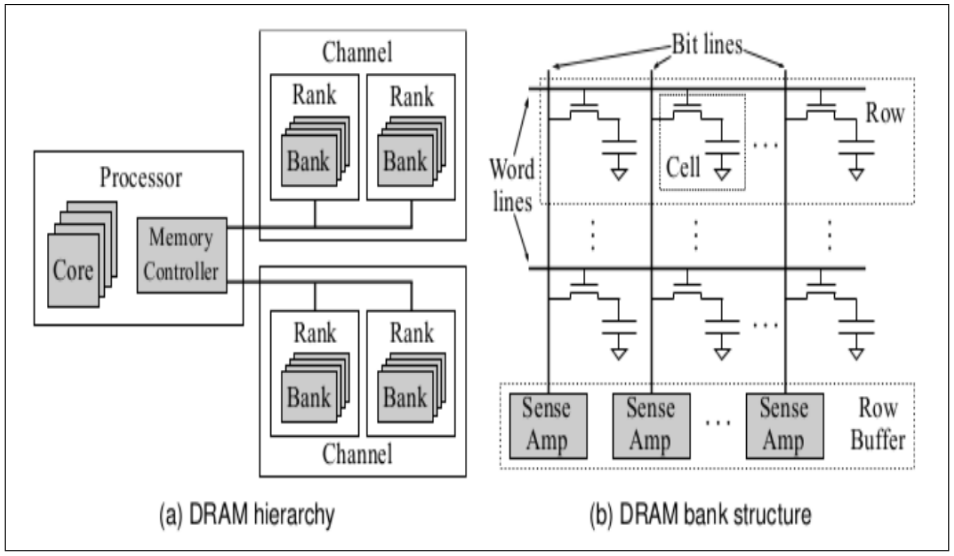

Figure 2.1: Organisation of DRAM

2.1.2 DRAM access commands

A number of DRAM commands such as PRECHARGE, ACTIVATE, READ, WRITE, REFRESH and IDLE are used to access DRAM cells for different DRAM operations. Fig.2.2 shows a brief overview of the state transition of a DRAM. There are two types of transitions in a DRAM, transition due to DRAM commands and transition triggered by time elapse. The curved arrows represent the transitions caused due to elapse of time and the straight arrows represent the transitions caused by issue of DRAM commands. PRECHARGE command precharges the bit lines and a new row is set ready to be accessed. PRECHARGE command is issued before accessing a new row. After fully precharging the bit lines, the bank goes back to Idle state. ACTIVATE command opens a row in a bank for access. The data is transfered from DRAM cells to the row buffer. While in Active state, a READ or a WRITE command may be issued. The same row remains active until a PRECHARGE. READ command initiates a burst read from an active row in a bank. WRITE command initiates a burst write to an active row in a bank. REFRESH command refreshes the target bank and rows within the DRAM and prevents decaying of data. So it is necessary to issue REFRESH command at regular intervals in order to prevent data loss in DRAMs.

2.1.3 Row Buffer Management Policy in DRAM

In DRAM devices, arrays of sense amplifiers act as buffers that provide temporary data storage. Policies that manage the operation of sense amplifiers are known as row buffer management policies. In ordinary DRAM devices, open-page policy and close-page policy are mainly used.

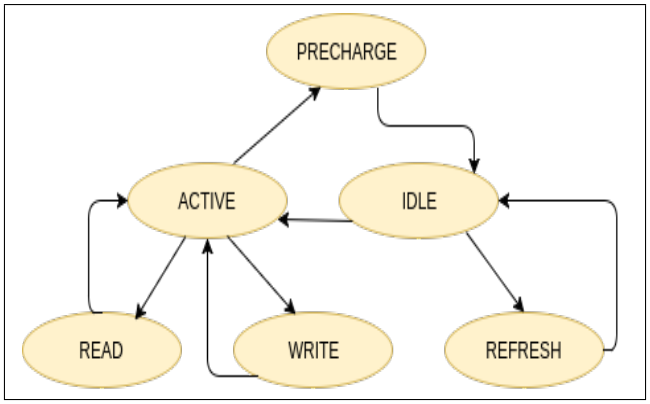

Figure 2.2: Simplified overview of important states of a DRAM

- Open-Page policy: The open-page row buffer management policy usually favors memory accesses to the same row of memory by keeping the sense amplifiers open and holding the data for ready access. Whenever a row of data is brought to the array of sense amplifiers in a DRAM cell, different columns of same row can be accessed again and again having only column access latency. Whenever a different row of the same bank needs to be accessed, the memory controller first precharges the DRAM array, then activates the desired row and finally allows for column access. This policy is best suited for sequential memory accesses and increases performance by better exploiting spatial and temporal locality in memory.

- Close-Page policy: The close-page row buffer management policy works better when we have random memory accesses across different rows. This policy precharges the row after every memory access. So we can keep the bank in Idle state after every row access in order to avoid the precharging overhead.

2.1.4 Different timing constraints between DRAM commands

Minimum delay must be maintained between two successive DRAM commands to ensure correct operation of a DRAM. Two types of timing constraints exist in DRAM:

- Intra Bank Timing constraints - When two successive commands access the same bank, some amount of delay must be maintained. However, these delays vary from device to device. Some DRAM timing constraints which are relevant to our implementation are:

  – Row Precharge time- time to precharge a complete row. It is actually the minimum delay between a PRECHARGE command and a subsequent command in the same bank.

  - Row Access time- minimum time interval between ACTIVATE and onset of next PRECHARGE command in the same bank.
  - Row to Column delay- time interval between ACTIVATE and a subsequent READ or WRITE command in the same bank.
  - Column Access delay- time required to access the columns of a particular row in the row buffer.
  - Row Cycle time- minimum interval to access different rows in the same bank in memory.

- Inter Bank Timing constraints- DRAM arrays for different banks are in the same DRAM device. Thus, hardware resources are sharable between banks. Address, commands and even data buses are shared by the ranks. Therefore, timing constraints are to be satisfied to avoid conflicts in hardware resources. Some inter-bank timing constraints are:

  - Row activation to Row activation delay- time interval between two ACTIVATEs to different banks.
  - Data Burst duration- busy period of the data bus.

2.1.5 Mixed Criticality Systems

Criticality is a designation of the level of assurance against failure needed for a system component. A mixed criticality system (MCS) [5] is one that has two or more distinct levels such as safety critical, mission critical or non critical etc. A safety-critical system [23] is a system whose failure or malfunction may result in death or serious injury to people, loss or severe damage to equipment/property or environmental harm. It is to be ensured that in order to maintain the safety of the system, a critical task should not miss its deadline.

A key aspect of Mixed-Criticality Systems is that a task can execute with different resource requirements in different criticality modes. The resource required to execute a critical task in its highest criticality mode is maximum, whereas, a task can execute with smaller amount of resources in lower criticality modes. Based on the availability of resources, a task may execute in multiple criticality levels in different cycles. A task executing in higher criticality mode with higher number of resources results in higher percentage of task completion. So, a task executing in higher criticality mode requires less number of cycles to complete the task. More critical a task is, more is the number of modes in which it can execute.

2.1.6 Scheduling and Schedulibility Analysis

The term scheduling analysis in real-time computing includes the analysis and testing of the scheduler system and the algorithms used in real-time applications. In computer science,

real-time scheduling analysis is the evaluation, testing and verification of the scheduling system and the algorithms used in real-time operations. For critical operations, a real-time system must be tested and verified for performance [24].

Real-time systems have timing requirements that must be guaranteed. Scheduling and schedulability analysis enable these guarantees to be provided. In scheduling theory, a real-time system comprises a set of real-time tasks; each task consists of an infinite or finite stream of jobs. The task set can be scheduled by a number of policies including fixed priority or dynamic priority algorithms. The success of a real-time system depends on whether all the jobs of all the tasks can be guaranteed to complete their executions before their deadlines. If they can, then we say the task set is schedulable [26].

2.2 Related research

Several techniques have been developed and adopted in real time predictable DRAM controllers. In normal systems, multiple requesters generate memory requests to the DRAM controller, which finally schedules the requests to the DRAM for processing the requests. In real time systems, each task is associated with a deadline which is expressed in terms of real time. Shared resource access interference, such as memory and system bus, is very challenging for designing predictable real time system as WCET of a real time task significantly differs between the different criticality levels. In commercial off-the-shelf (COTS) DRAM controllers, scheduling techniques are generally applied at the software level. In custom memory controller, either techniques focus mainly on scheduling the memory requests or controlling the commands. So based on our requirements, we need to decide on the suitable address mapping scheme and page policy to be used.

In COTS multicore systems, DRAM banks can be accessed independently. In [25], PALLOC, a DRAM bank aware memory controller has been proposed. These memory controllers exploit the page-based virtual memory system to avoid bank sharing among cores, thereby improving isolation on COTS multicore platforms without requiring any special hardware support. In [12], techniques have been proposed to provide a tight upper bound on the worst-case memory interference in COTS multi-core systems, where a task running on one core may get delayed by other task running on the other core due to shared resources. They have explicitly modeled the major resources in the DRAM system and considered timing characteristics by analyzing worst-case interference delay imposed by a parallelly running task on the other.

A predictable DRAM controller design has been proposed in PRET [17], where DRAM is considered as multiple resources that can be shared between one or more requests individually by interleaving accesses to blocks of DRAM. Thus contention for shared resources is eliminated within the device, making accesses temporally predictable and temporally isolated. Each memory request is scheduled by Time Division Multiplexing (TDM) and each

DRAM command's latency is predefined, so each request is isolated and tightly bounded. Though this scheme is very useful for critical tasks, unused memory slots cannot be used for non critical tasks and hence is very inefficient for those systems with large number of non critical tasks.

Another approach has been mentioned in [1] where bank interleaving and a close page policy are used with a pre-defined command sequence. Different scheduling approaches such as dynamic scheduling can be used for systems where we have varying memory request patterns. But request isolation is not gauranted. Again in [2], a Credit-Controlled Static-Priority (CCSP) has been suggested to provide minimum bandwidth for each request with bounded latencies.

Multicore processors handle hard real time systems for their good performance-watt-ratio and high performance capabilities. Unfortunately, multicores are limited by the gaurantee of time composability for mixed critical applications as WCET of a task depends on inter-task interferences with other tasks executing simultaneously on the same platform. In [15], an analytical model that computes worst case delay, more specifically known as Upper Bound Delay (UBD) has been computed considering all memory interferences generated by co-running tasks. Another approach has been proposed in [8] where a method for composable service to memory clients by composable memory patterns has been designed. A reconfigurable TDM, which can be changed at run time along with a reconfigurable protocol has been developed, whereas, predictable and composable performance is also offered to active memory clients which remain unaffected irrespective of configuration. Each request is isolated from memory interference if each task is allocated a slot in the TDM. But, a lot of slots are wasted if no memory request occurs resulting in decrease in throughput. Also, in absence of critical tasks, no non critical tasks are allowed to execute in the slot assigned for critical task.

Another approach has been developed in [11] where memory access groups (MAGs) are generated per bank and the tasks are combined to form these MAGs. Both critical and non critical MAGs are generated and each bank has certain critical space for execution of critical tasks. In absence of critical tasks, non critical tasks can execute in their place and they are pre-empted by critical tasks as soon as they arrive. The main disadvantage of this method is that each of critical MAG consists of one safety critical and two or more mission critical tasks. Therefore, some mission critical tasks may miss their deadlines while waiting in the MAG for memory access. For systems, where the ratio of safety critical and mission critical tasks are same, a lot of safety critical tasks miss their deadline.

In [10], memory requests are scheduled using time-division-multiplexing scheduler and a framework has been developed to statically analyse the tasks to meet the timing requirements of all tasks. This work proposes a mixed page policy that dynamically switches between close and open-page policies based on the request size to combine the benefits of both policies while avoiding their drawbacks. But in this method, many slots remain unused as requests are served in an interleaved manner and requests are served in a TDM manner.

Our bank partitioning problem and the solutions presented therein are based on the power mode partitioning problem discussed in [16] and the tenant mode allocation problem discussed in [13, 14]. However, in this work, we have adopted the same model in the context of designing efficient policies for DRAM controllers, while also deriving the hardness results. This gives us some novelty over what exists in literature.

Chapter 3

Bank Specification for serving memory requests

A mixed criticality system consists of tasks from different criticality levels working on the same platform. A task may be of certain criticality level based on the safety of the system. Upto 5 levels of criticality levels are identified in automobile/avionics systems. For an automobile/avionics system, our goal is to ensure that no high critical task should miss their deadlines at the cost of any low critical task.

Let us consider an automobile system consisting of engine, fuel system, exhaust system, cooling system, lubrication system, electrical system, transmission system, air-bag control system and the chassis. The chassis includes the brakes, tires, wheels, the suspension system and the body. When the system is in operating state, we have multiple messages coming from different applications. But some applications like the air-bag control system operates during emergency conditions to keep the system safe. So these messages are of higher criticality in order to ensure the safety of the system.

We have considered a task as a set of memory requests. Each task can be executed in any criticality mode. Our task model is based on the power mode partitioning problem discussed in [16] and the tenant mode allocation problem discussed in [13, 14]. In this chapter, we will mainly focus on the memory bank design and memory requirements so that we can serve the memory requests of different critical tasks executing at different criticality levels.

3.1 Memory Bank Requirements

Consider the memory hierarchy explained in the introductory chapters. The number of memory banks is unchangeable and needs to be specified at design time.

Task ID	Criticality Levels	Parallel Bank Access	% of task executed	Deadline
1	L_{11}, L_{12}, L_{13}	L_{11} - 6 L_{12} - 10 L_{13} - 0	L_{11} - 25% L_{12} - 35% L_{13} - 0%	4 cycles
2	L_{21}, L_{22}, L_{23}	L_{21} - 5 L_{22} - 10 L_{23} - 0	L_{21} - 20% L_{22} - 30% L_{23} - 0%	5 cycles
3	L_{31}, L_{32}, L_{33}, L_{34}	L_{31} - 0 L_{32} - 4 L_{33} - 8 L_{34} - 12	L_{31} - 0% L_{32} - 15% L_{33} - 25% L_{34} - 40%	6 cycles

Table 3.1: Task Specifications

3.1.1 Problem Formulation

Problem 3.1.1 *Given a set of mixed critical tasks with different deadlines and with different memory requirements at different criticality levels, the problem is to determine if the set of tasks can be served with a given number of memory banks such that all tasks finish execution within their deadlines.*

The above problem is introduced here with the help of an example.

Motivating Example

Consider a set of tasks with some memory requests that are needed to be served within their deadlines. Each task is associated with a set of critical levels or modes in which the task can be executed, along with the number of parallel bank accesses required at a particular criticality level or mode and the percentage of task which gets completed on serving the memory request at that criticality level or mode. A set of tasks with different task parameters are given in Table 3.1. All tasks are assumed to have arrived at the same instant for memory access. We need to check whether we can serve all the memory requests at any criticality level within their deadlines on a memory with 12 banks.

Intuitively, if we consider the maximum bank required by each task in any criticality mode, then we see that a memory with 32 banks can run all the tasks in parallel and hence will be able to schedule all the tasks within their deadlines. But our system may not always have huge amount of resources to execute all the tasks in parallel. So we need to schedule the tasks in such a way that we use our resources efficiently as well as we can meet the task deadlines.

Given the above set of tasks and a memory with say 12 banks, we need to answer whether

all these tasks can complete their execution within their deadlines. The above decision problem can be formulated using the following conventions-

Definition 3.1.1 *i^{th} task τ_i is defined as -*

$\tau_i = (L_i, B_i, E_i, D_i)$

where, $L_i \in \mathbb{Z}^+$, denoting j criticality levels which task τ_i can exibit,

$B_i : L_i \to \mathbb{N}$, B_i denotes the number of parallel bank access that τ_i exhibits at each criticality level in L_i,

$E_i : L_i \to \mathbb{R}$, E_i denotes the percentage of total task that can be completed by τ_i at each criticality level in L_i.

$D_i \in \mathbb{Z}^+$, denotes the deadline of i^{th} task τ_i

The problem is to synthesize a schedule for the tasks at each cycle, with each task assigned to one of its criticality levels, such that all tasks finish execution by their respective deadlines. A feasible schedule to the task set (a solution to the above problem) must therefore, satisfy the following conditions:

- Condition 1: In every cycle a task will execute in exactly one of its criticality levels.
- Condition 2: A task must complete 100% of its execution before its deadline.
- Condition 3: At any cycle the total number of banks accessed by all the executing tasks must be less than or equal to the number of banks available to the system.

We now attempt to characterize the hardness of the problem. As above, we are given a set T of tasks, each having deadline d(t) $\in \mathbb{Z}^+$, a number m $\in \mathbb{Z}^+$ of banks, number r $\in \mathbb{Z}^+$ of resource requirements (number of parallel bank access) of each task in each cycle depending on its criticality level, resource bounds m in each cycle, resource requirements $B_i(t)$ of i^{th} task, $0 \leq B_i(t) \leq$ m. The problem of finding a schedule for the above is NP-complete. The problem is definitely in NP since we can verify in polynomial time if a given schedule satisfies the requirement on bank access and task deadline. To show that our problem is NP-hard, we present below a polynomial time reduction from the partition problem [21] to our problem. Given an instance α of partition problem, we will generate an instance of our problem σ such that if α can be partitioned into p partitions, σ can also be scheduled over p cycles (where p denotes the maximum largest deadline value) with m banks.

Our problem is a variant of the resource constrained problem [6] with resource bounds equal to m and each task having individual task deadlines. A task can be identified as a set of subtasks executing with different amount of resources. In each cycle, we need to generate a partition consisting of subtasks such that the sum of resources at every cycle is atmost m and each task can complete 100% of its execution within their individual deadlines.

Our problem instance α consists of a set of elements, represented by E_i, i varies from 1 to $|\alpha|$. Each element of E_i is again a set of tuples - a number and a cost associated with that number. So, each E_i consists of tuples of the form (n_{ij},C_{ij}), where j varies from 1 to $|E_i|$. Our objective is to partition α into p partitions such that each partition p_k contains atmost one number n_{ij} from every E_i so that the sum of the numbers in each partition is atmost m and the sum of the costs corresponding to all the selected numbers of E_i over p partitions is atleast 100. Each element E_i again can be selected for some definite number of partitions. On selecting the j^{th} number from the i^{th} element in the set α, a subtask of i^{th} task is selected whose bank requirement is the j^{th} number that is selected and the cost associated with the number, C_{ij}, is the percentage of execution of i^{th} task at $j * th$ criticality level with n_{ij} number of parallel bank access. In every partition, p_k, the sum of all the numbers must be atleast m. When the total cost of i^{th} element is greater than or equal to 100 over all p partitions, then i^{th} task gets completed. On getting a partition at every iteration with the above constraints, we actually get a schedule for our problem. So if α can be partitioned into p partitions, σ is schedulable over p cycles with m banks. Again, the converse is also true. Suppose that our problem can complete execution of all the tasks over p cycles. We construct a schedule σ of our problem such that over p cycles, σ can schedule all the tasks within their deadlines with m number of resources (banks) in each cycle. For i^{th} task executing in j^{th} criticality level in the schedule σ, a number from the i^{th} element of the multiset α is selected. The total number of banks accessed in every cycle in the schedule σ is atmost m, ensuring the sum of the numbers in every partition cannot exceed m. Completion of i^{th} task in the schedule σ over all p cycles is equivalent to the sum of costs corresponding to every number selected from the i^{th} element in α over all p partitions to be atleast 100. A schedule σ exists, denoting that all the tasks can be completed in p cycles with m banks. This implies α can also be partitioned into p partitions. Thus, the problem of deciding if a schedule exists for all the tasks with individual task deadlines over m banks in every cycle is NP-complete.

3.1.2 Constraint Formulation

The decision problem introduced above can be stated with the following constraints. We are given a number of banks z. To formulate the above problem we define a decision variable say f_{ijk}.

$$f_{ijk} = \begin{cases} 1 & \text{if } i^{th} \text{ task executes in } j^{th} \text{ criticality level in } k^{th} \text{ cycle} \\ 0 & \text{otherwise} \end{cases}$$

The above decision problem can be expressed with a set of constraints which are as follows. From Condition 1, we get that for all cycles, a task can execute in exactly one of its criticality level. Therefore, for each task τ_i,

$$\sum_{j=1}^{|L_i|} f_{ijk} = 1, \forall k, 1 \leq k \leq D_i \tag{3.1}$$

From Condition 2, we get that all tasks should complete 100% execution before their deadlines. The percentage of task that gets completed at the last cycle may exceed 100. So we have used '$\geq$' instead of '='. Therefore,

$$\sum_{k=1}^{D_i} \sum_{j=1}^{|L_i|} E_i(L_{ij}) * f_{ijk} \geq 100, \forall i, 1 \leq i \leq n \tag{3.2}$$

Here, $E_{ij}(L_{ij})$ denotes the percentage of i^{th} task that can be executed at criticality level j, n denotes the total number of tasks. From Condition 3, we get that at any cycle the total number of banks required by all the tasks executing at that cycle must be atmost z.

$$\sum_{i=1}^{n} \sum_{j=1}^{|L_i|} B_i(L_{ij}) * f_{ijk} \leq z, \forall k, 1 \leq k \leq D_i \tag{3.3}$$

where n denotes the number of tasks and z denotes the number of banks available to the system. Here, $B_i(L_{ij})$ denotes the parallel bank access required by i^{th} task at criticality level j.
In the above decision problem the value of z is 12. For z = 12, we check whether all the tasks will meet their deadlines. We get a total of $(4 + 5 + 6) = 15$ constraints from Condition 1 for different tasks at each cycle k.

$$f_{111} + f_{121} + f_{131} = 1 \tag{3.4}$$

$$f_{211} + f_{221} + f_{231} = 1 \tag{3.5}$$

$$f_{311} + f_{321} + f_{331} + f_{341} = 1 \tag{3.6}$$

$$f_{112} + f_{122} + f_{132} = 1 \tag{3.7}$$

$$f_{212} + f_{222} + f_{232} = 1 \tag{3.8}$$

$$f_{312} + f_{322} + f_{332} + f_{342} = 1 \tag{3.9}$$

$$f_{113} + f_{123} + f_{133} = 1 \tag{3.10}$$

$$f_{213} + f_{223} + f_{233} = 1 \tag{3.11}$$

$$f_{313} + f_{323} + f_{333} + f_{343} = 1 \tag{3.12}$$

$$f_{114} + f_{124} + f_{134} = 1 \tag{3.13}$$

$$f_{214} + f_{224} + f_{234} = 1 \tag{3.14}$$

$$f_{314} + f_{324} + f_{334} + f_{344} = 1 \tag{3.15}$$

$$f_{215} + f_{225} + f_{235} = 1 \tag{3.16}$$

$$f_{315} + f_{325} + f_{335} = 1 \tag{3.17}$$

$$f_{316} + f_{626} + f_{336} = 1 \tag{3.18}$$

Again, from Condition 2, we get 3 constraints for different tasks.

$$25 * (f_{111} + f_{112} + f_{113} + f_{114}) + 35 * (f_{121} + f_{122} + f_{123} + f_{124}) \geq 100 \tag{3.19}$$

$$20 * (f_{211} + f_{212} + f_{213} + f_{214} + f_{215}) + 30 * (f_{221} + f_{222} + f_{223} + f_{224} + f_{225}) \geq 100 \tag{3.20}$$

$$15 * (f_{321} + f_{322} + f_{323} + f_{324} + f_{325} + f_{326}) + 25 * (f_{331} + f_{332} + f_{333} + f_{334} + f_{335} + f_{336}) + 40 * (f_{341} + f_{342} + f_{343} + f_{344} + f_{345} + f_{346}) \geq 100 \tag{3.21}$$

Again, from Condition 3, we get 6 constraints for different clock cycle k.

$$6 * f_{111} + 10 * f_{121} + 5 * f_{211} + 10 * f_{221} + 4 * f_{321} + 8 * f_{331} + 12 * f_{341} \leq 12 \tag{3.22}$$

$$6 * f_{112} + 10 * f_{122} + 5 * f_{212} + 10 * f_{222} + 4 * f_{322} + 8 * f_{332} + 12 * f_{342} \leq 12 \tag{3.23}$$

$$6 * f_{113} + 10 * f_{123} + 5 * f_{213} + 10 * f_{223} + 4 * f_{323} + 8 * f_{333} + 12 * f_{343} \leq 12 \tag{3.24}$$

$$6 * f_{114} + 10 * f_{124} + 5 * f_{214} + 10 * f_{224} + 4 * f_{324} + 8 * f_{334} + 12 * f_{344} \leq 12 \tag{3.25}$$

$$5 * f_{215} + 10 * f_{225} + 4 * f_{325} + 8 * f_{335} + 12 * f_{345} \leq 12 \tag{3.26}$$

$$4 * f_{326} + 8 * f_{336} + 12 * f_{346} \leq 12 \tag{3.27}$$

Solving these set of constraints, we get that the above tasks cannot meet their deadlines on a memory with bank size 12. So the above set of tasks is not schedulable for a memory with 12 banks. This leads us to two questions which are discussed in the following sections-

1: The maximum number of tasks that can be executed at different criticality levels within their deadlines on a memory with z banks.

2: The minimum number of banks required to design a system consisting of tasks with different criticality levels and different deadlines.

3.2 Task Maximization Problem

The above decision problem leads to the question that if a set of tasks is not schedulable with a bank of particular size, then what is the maximum number of tasks which can be schedulable for the set of tasks with the above parameters.

3.2.1 Problem Formulation

Problem 3.2.1 *Given a set of tasks with different sets of criticality levels along with number of parallel bank access required at each criticality level, and percentage of task completed if executed at that criticality level. Each task is also associated with a deadline. We are given a memory with z number of banks, we need to find out the maximum number of tasks that can complete their execution within their deadlines.*

We explain the task maximization problem with the set of tasks in Table 3.1. The problem is NP-hard. To show that our problem is NP-hard, we give a polynomial time reduction from partition problem [21] to our problem. Given a instance α of partition problem, we will generate an instance of our problem σ such that if the gain over α is atleast K over p partitions, σ can also ensure completion of K tasks over p cycles. Our problem is a variant of scheduling to minimize weighted completion time [6] with resource bounds equal to m in each cycle. A task instance in our problem can be identified as a set of subtasks executing with different amount of resources. In each cycle, we need to generate a partition consisting of subtasks such that the sum of resources at every cycle is atmost m and atleast K tasks can be completed within their individual deadlines. We associate weights w(t) to each task such that on completion of each task a weight w(t) is added to the total cost of the system. We need to guarantee that the total weight associated with the system must be atleast K * w(t).

Our problem instance α consists of a set of elements, represented by E_i, i varies from 1 to $|\alpha|$. Each element of E_i is again a set of tuples - a number and a cost associated with that number. So, each E_i consists of tuples of the form (n_{ij},C_{ij}), where j varies from 1 to $|E_i|$. We need to partition α into p partitions such that each partition p_k contains atmost one number n_{ij} from every E_i so that the sum of the numbers in each partition is atmost m and the sum of the costs corresponding to all the selected numbers of E_i over p partitions is atleast 100. Our objective is to guarantee that the number of elements E_i whose cost over p partitions has exceeded 100 is atleast K, ie, we have a gain of atleast K. Each element E_i again can be selected for some definite number of partitions.

On selecting j^{th} number from i^{th} element in the set α, a subtask of i^{th} task is selected whose bank requirement is the j^{th} number that is selected and the cost associated with the number, C_{ij}, is the percentage of execution of i^{th} task at j^{th} criticality level with n_{ij} number of parallel bank access. In every partition, p_k, the sum of all the numbers must

be atleast m. When the total cost of i^{th} element is greater than or equal to 100 over all p partitions, then i^{th} task gets completed and 1 is added to the gain function. If α results in a gain of atleast K over p partitions, then σ can also ensure completion of K tasks over p cycles with a total weight of atleast K * w(t) over p cycles.

Again, the converse is also true.

Suppose that our problem can complete execution of K tasks over p cycles. We construct a schedule σ of our problem such that over p cycles, σ can schedule K tasks within their deadlines with m number of resources (banks) in each cycle. For i^{th} task executing in j^{th} criticality level in the schedule σ, a number from the i^{th} element of the multiset α is selected. The total number of banks accessed in every cycle in the schedule σ is atmost m, ensuring the sum of the numbers in every partition cannot exceed m. Completion of i^{th} task in the schedule σ over all p cycles is equivalent to the sum of costs corresponding to every number selected from the i^{th} element in α over all p partitions to be atleast 100. Now, completion of K tasks over p cycles in σ means the total cost to the system is K * w(t). Therefore, number of elements in α whose sum is atleast 100 is K. So, α has a gain of K over p partitions.

Therefore, maximum task execution over m banks with individual task deadlines and a resource bound of m in every cycle is NP-hard.

3.2.2 Maximum Task Execution : Constraint Formulation

We introduce a new decision variable C_i to keep track of the task completion. C_i marks the completion of task T_i.

$$C_i = \begin{cases} 1 & i^{th} \text{ task } T_i \text{ has completed 100\% of the task} \\ 0 & \text{otherwise} \end{cases}$$

We define the task maximization problem using Integer Linear Programming (ILP) as follows -

$$\text{Maximize } \sum_{i=1}^{n} C_i, \text{ subject to}$$

For each task τ_i,

$$\sum_{j=1}^{|L_i|} f_{ijk} \leq 1, \forall k, 1 \leq k \leq D_i \tag{3.28}$$

$$\sum_{k=1}^{D_i} \sum_{j=1}^{|L_i|} E_i(L_{ij}) * f_{ijk} \geq 100 * C_i, \forall i, 1 \leq i \leq n \tag{3.29}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{|L_i|} B_i(L_{ij}) * f_{ijk} \leq z, \forall k, 1 \leq k \leq D_i \tag{3.30}$$

From Condition 1, we get that at any cycle k, a task can exist in any one of its criticality level. Therefore, Eqn. 3.28 holds from Condition 1. Similarly, Eqn. 3.29 is obtained from Condition 2. Hence, if any task say T1 has completed its 100% execution, then the flag C_1 is set and the Condition 2 holds. If C_1 flag is 0, meaning that the task T1 has not yet completed its 100% execution. So, Condition 2 holds in both the cases. From Condition 3, we get Eqn. 3.30 where the number of banks required must be less than or equal to z at every cycle. Formulating the above equations in the form of constraints on the set of tasks in Table 3.1 for the task maximization problem, we get a total of 18 constraints from Condition 1, 3 constraints from Condition 2 and 6 constraints from Condition 3. Solving the above set of 27 constraints, we get the answer for the maximum number of tasks that can be executed and also the schedule for the task maximization problem. Table 3.3 in the Results section of this chapter shows the maximum number of tasks that can be schedulable for a given set of tasks and a given set of system resources.

3.3 Bank Minimization Problem

The decision problem discussed in the first section of this chapter can only tell us if a set of tasks with a fixed number of banks and with certain task parameters is schedulable or not. But if no feasible schedule exists then we need to know the minimum number of banks required to schedule a given set of tasks.

3.3.1 Problem Formulation

Problem 3.3.1 *Given a set of tasks with different deadlines. Each task is associated with different criticality levels, number of parallel bank access at that criticality level and percentage of task executed on serving those memory requests at that criticality level. We need to answer the minimum number of banks required to execute all the tasks within their deadlines.*

This problem can be formulated in a similar way like the above decision problem discussed in the first section with minor changes in the constraints and the objective function.

3.3.2 Hardness Characterization of the problem

Problem 3.3.2 *Given a set T of tasks, each having length l(t) $\in \mathbb{Z}^+$, with deadline d(t) $\in \mathbb{Z}^+$, number r $\in \mathbb{Z}^+$ of resource requirements (number of parallel bank access), resource requirements $B_i(t)$ of i^{th} task, for each task t - Can we get the minimum number of banks K required to schedule all the tasks t $\in$ T with resource bound of K in each cycle over a period of p cycles.*

It can be shown that this problem is NP-hard using a similar reduction technique as in the case of the decision problem.

3.3.3 Constraint Formulation

Let z be the number of memory banks and n be the number of tasks to be executed at a particular instant. Our problem can be expressed using Integer Linear Programming as -

Minimize z subject to

For each task τ_i,

$$\sum_{j=1}^{|L_i|} f_{ijk} \leq 1, \forall k, 1 \leq k \leq D_i \tag{3.31}$$

$$\sum_{k=1}^{D_i} \sum_{j=1}^{|L_i|} E_i(L_{ij}) * f_{ijk} \geq 100, \forall i, 1 \leq i \leq n \tag{3.32}$$

$$\sum_{i=1}^{n} \sum_{j=1}^{|L_i|} B_i(L_{ij}) * f_{ijk} \leq z, \forall k, 1 \leq k \leq D_i \tag{3.33}$$

By applying the above constraints on the tasks in Table 3.1, we require a memory with atleast 15 banks to complete the tasks within their deadlines. A memory with bank size less than 15 will fail to meet all the task deadlines. Table 3.2 in the Results section of this chapter shows some sets of critical tasks and corresponding minimum number of banks required to execute all the tasks within their deadlines.

3.4 Bank Minimization using Binary Search

We know that for the bank minimisation problem, the solution for the minimum number of banks will lie in between 1 to M where M is the sum of maximum number of banks required

by any task at its any criticality level. For the tasks in Table 3.1, the optimal solution will lie somewhere in between 1 to (10 + 10 + 12) = 32, (the sum of the maximum number of banks of each task). Using Integer Linear Programming to minimise or maximize the objective function requires a lot of computation cost for a very large problem with many constraints. Also it increases the search space and time complexity of the problem. We can significantly decrease the computational cost and space complexity of the problem by applying a binary search on the search space. Since, we know that a solution will always exist in between 1 and M, therefore, instead of solving for the minimum number of banks we can apply a binary search on the value of z and check if the solution is optimal for that value of z. The solution is a straight adaptation of the one in [16].

3.4.1 Binary Search Algorithm on the bank minimization problem

Algorithm 1, 2 and 3 give a description of our binary search algorithm on bank minimisation problem.

Algorithm 1 Bank Minimization using binary search

```
1: function BINSRCH(n, TaskList)  ▷ n denotes the number of tasks, Tasklist is the list of n tasks along with all the task parameters
2:     min = 1  ▷ Minimum number of banks can be 1
3:     max = compute_max_banks(n, TaskList)  ▷ find upper bound for the number of banks required
4:     result = -1
5:     while (max > min) and (result ≠ 0) do
6:         mid = (min + max)/2
7:         result = check_feasibility(mid, n, TaskList)  ▷ a Function which returns positive value if mid is greater than optimal solution and negative value if mid is less than optimal
8:         if result < 0 then
9:             min = mid + 1
10:        else if result > 0 then
11:            max = mid -1
12:        end if
13:    end while
14:    return mid
15: end function
```

Algorithm 2 compute maximum number of banks required

```
1: function COMPUTE_MAX_BANKS(n, TaskList)   ▷ calculates the upper bound for the number of banks required
2:     max = 0
3:     for i = 1 to n do
4:         for j = 1 to length(TaskList[i].Criticality_Level) do
5:             if TaskList[i].Bank[j] > max then
6:                 max = TaskList[i].Bank[j]
7:             end if
8:         end for
9:     end for
10:    return max
11: end function
```

Algorithm 3 check feasibility of the solution

```
1: function CHECK_FEASIBILITY(val, n, TaskList)
2:     for i = 1 to n do
3:         for j = 1 to length(TaskList[i].Criticality) do
4:             for k = 1 to TaskList[i].Deadline do
5:                 Apply the constraints for a bank of size val   ▷ call LpSolver and apply the constraints for a bank of size val
6:             end for
7:         end for
8:     end for
9:     result = assert_optimal()   ▷ check if the status of the solution is optimal for a bank of size val
10:    return result   ▷ result < 0 for infeasible solution, result > 0 for feasible solution, result equal 0 for optimal solution
11: end function
```

3.4.2 Complexity Analysis

Unlike conventional procedures for getting the minimum number of banks required to schedule a set of tasks within their deadlines, binary search reduces the search space by half at each step. Thus, in the worst case, number of function calls to check the feasibility of the solution is equal to the number of times we calculate the mid in the binary search problem. The check_feasibility() function calls the decision problem to check for the optimality of the solution. It checks for which value of mid our solution is optimal (the point where the function returns 0). This is equivalent to finding 0 in an array of sorted numbers by binary search. For a problem of size n, our solution is logarithmic in n, as is the usual case with binary search. So the search space explored on which the decision problem is called is smaller, corresponding to the mid values, and hence space complexity of the problem reduces significantly. Moreover, since the decision problem is called log(n) times in the worst case, the space required by the decision problem can be evaluated as the peak requirement among all the space required by the log(n) function calls. As these log(n) function calls are called individually, so the space allocated at each function call is significantly much lower than that allocated for solving the bank minimization problem. On the other hand, we apply ILP to get an answer to the minimization problem on the entire search space of the problem. Though the solver uses some heuristics to minimize the search space, the space complexity is larger. A comparative study of the space complexities of binary search in contrast to our ILP based solution on different task sets is shown in the results section of this chapter.

3.4.3 Correctness of binary search based optimal solution generation

We wish to prove here that our approach always generates a valid optimal solution for the minimum number of banks. Let P(n) be the assertion that binsrch works correctly for inputs where right - left = n. If we can prove that P(n) is true $\forall$ n, then we know that binsrch will definitely give us the value of n for which the optimal solution exists.

Base Case. In the above example when n = 0, min = max = m (say). Since we have assumed that the iteration would continue till the optimal solution is found between min and max, it must be the case that x = 0, ie, the optimal solution lies at m, and the function will return m, a value in between min and max.

Inductive Step. We assume that binsrch works as long as left - right $\leq$ k. Our goal is to prove that it works on an input where left - right = k + 1. There are three cases, where x = 0, where x < 0 and where x> 0.

Case x = 0, the optimal solution lies at m. Clearly the function works correctly.

Case x > 0, the optimal solution lies at some value smaller than x. We know that natural

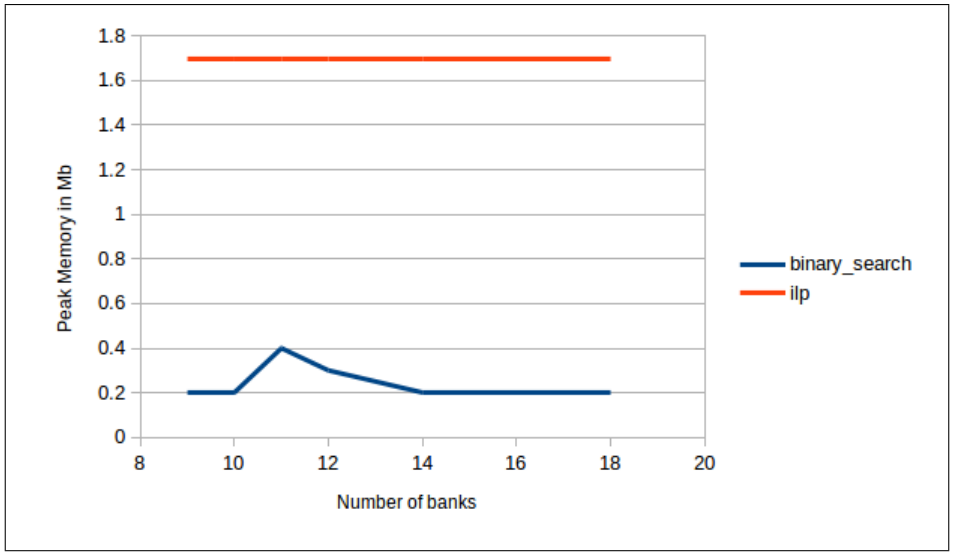

Figure 3.1: Peak memory plot on Taskset I by Binary Search vs ILP

numbers between min and max are in ascending order, hence they are sorted. Therefore, the optimal solution must be found between min and m - 1. The n for the next iteration is n = m - 1 - left = (left + right)/2 - 1 - left. (Note that x is the floor of x, which rounds it down toward negative infinity.) If left + right is odd, then n = (left + right - 1)/2 - 1 - left = (right - left)/2 - 1, which is definitely smaller than right - left. If left + right is even then n = (left + right)/2 - 1 - left = (right - left)/2, which is also smaller than k + 1 = right - left because right - left = k + 1 > 0. So the recursive call must be to a range of a that is between 0 and k cells, and must be correct by our induction hypothesis.

Case x < 0. The optimal solution lies at some value greater than x. This is more or less symmetrical to the previous case. We need to show that r - (m + 1) ≤ right - left. We have r - (m + 1) - l = right - (left + right)/2 - 1. If right + left is even, this is (right - left)/2 - 1, which is less than right - left. If right + left is odd, this is right - (left + right - 1)/2 - 1 = (right - left)/2 - 1/2, which is also less than right - left. Therefore, the iterative call is to a smaller range of the array and can be assumed to work correctly by the induction hypothesis. We can thus conclude that binsrch is correct.

3.5 Results

This section presents different results generated on different set of tasks.

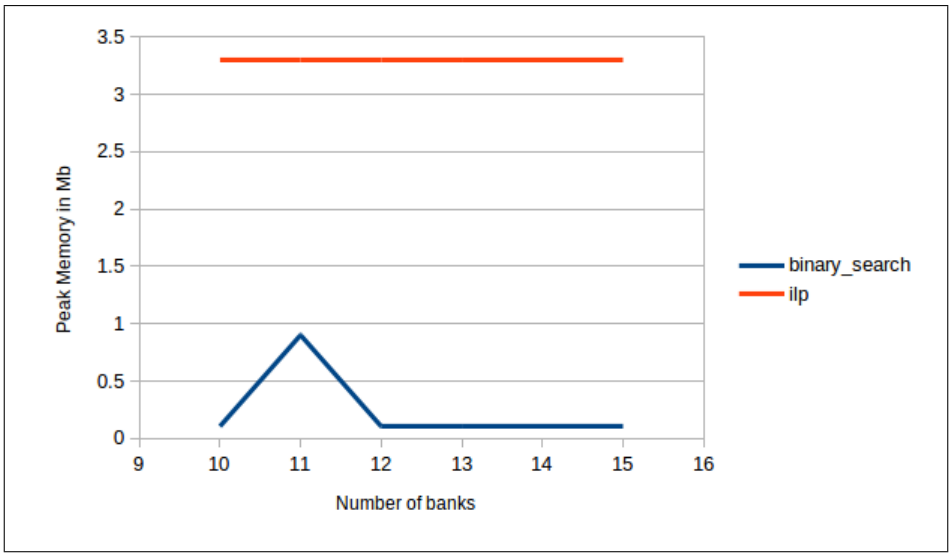

Figure 3.2: Peak memory plot on Taskset II by Binary Search vs ILP

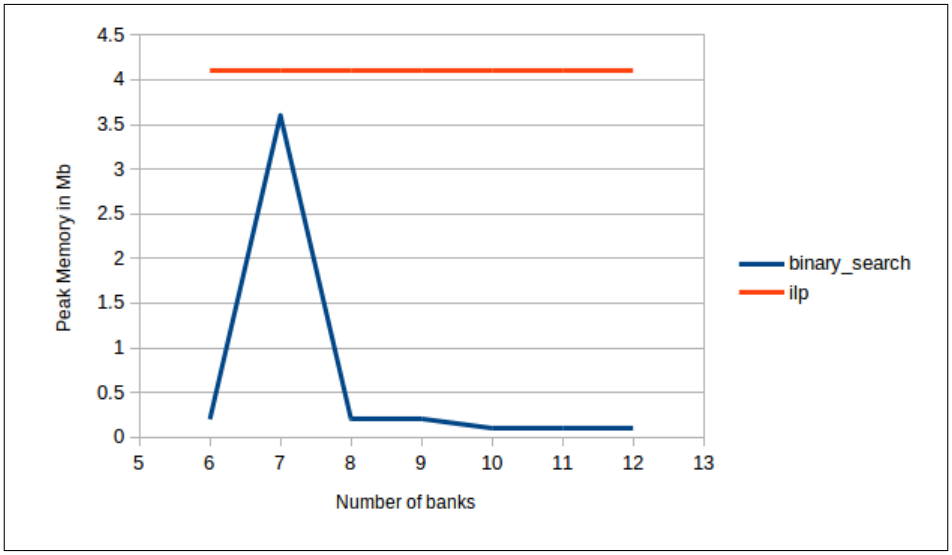

Figure 3.3: Peak memory plot on Taskset III by Binary Search vs ILP

Task ID	Criticality Level	Parallel Bank Access	Percentage of task executed	Deadline	Minimum Bank Requirement
1	L1	7	18%	5 cycles	18 banks
	L2	10	40%		
	L3	15	45%		
2	L1	5	15%	7 cycles	
	L2	8	25%		
	L3	12	35%		
3	L1	10	30%	8 cycles	
	L2	13	45%		
	L3	0	0%		
4	L1	6	20%	5 cycles	
	L2	9	30%		
1	L1	8	25%	6 cycles	12 banks
	L2	12	50%		
2	L1	6	30%	4 cycles	
	L2	8	40%		
	L3	10	55%		
3	L1	5	35%	8 cycles	
	L2	10	45%		
1	L1	5	10%	6 cycles	15 banks
	L2	10	30%		
2	L1	0	0%	4 cycles	
	L2	8	40%		
	L3	10	50%		
3	L1	4	20%	8 cycles	
	L2	6	30%		
	L3	7	40%		
4	L1	0	0%	10 cycles	
	L2	3	15%		
	L3	6	25%		
5	L1	8	20%	9 cycles	
	L2	12	35%		
1	L1	8	30%	6 cycles	14 banks
	L2	10	40%		
2	L1	5	25%	8 cycles	
	L2	8	35%		
3	L1	4	15%	7 cycles	
	L2	10	30%		
4	L1	7	20%	9 cycles	
	L2	9	35%		

Table 3.2: Minimum number of banks required to execute different sets of critical tasks

Task ID	Criticality Level	Parallel Bank Access	Percentage of task executed	Deadline	Number of banks, Task ID of the completed tasks
1	L1	4	16%	8 cycles	7 banks - Task ID (1,2,4)
	L2	5	20%		
	L3	7	25%		
2	L1	3	25%	6 cycles	5 banks - Task ID (2,4)
	L2	4	30%		
	L3	6	35%		
3	L1	1	5%	5 cycles	6 banks - Task ID (1,4)
	L2	2	10%		
	L3	4	15%		
	L4	6	28%		
4	L1	2	15%	9 cycles	4 banks - Task ID (2)
	L2	3	18%		
	L3	5	22%		
1	L1	4	5%	14 cycles	3 banks - Task ID (3)
	L2	6	8%		
2	L1	5	10%	11 cycles	8 banks - Task ID (2,3)
	L2	7	12%		
3	L1	3	15%	7 cycles	14 banks - Task ID (1,2,3)
	L2	5	18%		
1	L1	2	3%	11 cycles	7 banks - Task ID (1)
	L2	6	10%		
2	L1	3	5%	9 cycles	13 banks - Task ID (1,2)
	L2	5	8%		
	L3	7	12%		
1	L1	6	10%	11 cycles	6 banks - Task ID (2)
	L2	7	12%		
2	L1	2	8%	12 cycles	8 banks - Task ID (2,3)
	L2	3	10%		
3	L1	3	10%	8 cycles	12 banks - Task ID (1,2,3)
	L2	5	15%		

Table 3.3: Maximum number of tasks that can complete execution on a given number of banks

Task ID	Criticality Level	Parallel Bank access	Percentage of task executed	Deadline
1	L1	2	15%	10 cycles
	L2	4	24%	
2	L1	1	10%	7 cycles
	L2	3	15%	
	L3	5	20%	
3	L1	2	15%	8 cycles
	L2	3	20%	
	L3	7	25%	
	L4	8	30%	
4	L1	4	12%	7 cycles
	L2	5	25%	
5	L1	2	10%	9 cycles
	L2	4	12%	
	L3	7	18%	
6	L1	3	15%	12 cycles
	L2	5	20%	
	L3	7	30%	

Table 3.4: Taskset I

Task ID	Criticality Level	Parallel Bank access	Percentage of task executed	Deadline	Minimum number of banks
1	L1	3	18%	8 cycles	
	L2	5	25%		
2	L1	4	21%	6 cycles	
	L2	5	26%		12 banks
	L3	6	32%		
3	L1	3	6%	7 cycles	
	L2	6	13%		
	L3	8	17%		
	L4	9	25%		

Table 3.5: Taskset II

Task ID	Criticality Level	Parallel Bank access	Percentage of task executed	Deadline	Minimum number of banks
1	L1	4	15%	10 cycles	
	L2	6	30%		
2	L1	2	22%	10 cycles	
	L2	5	30%		
3	L1	3	18%	10 cycles	8 banks
	L2	7	23%		
4	L1	3	15%	10 cycles	
	L2	5	21%		

Table 3.6: Taskset III

Chapter 4

Memory Bank Prioritization

In the previous chapter, we have discussed about the methods to evaluate the optimal number of memory banks required to design a mixed-criticality system. Also, given a mixed-criticality system with t number of tasks, whether a memory with n number of banks can support such a system can also be evaluated by solving an ILP discussed in the previous chapter. With the help of the methods explained in the previous chapter, for a particular mixed criticality-system, we can get a schedule which determines which task should be executed at which criticality level/mode in every cycle so that the mixed-criticality system can be executed with the given memory system.

Each critical task in the mixed-criticality system consists of multiple memory requests and number of parallel memory bank access at each criticality level. The computation to determine the optimal number of banks for the design of such a mixed-criticality system is evaluated offline. The main disadvantage of this method is that the memory requests may not always allow parallel bank access when requests are served in real systems. After address translation, two or more memory requests may get mapped to same bank. Again, due to the open row policy, discussed in the introductory chapters, some memory requests get prioritized over the other. This problem have been discussed in the next section with the help of an example.

4.1 Motivating Example

Consider a mixed-criticality system consisting of two tasks with 5 and 8 memory requests respectively. The tasks specifications are shown in Table 4.1. It has been found that to execute the tasks within their deadlines we need a memory with atleast 3 banks. Each bank is assumed to be of size 128B with row size of 16B. So each bank has 8 rows. Each row again consists of 4 columns of 4B each. The memory is word addressable, so each address

Task ID	Criticality Level	Parallel Bank Access	Percentage of task executed	Deadline
1	L1	1	20%	4
	L2	2	40%	
	L3	3	60%	
2	L1	1	12.5%	6
	L2	2	25%	
	L3	4	50%	

Table 4.1: Task Specifications

is of 4B. The address mapping of each bank is given below -

Bank 0: Address 0-31 is mapped on this bank. Each row consists of 4 addresses, address 0-3 maps to row 0 of Bank 0, similarly address 4-7 maps to row 1 of Bank 0.

Bank 1: Address 32-63 is mapped on this bank. Address 32-35 is mapped to row 0 of Bank 1. Similarly, address 60-63 is mapped to row 7 of Bank 1.

Bank 2: Address 64-95 is mapped on this bank. Address 64-67 is mapped to row 0 of Bank 2. Similarly, address 91-95 is mapped to row 7 of Bank 2.

Now when these tasks are allowed to execute, the memory addresses generated by the above tasks are listed in Table 4.2. If we schedule the above tasks according to the optimal schedule generated by ILP, we need to execute the tasks in the following sequence in every cycle -

Cycle 1: T1 executes in Criticality level 1 with one bank access. T2 executes in Criticality level 2 with 2 parallel bank access. So in Cycle 1 we require 3 parallel bank accesses.

Cycle 2: T1 executes in Criticality level 2 with two parallel bank access. T2 executes in Criticality level 1 with one bank access. So, in Cycle 2, we have 3 parallel bank accesses.

Cycle 3: T1 executes in Criticality level 2 with 2 parallel bank access. T2 executes in Criticality level 1 with one bank access. So, we have 3 parallel bank accesses in Cycle 3.

Cycle 4: T2 executes in Criticality level 2 with 2 parallel bank accesses.

Cycle 5: T2 executes in Criticality level 2 with 2 parallel bank accesses.

After address translation, the mapping of different requests on different memory banks is explained below.

Task ID	Number of Memory Requests	Memory Addresses	Bank Number	Row Number
1	5	12	0	3
		36	1	1
		15	0	3
		39	1	1
		42	1	2
2	8	10	0	2
		22	0	5
		66	2	0
		37	1	1
		72	2	2
		75	2	2
		79	2	3
		83	2	4

Table 4.2: Memory Address Mapping

Explanation: In cycle 1, Row 3 of Bank 0 (address 12) is brought in the rowbuffer of Bank 0. Row 2 of Bank 0 (address 10) and Row 5 of Bank 0 (address 22) though scheduled to be served in Cycle 1, but cannot be served as some other request from T1 has already been scheduled to be served in the rowbuffer of Bank 0. So these two requests are kept in the waiting queue to be served later.

In cycle 2, Row 1 of Bank 1 (address 36) is brought to the rowbuffer of Bank 1. Row 3 of Bank 0 (address 15) results in row hit in Bank 0. So memory request to access address 15 is prioritized over pending requests to access address 10 and address 22. Thus, address 15 is served first. Similarly, in Cycle 3, request corresponding to address 39 is prioritized over other pending requests as address 39 is mapped to Row 1 of Bank 1 which is already in rowbuffer of Bank 1.

Table 4.3 shows in detail the overview of memory in every cycle. After address mapping of several memory requests to different memory banks, it has been found that both T1 and T2 fails to meet their deadline when executed on a memory with 2 banks.

4.2 Disadvantage of the existing method

The main disadvantage of the existing method is due to the address mapping policy, many requests get mapped to same bank and hence get buffered in the waiting queue. They have to wait for the bank availability in the subsequent cycles. When a memory address is

Cycle	Incoming Requests	Previous Pending Requests	Requests Served in the current cycle	Deadline Miss
1	12 (Bank 0, Row 3) 10 (Bank 0, Row 4) 22 (Bank 0, Row 5)		12 (Bank 0, Row 3)	
2	36 (Bank 1, Row 1) 15 (Bank 0, Row 3) 66 (Bank 2, Row 0)	10 (Bank 0, Row 4) 22 (Bank 0, Row 5)	15 (Bank 0, Row 3) 36 (Bank 1, Row 1) 66 (Bank 2, Row 0)	
3	39 (Bank 1, Row 1) 42 (Bank 1, Row 2) 37 (Bank 1, Row 1)	10 (Bank 0, Row 4) 22 (Bank 0, Row 5)	10 (Bank 0, Row 4) 39 (Bank 1, Row 1)	
4	72 (Bank 2, Row 2) 75 (Bank 2, Row 2)	22 (Bank 0, Row 5) 37 (Bank 1, Row 1)	22 (Bank 0, Row 5) 37 (Bank 1, Row 1) 72 (Bank 2, Row 2)	
5	79 (Bank 2, Row 3) 83 (Bank 2, Row 4)	75 (Bank 2, Row 2)	75 (Bank 2, Row 2)	T1
6		79 (Bank 2, Row 3) 83 (Bank 2, Row 4)	79 (Bank 2, Row 3)	
7		83 (Bank 2, Row 4)		T2

Table 4.3: Overview of Memory from cycle to cycle

served, the row to which this address belongs is brought to the rowbuffer in the memory. The memory request is then served by reading/writing at the appropriate column in that row in the rowbuffer. On the next cycle, if some memory request arrives which maps to the same bank and same row in the rowbuffer, the address is served first though some other addresses are waiting for memory access in the waiting queue. Due to this phenomenon, some tasks miss their deadlines in spite of waiting for cycle in the waiting queue.

4.3 Hardness Characterisation of the problem

Problem 4.3.1 *Given a set of n memory requests (M1, M2, M3 ... Mn) coming from a set of k tasks having different deadlines (D1, D2, ... Dk), executing on a memory with m banks. Can we schedule n memory requests on m banks over T cycles based on the address to which they are mapped so that all t tasks can meet their deadlines.*

The problem is NP-complete.

The problem is in NP.

Given a schedule to serve n memory requests, whether all n memory requests corresponding

to t tasks can be executed over T cycles can be verified in polynomial time.

The problem is NP-hard.

To show that our problem is NP-hard, we give a polynomial time reduction from CNF-SAT problem [23] to our problem. Given an instance α of CNF-SAT problem, we will generate an instance of our problem σ such that if α is satisfiable, σ can serve n memory requests over T cycles within the individual task deadlines.

Our problem can be considered as a variant of the CNF-SAT problem [23]. In Boolean logic, a formula is in conjunctive normal form (CNF) or clausal normal form if it is a conjunction of clauses, where a clause is a disjunction of literals, otherwise put, it is an AND of ORs. n memory requests get mapped to m different banks based on the address of each request. At a particular instant, a bank may remain free, whereas, many requests may get buffered at some other bank. We can think of n memory requests as a set of n clauses. We can think of our problem C as a set of n clauses $C_1, C_2, C_3, \ldots C_n$. i^{th} clause, C_i is True, if i^{th} memory request gets served within the task deadline, otherwise, C_i is False. Our goal is to give an assignment to the variables σ : V $\rightarrow$ True, False so that C evaluates to True. So, if α is satisfiable, σ can serve n memory requests over T cycles within the task deadlines.

The converse is also true.

Given σ can execute n memory requests within their deadlines. This indicates that all the memory requests can be served within the task deadlines over T cycles. Now, when i^{th} task gets executed within the task deadline, C_i is assigned True. So, if all n memory requests get served within the task deadlines, then C_i is assigned True, where i varies from 1 to n. Thus, C becomes satisfiable. So, if σ can complete execution of n memory requests within the task deadlines, α becomes satisfiable.

Thus, whether a set of n memory requests from t tasks can be served within individual task deadlines is NP-complete.

4.4 Solution to the problem

To solve the above problem, we propose a novel bank aware address mapping heuristic which partitions the memory requests based on a gain function calculated locally on arrival of a memory request. We use a partitioning heuristic to partition the memory requests across banks.

4.4.1 Bank scheduling using Task partitioning

We use an iterative heuristic for partitioning the network whose worst case computation time per pass grows linearly with size of the network [4] and which in general converges after several passes. Generally, very small number of passes are needed leading to a fast approximation algorithm for mincut partitioning, which is the basis of our work.

Given a network consisting of a set of cells(modules) connected by a set of nets(signals), the mincut partitioning problem consists of finding a partition of the set of cells into two blocks A and B such that the number of nets in the block is minimum. This is the main objective of our algorithm.

4.4.2 Our proposed methodology

Consider a mixed-criticality system where n is the optimal number of banks required to design such a system. We suggest to keep two extra banks which will not participate in address mapping. These two extra banks will be used to serve the pending requests in the waiting queue so that we can minimize the deadline misses of tasks. We consider the memory requests as the nodes in our problem. Requests which are mapped to same bank form a partition. We start with an initial partition of requests, which makes up our initial partition. We calculate a gain function over the task as the difference in the gains achieved by serving a memory request in the same bank or in some reserved bank. This difference in the two gains helps to decide whether a memory request will be served inplace or in the reserved bank. Gain function can be explained formally as -

Gain(Req_i) = Gain(Serving Req_i in the same bank) - Gain(Serving Req_i in some reserved bank)

= [T.deadline - (total memory access time + total waiting time in existing method)] - [T.deadline - (total memory access time + total waiting time in our method)]

Here, T is the task whose memory request is being served and T.deadline is the deadline of the task T. Let us consider that the i^{th} memory request Req_i has been mapped to j^{th} row (Row_j) of k^{th} bank ($Bank_k$). There are few cases with the help of which we can highlight the above problem.

Case 1: If the rowbuffer of $Bank_k$ currently holds the Row_j and if no other request is found to be mapped to Row_j, then the Req_i will be served immediately. So we do not need to replace Req_i. The cost to serve Req_i is equal to 1 row access time.

Case 2: If p number of requests all mapping to different rows of same bank, are ahead of Req_i in the waiting queue, then according to the FR-FCFS policy, Req_i will be served

after p requests. In this case, if the cost to serve Req_i in any one of the reserved banks is greater than that to solve after p memory requests of same bank, we allow Req_i to be served in $Bank_k$. So the cost to serve Req_i is the total time to serve p row misses followed by 1 row miss time. On the other hand, if p' out of p requests map to the same bank as the rowbuffer, then Req_i will be served after p' row hits followed by $p - p'$ row misses.

Case 3: If the rowbuffer of $Bank_k$ currently holds some row other than Row_j, say $Row_{j'}$, and if say k' number of requests are infront of Req_i in the waiting queue, then we need to calculate the cost to serve Req_i in $Bank_k$ after serving k' requests. Considering all k' requests result in row miss, the time to serve Req_i is the total time to serve k' row misses followed by 1 row miss corresponding to Req_i. We need to calculate the cost to serve Req_i in any reserved bank, say Res_l. To serve Req_i in any reserved bank, Res_l, we need to copy Row_j of $Bank_k$ to any row, say l' of Res_l and bring that row in the rowbuffer. So the total time to serve Req_i is the time to copy Row_j to Res_l followed by the time to bring the Row_j to the rowbuffer.

If the difference of the two costs in the Gain function is greater than 0, we serve Req_i in $Bank_k$, otherwise, we serve Req_i in one of the reserved banks.

4.4.3 Algorithm for Bank Aware partitioning

Algorithm 4 explains Bank Aware partitioning. We give a detailed explanation of our algorithm. Our algorithm takes as input a memory request of a task T in the form of $< address, requesttype >$, number of parallel memory accesses of the task T that are left to be scheduled and deadline of the task as input to our algorithm. Our output is the bank number to which each address will get mapped.

Step 1: The bank to which the Req is mapped is evaluated. The variable Bank is assigned that value. n is assigned the number of requests ahead of Req in the waiting queue.
Step 2: The number of requests in the waiting queue which are mapped to the same bank and same row as the Req is calculated. n2 is assigned that value. The number of requests which are mapped to some other bank other than Bank is calculated and n3 is assigned that value.
Step 3: If the row of the Req is same as that in the rowbuffer of the Bank, then we calculate the waiting time of Req. Three cases can occur-
Case 1: n2 = 0, then the waiting time of Req is $wait\_time_1 = 0$.
Case 2: n2 = n, then the waiting time of Req is $wait\_time_1$ = n * t1, where t1 is the memory access time when there is a row hit.
Case 3: n2 > 0 and n2 < n, then the waiting time of Req is the time to serve n2 row hits followed, ie,$wait\_time_1$ = n2*t1.
Step 4: If the row of the Req is not the same as that in the rowbuffer of the Bank, then

Cycle	Incoming Requests	Previous Pending Requests	Requests Served in the current cycle	Deadline Miss
1	12 (Bank 0, Row 3) 10 (Bank 0, Row 4) 22 (Bank 0, Row 5)		12 (Bank 0, Row 3)	
2	36 (Bank 1, Row 1) 15 (Bank 0, Row 3) 66 (Bank 2, Row 0)	10 (Bank 0, Row 4) 22 (Bank 0, Row 5)	15 (Bank 0, Row 3) 36 (Bank 1, Row 1) 66 (Bank 2, Row 0)	
3	39 (Bank 1, Row 1) 42 (Bank 1, Row 2) 37 (Bank 1, Row 1)	10 (Bank 0, Row 4) 22 (Bank 0, Row 5)	10 (Bank 0, Row 4) 39 (Bank 1, Row 1)	
4	72 (Bank 2, Row 2) 75 (Bank 2, Row 2)	22 (Bank 0, Row 5) 37 (Bank 1, Row 1)	22 (Bank 0, Row 5) 37 (Bank 1, Row 1) 72 (Bank 2, Row 2) 42 (Res 0, Row 2)	
5	79 (Bank 2, Row 3) 83 (Bank 2, Row 4)	75 (Bank 2, Row 2)	75 (Bank 2, Row 2)	
6		79 (Bank 2, Row 3) 83 (Bank 2, Row 4)	79 (Bank 2, Row 3) 83 (Res 0, Row 4)	
7				

Table 4.4: Overview of Memory from cycle to cycle on using Bank Aware Partitioning

the waiting time of Req, $wait\_time_1$ = n2*t1 + (n - n2 - n3)*t2, where t2 is the memory access time when there is a row miss.
Step 5: The waiting time of Req if served in any one of the reserved memory bank is $wait\_time_2$ = t3, where t3 includes the time to copy a row from one bank to another and the time to bring the copied row to the rowbuffer of the reserved memory bank.
Step 6: We calculate gain in the above two methods. Gain by method 1 is given as $Gain_1$ = deadline - ($wait\_time_1$ + n1*M), where M is the worst case memory access time. Gain by method 2 is given as $Gain_2$ = deadline - ($wait\_time_2$ + n1*M).
Step 7: If $Gain_1$ is greater than $Gain_2$, then we serve the request in Bank, otherwise we serve the memory request in any reserved memory bank.

4.4.4 Solution with an Example

Consider the same set of tasks as in Table 4.1 and Table 4.2 respectively. The above discussed method when applied on the same set of tasks with same memory requests, generates the following result as in Table 4.4. We have assumed that the time to serve each memory request is 1 cycle. We have assumed that our system in this example has one extra reserved memory bank denoted by Res. It has been found that on adopting our method, both T1 and T2 can be served within their deadlines.

Algorithm 4 Bank Aware Partitioning(Req, n1, deadline)

1: **Input** Req: Memory request to be served, n1: number of pending parallel accesses, deadline: deadline of the task
2: **Output** Bn : Bank number on which Req is finally scheduled
3: t1 = memory access time when row hit
4: t2 = memory access time when roow miss
5: t3 = time to copy a row to the Reserved Bank
6: M = Worst Case memory access time
7: n = number of requests in the waiting queue before task T
8: Bank = Req.bank ▷ Bank to which Req is initially mapped
9: n2 = 0
10: n3 = 0
11: **for** i = 1 to n **do**
12: **if** $Req_i.bank == Bank$ **then** ▷ checks if some other request is mapped to the same bank as Req
13: **if** Req_i.row == Bank.rowbuffer **then** ▷ checks if some other request is mapped to the same row as Req
14: n2 = n2 + 1
15: **end if**
16: **else**
17: n3 = n3 + 1
18: **end if**
19: **end for**
20: **if** Req.row == Bank.rowbuffer **then**
21: **if** n2 == 0 **then**
22: $wait\_time1 = 0$
23: **else if** n2 > 0 and n2 < n1 **then**
24: $wait\_time1$ = n2 * t1
25: **else if** n2 == n **then**
26: $wait\_time1$ = n * t1
27: **end if**
28: **else if** Req.row ≠ Bank.rowbuffer **then**
29: $wait\_time1$ = n2 * t1 + (n - n2 - n3) * t2
30: **end if**
31: $wait\_time2$ = t3
32: $Gain_1$ = Task.deadline - $(wait\_time_1 + n1 * M)$
33: $Gain_2$ = Task.deadline - $(wait\_time_2 + n1 * M)$
34: **if** $Gain_1 > Gain_2$ **then**
35: Bn = Bank.row
36: **else**
37: copy Bank.row to any row in the Reserved bank
38: Bn = Reserved.row
39: **end if**
40: return Bn

4.4.5 Performance Analysis

Consider two memory requests, M_1 arriving at instant t_1 and M_2 arriving at instant t_2, where $t_1 < t_2$. Let M_1 and M_2 are mapped to the same Bank B at rows R_1 and R_2 respectively. Let R be the active row in the rowbuffer of bank B. If $R_1 = R$, then M_1 will be served first followed by M_2. But if $R_2 = R$, then M_2 will be served first due to open row policy, followed by M_1.

Now, consider there be n requests arriving after M_2, all mapping to the same row R. According to the open row policy, all the n requests will be served first, then M_2 will be served. Let τ_1 units be the time to execute a memory request if there is a row hit. Let τ_2 units be the time to execute a memory request if there is a row miss. Clearly, $\tau_2 > \tau_1$. τ_1 includes only row access time, whereas, τ_2 includes loading of a row to the rowbuffer of a bank followed by row access time. So, M_2 will get a chance to execute after (n * τ_1 + τ_2) units. Let M_2 has arrived from Task T whose deadline is given by $T.deadline$ units. Let X be the number of parallel accesses remaining to be served in the memory after M_2 gets served. Now, the Gain of M_2 by following the conventional method for scheduling in DRAM, denoted by $Gain_1(M_2)$ is given by -

$Gain_1(M_2) = T.deadline$ - (n * τ_1 + τ_2 + X * α)

where α is the maximum memory access time to execute an instruction,

Let τ_3 be the time to copy a row from Bank B to one of the reserved bank R_b. Following our method, the Gain of M_2, denoted by $Gain_2(M_2)$ is given by -

$Gain_2(M_2) = T.deadline$ - (τ_3 + τ_1 + X * α)

If $Gain_2(M_2) > Gain_1(M_2)$, we follow our method. Otherwise, we follow the existing method.

$Gain_2(M_2)$ - $Gain_1(M_2) > 0$

$\implies$ [$T.deadline$ - (τ_3 + τ_1 + X * α)] - [$T.deadline$ - (n * τ_1 + τ_2 + X * α)] > 0

$\implies$ n * τ_1 + (τ_2 - τ_1) - $\tau_3 > 0$

$\implies$ n * τ_1 - $\tau_3 > 0$ [Since, τ_2 - $\tau_1 > 0$]

when n is very large, n * $\tau_1 >> \tau_2$

So, $Gain_2(M_2)$ - $Gain_1(M_2) > 0$.
Hence, our method is expected to yield better results than the conventional open row strat-

egy. Further, we discuss the advantages of our method over existing method.

1. Requests mapping to two different rows in the same bank in an interleaved manner -

Let $\alpha_1, \alpha_2, \ldots \alpha_n$ be n memory requests mapping to bank B in such a manner that all α_{2i} are mapped to row R1 and all α_{2i+1} are mapped to row R2 of B. If the rowbuffer of bank B holds row R1, then all odd requests will be pending in the waiting queue, whereas all even requests will get served. So, by open row policy, there will be n/2 row misses. Now, if we apply our method, row R2 will be mapped to some reserved bank in the system, say B' and all the odd requests will be directed to bank B' and served in that bank. So, we can increase the probability to meet the deadlines of the tasks.

2. Two critical tasks from two critical applications accessing two different chunks of memory mapped in consecutive locations -

Let us consider two critical tasks from two different applications are accessing memory locations in such a way that task from the first application are all mapped to row R1 of bank B, whereas, task from the second application are mapped to row R2 of bank B. Following the existing policy, either one of the tasks can meet their deadlines. The other task from the second application will not be able to meet their deadlines. Following our proposed method, the row from the second task will be copied to the reserved bank and all the remaining requests corresponding to that row get diverted to the reserved bank and meet deadline of both the tasks from different applications.

4.5 Results

This section deals with our discussions on implementation, benchmark programs used by us and results obtained from normal DRAMs vs our modified DRAM controller. We have implemented our own simulator. We have generated our results on Malardalen WCET benchmarks [9]. Table 4.5 gives a a description of the benchmark programs. We have generated our memory trace on these benchmark programs. Table 4.6 gives a description of some of the tasksets which are generated based on the number of instructions in the memory traces. We have compared the performances of our modified DRAM controller with existing DRAM controllers on a memory with bank of size 32B and 64B respectively. Some of the results have been shown in Table 4.7.

Benchmark Program	Number of Instructions	Memory Instructions
bs.c	179	64
cnt.c	295	95
duff.c	257	111
fac.c	164	48
fibcall.c	163	55
insertsort.c	189	66
lcdnum.c	198	49
ns.c	245	71
prime.c	227	69

Table 4.5: Details of Malardalen WCET benchmark programs

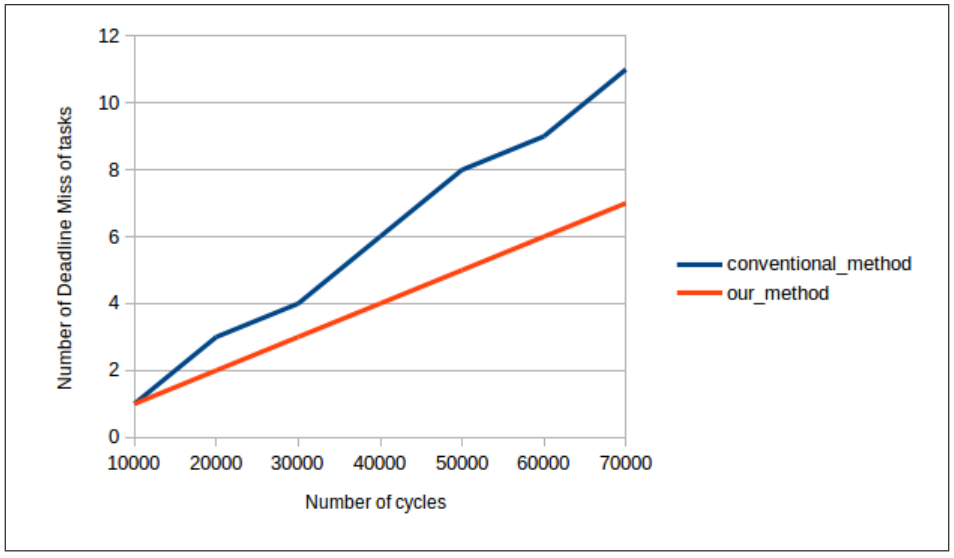

Figure 4.1: Number of Task Misses on Dataset I for a memory with 10 banks and row size 64B over 70,000 cycles

Dataset Number	Benchmark program	criticality level	number of parallel accesses	percentage of task completed
Dataset I	fibcall.c	L1	3	6%
		L2	4	8%
		L3	6	12%
	lcdnum.c	L1	2	3%
		L2	3	5%
		L3	4	7%
	fac.c	L1	1	2%
		L2	2	4%
		L3	4	8%
Dataset II	bs.c	L1	5	7%
		L2	8	12.5%
		L3	10	15%
	cnt.c	L1	5	5%
		L2	7	7%
		L3	10	10%
	duff.c	L1	7	6%
		L2	9	8%
		L3	11	9%
Dataset III	prime.c	L1	3	4%
		L2	6	8%
		L3	9	13%
	ns.c	L1	4	5%
		L2	6	8%
		L3	10	14%
	insertsort.c	L1	3	4%
		L2	6	9%
		L3	8	12%
Dataset IV	lcdnum.c	L1	5	10%
		L2	7	14%
		L3	9	18%
	insertsort.c	L1	6	9%
		L2	8	12%
		L3	10	15%
	fibcall.c	L1	5	9%
		L2	8	14%
		L3	11	20%

Table 4.6: Some Dataset generated on different benchmark programs

Dataset Number	Number of cycles	Number of Banks	Rowsize	Number of Task Miss (conventional method)	Number of Task Miss(our method)
Dataset I	10,000	10 banks	64B	1	1
	20,000			3	2
	30,000			4	3
	40,000			6	4
	50,000			8	5
	60,000			9	6
	70,000			11	7
	10,000	10 banks	32B	2	1
	20,000			4	2
	30,000			5	3
	40,000			8	5
	50,000			11	6
	60,000			12	7
	70,000			14	9
Dataset II	10,000	19 banks	64B	3	0
	20,000			6	1
	30,000			9	1
	40,000			14	4
	10,000	19 banks	32B	3	2
	20,000			5	3
	30,000			8	6
	40,000			13	11
Dataset III	10,000	15 banks	64B	2	1
	20,000			4	1
	30,000			5	2
	40,000			8	3
	10,000	15 banks	32B	3	2
	20,000			5	3
	30,000			6	4
	40,000			9	6
Dataset IV	10,000	12 banks	64B	2	1
	20,000			5	2
	30,000			7	4
	40,000			12	4
	10,000	12 banks	32B	3	2
	20,000			6	4
	30,000			9	6
	40,000			14	11

Table 4.7: Results on some of the Datasets

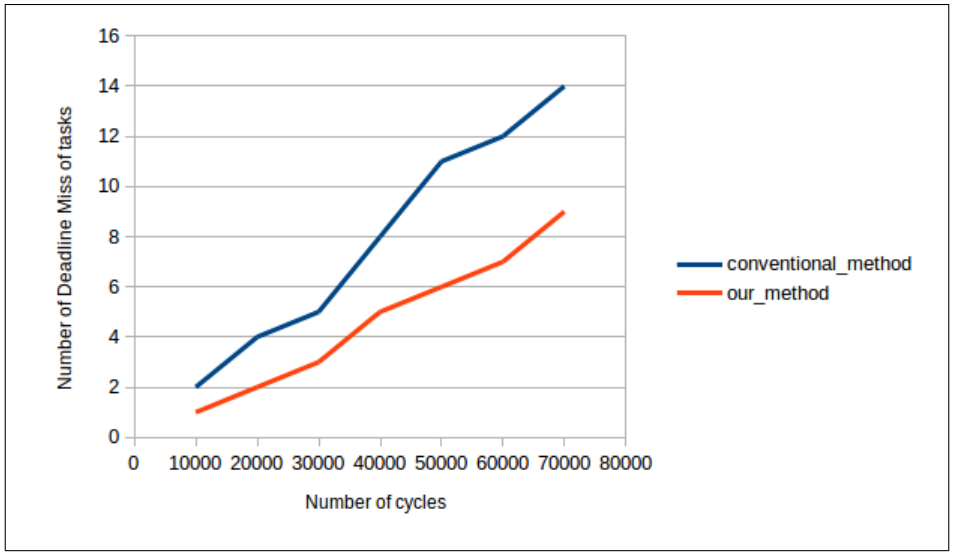

Figure 4.2: Number of Task Misses on Dataset I for a memory with 10 banks and row size 32B over 70,000 cycles

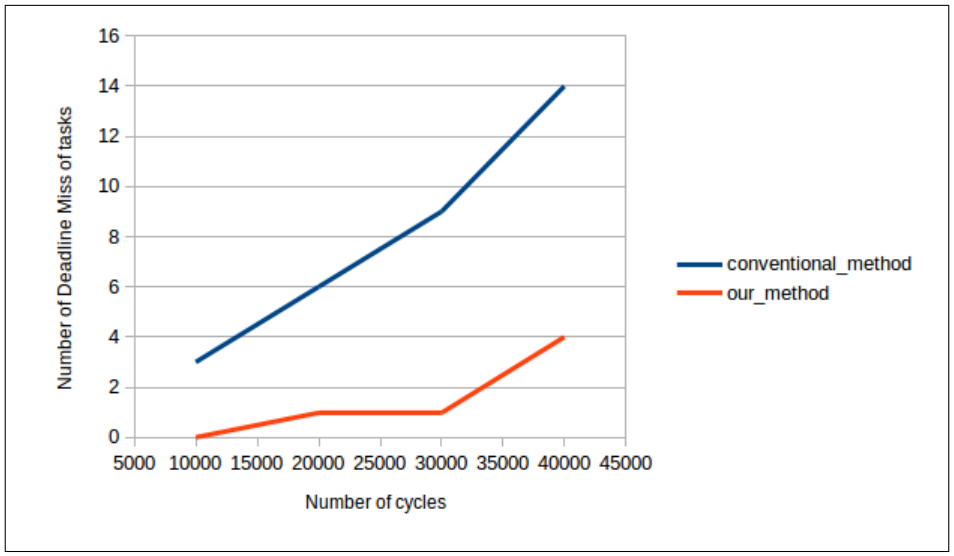

Figure 4.3: Number of Task Misses on Dataset II for a memory with 19 banks and row size 64B over 40,000 cycles

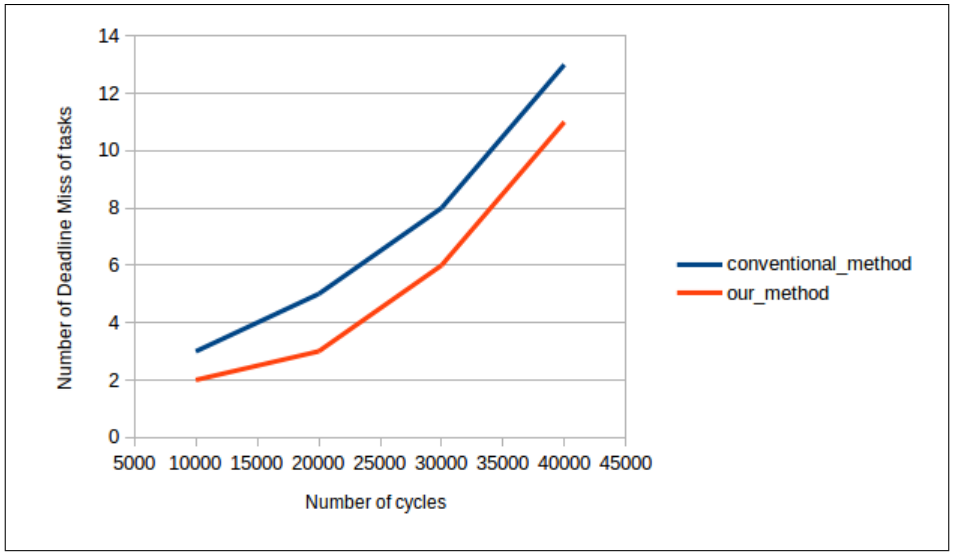

Figure 4.4: Number of Task Misses on Dataset II for a memory with 19 banks and row size 32B over 40,000 cycles

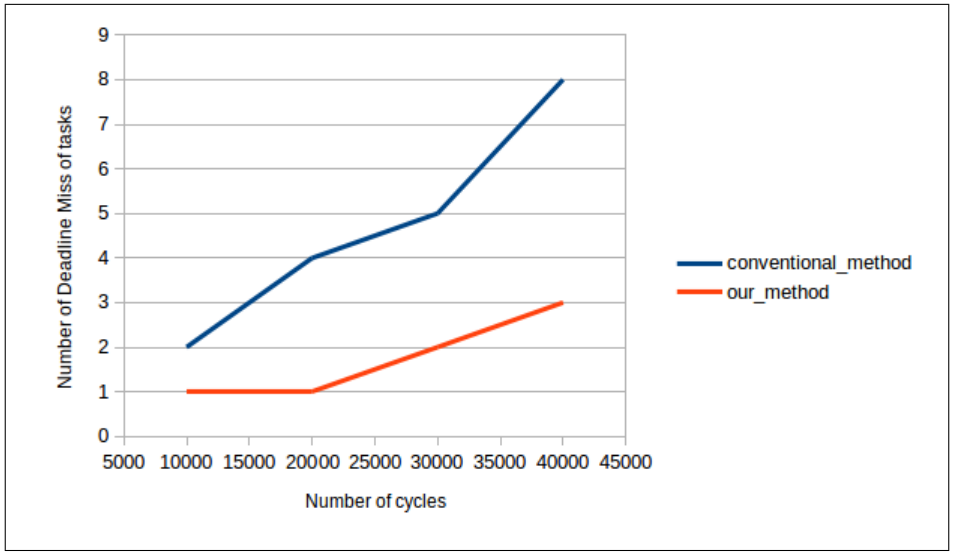

Figure 4.5: Number of Task Misses on Dataset III for a memory with 15 banks and row size 64B over 40,000 cycles

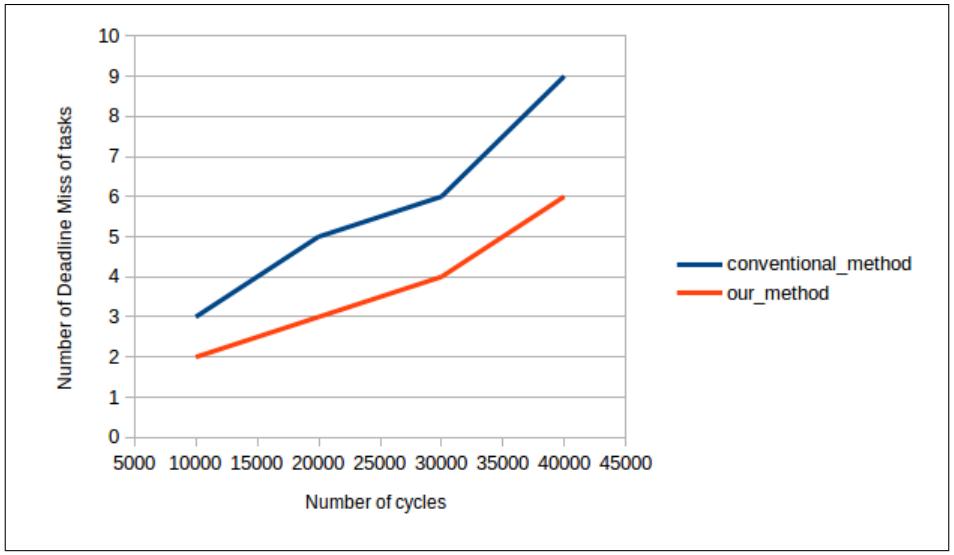

Figure 4.6: Number of Task Misses on Dataset III for a memory with 15 banks and row size 32B over 40,000 cycles

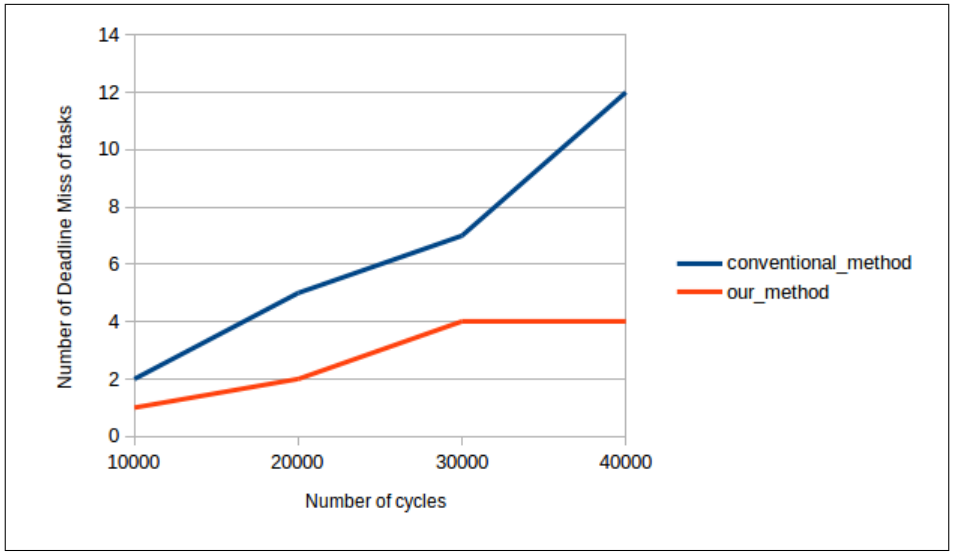

Figure 4.7: Number of Task Misses on Dataset IV for a memory with 12 banks and row size 64B over 40,000 cycles

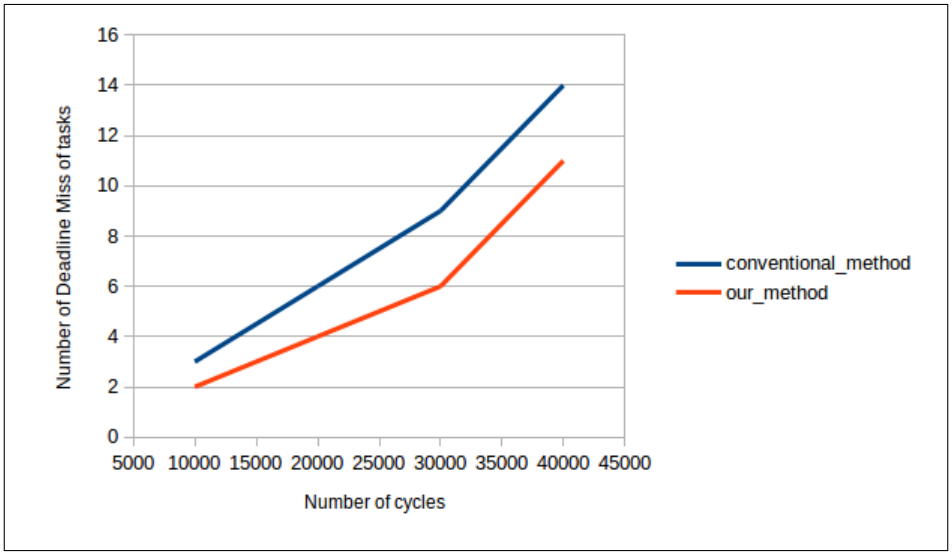

Figure 4.8: Number of Task Misses on Dataset IV for a memory with 12 banks and row size 32B over 40,000 cycles

Chapter 5

Conclusion

In this dissertation, we study the problem of memory scheduling for mixed criticality systems. We first take up the problem of deciding the required number of banks that can ensure a given set of mixed criticality tasks can meet their deadlines. We study both the decision and optimization problems for the same. Following this, we take up the issue of bank scheduling. The major problem with the conventional open row policy at the DRAM is that most of the high criticality tasks fail to meet their deadlines while waiting for memory access if not scheduled and executed within their deadlines. We propose a heuristic to address this limitation. Experimental results on a number of task sets with varying characteristics show the efficiency of our methods. We believe our research will open up future avenues in memory scheduling of mixed criticality systems.

Bibliography

[1] Akesson, B., and Goossens, K. Architectures and modeling of predictable memory controllers for improved system integration. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011* (2011), IEEE, pp. 1–6.

[2] Akesson, B., Steffens, L., Strooisma, E., and Goossens, K. Real-time scheduling using credit-controlled static-priority arbitration. In *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA'08. 14th IEEE International Conference on* (2008), IEEE, pp. 3–14.

[3] Burns, A., and Davis, R. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep* (2013).

[4] Fiduccia, C. M., and Mattheyses, R. M. A linear-time heuristic for improving network partitions. In *Papers on Twenty-five years of electronic design automation* (1988), ACM, pp. 241–247.

[5] forum, M. C. Mixed criticality system, 2017.

[6] Garey, M. R., and Johnson, D. S. *Computers and intractability*, vol. 29. wh freeman New York, 2002.

[7] Giannopoulou, G., Stoimenov, N., Huang, P., and Thiele, L. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *Proceedings of the Eleventh ACM International Conference on Embedded Software* (Piscataway, NJ, USA, 2013), EMSOFT '13, IEEE Press, pp. 17:1–17:15.

[8] Goossens, S., Kuijsten, J., Akesson, B., and Goossens, K. A reconfigurable real-time sdram controller for mixed time-criticality systems. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2013 International Conference on* (2013), IEEE, pp. 1–10.

[9] Gustafsson, J., Betts, A., Ermedahl, A., and Lisper, B. The mälardalen wcet benchmarks: Past, present and future. In *OASIcs-OpenAccess Series in Informatics* (2010), vol. 15, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[10] Hassan, M. Predictable shared memory resources for multi-core real-time systems.

[11] Kim, H., Broman, D., Lee, E. A., Zimmer, M., Shrivastava, A., and Oh, J. A predictable and command-level priority-based dram controller for mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE* (2015), IEEE, pp. 317–326.

[12] Kim, H., De Niz, D., Andersson, B., Klein, M., Mutlu, O., and Rajkumar, R. Bounding memory interference delay in cots-based multi-core systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th* (2014), IEEE, pp. 145–154.

[13] Nandi, B. B., Banerjee, A., Ghosh, S. C., and Banerjee, N. Dynamic SLA based elastic cloud service management: A saas perspective. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), Ghent, Belgium, May 27-31, 2013* (2013), pp. 60–67.

[14] Nandi, B. B., Ghosh, S. C., Banerjee, A., and Banerjee, N. Customer on-boarding strategies for cloud computing services with dynamic service-level agreements. *Service Oriented Computing and Applications 11*, 1 (2017), 47–63.

[15] Paolieri, M., Quiñones, E., and Cazorla, F. J. Timing effects of ddr memory systems in hard real-time multicore architectures: Issues and solutions. *ACM Transactions on Embedded Computing Systems (TECS) 12*, 1s (2013), 64.

[16] Ray, S., Dasgupta, P., and Chakrabarti, P. P. A new pseudo-boolean satisfiability based approach to power mode schedulability analysis. In *20th International Conference on VLSI Design (VLSI Design 2007), Sixth International Conference on Embedded Systems (ICES 2007), 6-10 January 2007, Bangalore, India* (2007), pp. 95–102.

[17] Reineke, J., Liu, I., Patel, H. D., Kim, S., and Lee, E. A. Pret dram controller: Bank privatization for predictability and temporal isolation. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2011 Proceedings of the 9th International Conference on* (2011), IEEE, pp. 99–108.

[18] Rixner, S., Dally, W. J., Kapasi, U. J., Mattson, P., and Owens, J. D. Memory access scheduling. *SIGARCH Comput. Archit. News 28*, 2 (May 2000), 128–138.

[19] Wikipedia. Dynamic random-access memory — wikipedia, the free encyclopedia, 2017. [Online; accessed 10-April-2017].

[20] Wikipedia. Earliest deadline first scheduling — wikipedia, the free encyclopedia, 2017. [Online; accessed 11-April-2017].

[21] Wikipedia. Partition problem — wikipedia, the free encyclopedia, 2017.

[22] Wikipedia. Rate-monotonic scheduling — wikipedia, the free encyclopedia, 2017. [Online; accessed 11-April-2017].

[23] Wikipedia. Safety-critical system — wikipedia, the free encyclopedia, 2017.

[24] Wikipedia. Scheduling analysis real-time systems — wikipedia, the free encyclopedia, 2017.

[25] Yun, H., Mancuso, R., Wu, Z.-P., and Pellizzoni, R. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th* (2014), IEEE, pp. 155–166.

[26] Zhang, F., and Burns, A. Schedulability analysis for real-time systems with edf scheduling. *IEEE Transactions on Computers 58*, 9 (2009), 1250–1258.