Multi-resoruce Fair Scheduler for Linux

by

Zehan Gao

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2022

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Resource management is one of the main responsibilities of operating systems. In Linux, the Completely Fair Scheduler (CFS) allocates CPU time to processes, allowing them to share CPU time. Although effective in allocating CPU time, CFS does not consider the consumption of other system resources (e.g., memory bandwidth, I/O devices, and power supply). It has been shown that the contention on memory bandwidth has a significant impact on the performance of processes [13]. A proper solution for memory bandwidth allocation should consist of a source throttling part and a memory request scheduler. In this work, we consider the multi-resource fair scheduling problem. We focus on CPU and memory bandwidth as the main resources. We take a software-hardware co-design approach to design and implement our proposed multi-resource fair scheduler, Dominant Resource Fair Queueing Scheduler (DRFQS). First, we propose simple modifications to the memory controller to monitor memory bandwidth usage per process and schedule memory requests. Second, we propose DRFQS to replace the CFS in Linux. DRFQS schedules processes based on the dominant resource fair queueing (DRFQ) method [9]. DRFQS utilizes the memory monitoring module and controls the memory scheduler in the modified memory controller. We implement our proposed memory controller design in the `gem5` simulator and implement our CPU scheduler based on Brain Fuck Scheduler (BFS) in the Linux kernel v4.8. We evaluate our design and implementation by: (i) measuring allocation of CPU time and mmoery bandwidth, and comparing them with the desired allocations, (ii) running benchmarks under memory bandwidth contention and comparing their performance with the original hardware and software and with our proposed modifications. Our evaluation demonstrates that our design guarantees fairness while achieving high performance. We reduce the running time of memory-intensive benchmarks under memory bandwidth contention by close to 50%.

Acknowledgements

I would like to thank my supervisor Seyed Majid Zahedi for his dedication, encouragement, and guidance. This work and thesis would not have been possible without his insight and constant support. I sincerely thank Professor Hiren Patel and Professor Rodolfo Pellizzoni for reviewing this thesis.

Dedication

This is dedicated to the world and the people I love.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Resource sharing is a main building block of most computer systems from embedded systems to personal computers to datacenter clusters [5, 26]. In most systems, the operating system facilitates resource sharing among processes. In many operating systems, including Linux, processes are scheduled to have a fair share of CPU time. However, the consumption of CPU time does not always directly correlate with the consumption of other system resources. Therefore, fair allocation of CPU time does not guarantee fair allocation of other resources [10, 9]. For example, the consumption of memory bandwidth by DMA-capable devices is not considered when scheduling processes [18]. With DMA-capable devices, a process can generate a large amount of memory traffic. The process for which DMA memory requests are generated might not use CPU while the requests are being processed. However, the consumed memory bandwidth slows down other processes that run on the CPU and require memory. To demonstrate this problem, we run two processes in a target simulated ARM system (see Section 5.1 for details). Process A generates a large number of DMA memory requests, and process B generates a large number of CPU memory requests. When process A and B run together, process B is slowed down, in some cases by a factor of more than 2, while process A does not experience any meaningful slowdown. In our experiment, we use the `fio` benchmark as process A and `mbw`, `lbm`, `Gems`, and `mcf` (see Section 5.2 for details) as process B. The slowdown of all benchmarks is shown in Fig. 1.1. In the experiment, the processes do not compete for CPU. Therefore, the result shows that the design of today's memory controllers makes DMA requests prioritized. It is also because the Linux scheduler, the Completely Fair scheduler (CFS), does not support

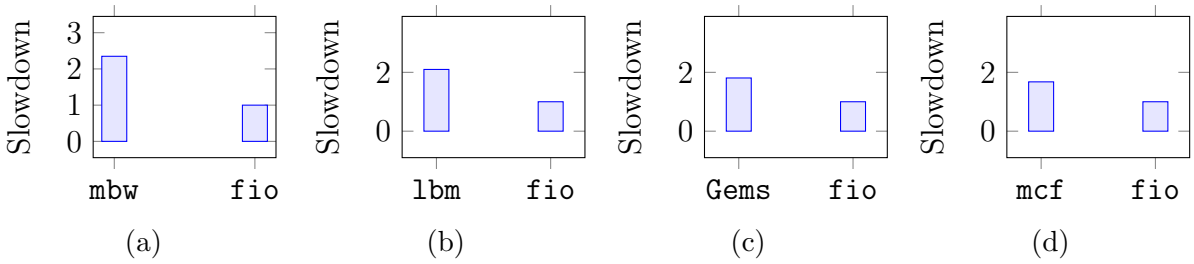


Figure 1.1: Slowdown of Benchmarks when Running Concurrently

multi-resource fair allocation of CPU time and memory bandwidth. If the two processes compete for CPU, the slowdown would be worse for process B compared to process A under CFS. CFS schedules processes based on their CPU usage. Process A consumes much less CPU time than process B to generate the same number of memory requests.

The example above shows that failing to consider contention on multiple resources could violate fairness among processes. For both embedded systems and cloud servers, unfair allocation of resources could lead to critical problems. In embedded systems, the completion time of critical and real-time tasks may increase and become unpredictable [1, 25]. In cloud servers, it can lead to the violation of service level agreement (SLA) [33], or less utilization of resources if inefficient performance isolation is used [10].

1.2 State of the Art

To allocate other resources, Linux supports `cgroup`, which sets a budget on multiple resources [2] (e.g. memory space, disk I/O, and network bandwidths) for a group of one or multiple processes. When a group of process(es) exceeds its budget on a resource, they are throttled from consuming more of that resource. However, `cgroup` does the allocation on a per-resource basis and lacks support for memory bandwidth allocation.

Prior work can be divided into two main categories: multi-resource fair allocation and memory bandwidth allocation.

1.2.1 Multi-resource Fair Allocation

Space-shared Resources. To allocate multiple resources fairly, Ghodsi et al. proposed Dominant Resource Fairness (DRF) [10]. The resource that a user has the highest

share of is the dominant resource of the user. In a multi-user environment, the dominant resource of different users might be of different types. DRF maximizes the minimum share of the dominant resource across all users. This means that all users are guaranteed a fair share of their dominant resource. DRF incentivizes users to share the resources with other users. In most cases, users get more resources under DRF than the equal partitions of all system resources. In the worst case, users are guaranteed to get the same amount of resources as equal partitions (see Section 2.4.1 for details).

Time-shared Resources. As an extension to DRF, Ghodsi et al. proposed Dominant Resource Fair Queueing (DRFQ)[9] for allocating time-shared resources. The proposed scheduler is based on virtual time. The scheduler picks the next packet with the lowest virtual time to be serviced when the resource it requires becomes available. The virtual time of a packet when the system is backlogged is based on the dominant resource time consumption of the previous packet of the same user.

1.2.2 Memory Bandwidth Allocation

The memory bandwidth allocation problem consists of two parts: (i) monitoring the memory bandwidth consumption of processes and (ii) enforcing the allocation. Prior works monitor the memory bandwidth consumption of processes running on CPU by utilizing the Performance Monitoring Unit (PMU) [32, 17]. Monitoring the per-process memory bandwidth consumption by DMA-capable devices using existing hardware features (e.g., IOMMU) is also discussed in [7], although existing hardware features are not sufficient for such monitoring. Enforcing the memory bandwidth allocation can be implemented at the source of the memory requests (e.g., CPU) [32, 13] or at the target (i.e., the memory controller) [22, 8, 13]. However, source-based only and target-based only solutions are demonstrated to be insufficient, and a solution that combines both source-based and target-based enforcement is more effective [13].

1.3 Methodology

In this thesis, we consider the problem of multi-resource fair scheduling in modern computer systems. To address this problem, we propose the Dominant Resource Fair Queueing Scheduler (DRFQS) to achieve fair allocation of CPU time and memory bandwidth in Linux. We take a software-hardware co-design approach to design and implement DRFQS. The main idea is to consider processes as packet flows and each run of the process

as a packet. We design and implement a CPU scheduler for DRFQS which is an extension of DRFQ and relies on the hardware to provide information about the memory bandwidth consumed by each process. The memory bandwidth allocation is enforced by source-based throttling provided by our proposed CPU scheduler and target-based arbitration provided by our proposed memory scheduler hardware.

To allocate memory bandwidth, we add performance counters and a control interface to provide per-process memory bandwidth consumption information to the kernel CPU scheduler. To track memory usage per process, we use different approaches for CPU and DMA-capable devices. For CPU memory requests, since there is only one process running on a CPU core at any given time, we use the core ID and the mapping from core ID to process ID (PID) provided by the CPU scheduler to identify the process responsible for CPU memory requests. Prefetch and writeback requests generated by the cache system as a result of a CPU memory request are attributed to the same process responsible for the CPU memory request. For DMA-capable devices, we use the `subStreamID` (ARM) field in memory request packets to provide the PID of the responsible process to the memory controller. To evaluate our design, we use the `gem5` simulator. We also use SimpleSSD, an NVMe SSD simulation as the DMA-capable device. We modify the NVMe protocol, the NVMe driver in Linux, and SimpleSSD to support using PID as `subStreamID`. We verify our memory bandwidth tracking design to ensure that it tracks the memory bandwidth consumed by processes through different routines.

We modify the memory scheduler in the memory controller to make it use the virtual time of processes for scheduling. The priority of memory requests is determined by the virtual time of the responsible process. Requests generated by a process with lower virtual time have higher priority. The scheduler picks the highest priority request from all queued requests to be processed. The virtual time of processes is provided by the CPU scheduler by writing to registers in the memory controller. The design of the queues of the memory scheduler is also modified to better handle high request rate.

We implement our kernel CPU scheduler based on CFS in Linux. We replace the per-CPU run queue design of the CFS with a single-queue design. We implement our single-queue design based on the code of the Brain Fuck Scheduler (BFS) [30]. Following DRFQ, we use virtual time in our proposed scheduler. The scheduler reads the data from the performance counters in the memory controller, makes scheduling decisions, and writes to the registers to control the memory scheduler.

To summarize, our work consists of the following parts:

1. We propose a design tracking per-process memory bandwidth usage by CPU and DMA-capable devices. The design is implemented in the kernel scheduler, the device driver,

and the memory controller (Chapter 3).

2. We propose a CPU scheduler using virtual time based on the usage of both CPU time and memory bandwidth. The calculation of virtual time is based on the Dominant Resource Fair Queueing [9]. We implement the scheduler Linux kernel v4.8 using the BFS scheduler (Section 4.2).

3. We propose a memory request scheduler in the memory controller with input from the CPU scheduler. We evaluate our design using `gem5` (Section 4.4).

4. We address the problems of memory bandwidth allocation and performance turbulence caused by the difference between the original use case of DRFQ and our target system. We introduce delayed wakeup for DMA-intensive processes for fairness (Section 4.5.1). We also use a smoothened virtual time as the input for the memory packet scheduler to reduce performance turbulence for memory-intensive processes (Section 4.5.3).

Chapter 2

Background and Related Works

In this chapter, we first review previous works on CPU scheduling and memory bandwidth scheduling. Then, we introduce the multi-resource fair allocation problem. Finally, we provide a brief overview of the dominant resource fair (DRF) allocation and its extension to time-shared resources, dominant resource fair queueing (DRFQ).

2.1 Resource Allocation Model

Resource sharing is a classic topic in computer systems. We consider a typical multiple-user computer system, in which each user can run multiple instances of different applications. Each instance of an application is a task. In process-based operating systems, the task is described as a process. Resources that different users may compete on include CPUs, memory capacity, memory bandwidth, and external devices, e.g., disks, network cards, and GPUs. In general, resources can be shared in two ways: (i) space-shared and (ii) time-shared. With space-shared, allocations remain constant over time. With time-shared, time is divided into time slices, and allocation may change from one time slice to another. It can also be said that space-shared resource is a special case of time-shared resource. When a space-shared resource is allocated to a user, the user can use it for unlimited time slices.

In process-based operating systems, the CPU is usually time-shared. To allow different processes to run "concurrently" on CPU, operating systems schedule processes to run for a time slice. After the time slice is expired, the process may be replaced by another process. In other computer systems, the CPUs can be treated as a space-shared resource. For

example, big data frameworks like Hadoop allocate CPU cores and memory space to tasks. A task has full access to the resource allocated to it until it finishes. In this case, CPU cores and memory space are space-shared resources.

Fairness for space-shared and time-shared resources is evaluated based on the share of each resource allocated to each user. For space-shared resources, the share allocated to the user is the amount of the resource allocated to the user divided by the total amount of the available resource. For time-shared resources, it is the total time allocation on the resource during the allocation period multiplied by the average share of allocation then divided by the length of the period. The length of the allocation period varies based on how the system measures resource allocation.

For space-shared resources, the share of allocation is determined before the user starts to run the application. The system usually does not know the actual utilization of the allocated resources. For time-shared resources, the system usually knows the actual consumption of the resource. Spillover time-shared resources can be reclaimed to be used by other users. For example, if a process is blocked before it consumes all allocated CPU time, the scheduler picks another process to run. If all other processes have consumed their allocation, the scheduler still picks one of them to run and consume the spillover CPU time.

2.2 CPU Scheduling

Round-robin is a classic CPU scheduling policy. In round-robin scheduling, time is divided into multiple time slices. Each time slice is allocated to the process at the head of the run queue. Processes take turns running on CPU for the period of the time slice in circular order. To support priority-based scheduling, two common techniques are weighted round-robin (WRR) scheduling and strict priority (SP) Scheduling [23]. Weighted round-robin scheduling makes the length of time slices for each process proportional to the weight of the process [19]. Strict priority scheduling assigns fixed priority to each process. Processes with lower priority execute only when there is no higher priority process ready.

Based on weighted round-robin and strict priority, multilevel queueing (MQ) scheduling classifies processes into different queues of fixed priority. Lower priority queues execute only when there is no task in higher priority queues ready. Each queue has its own scheduling policy, e.g., round-robin or weighted round-robin.

Waldspurger et al. [29] propose lottery scheduling and stride scheduling [28] to achieve QoS in scheduling. Lottery scheduling assigns processes tickets in proportion to their

weights. It guarantees that all processes get at least one ticket. The scheduler randomly picks a ticket and schedules the process which owns the ticket to execute. The ticket is removed after being picked. When all tickets are removed, the scheduler assigns tickets to processes again. It ensures that every process will have a chance to execute by giving them at least one ticket.

Stride scheduling improves lottery scheduling by making the scheduling deterministic. It introduces the *stride* for each process, which is inversely proportional to the number of tickets held by the process. It also introduces the *pass* for each process, which increases by the stride each time a process receives a time slice. The scheduler picks the process with the lowest pass to execute, removing the randomness of lottery scheduling. Compared to weight round-robin, lottery scheduling and stride scheduling gives processes with higher priority more chances to get a time slice instead of giving them longer time slices. This prevents high-weight processes from occupying CPU for a long period of time, which in turn prevents other processes from starving.

In networking, to achieve "perfect" theoretical fairness, Generalized Processor Sharing (GPS) provides an ideal algorithm to share a network link among packets from multiple flows with different weights. With GPS, the bandwidth allocated to each flow is proportional to its weight. GPS is only a theoretical benchmark and is not practical because it assumes that the network traffic can be arbitrarily split. An approximation of GPS is the Weighted Fair Queueing (WFQ) [3]. Virtual time is introduced in the implementations of WFQ as a simplified description that links resource consumption and progress of a flow to its priority [3]. Virtual time allows flexible manipulation of the priority of processes. For example, the weight of a process determines the ratio between virtual time increment and actual time consumed.

The Linux CPU scheduler is called the Completely Fair Scheduler (CFS) [12]. CFS is considered to be an implementation of WFQ. In CFS, each CPU has its own run queue. Ready processes (i.e., in `TASK_RUNNING` state) are assigned to a CPU and added to its run queue. In each run queue, process descriptors are sorted by their virtual time and stored in a red-black tree (rbtree) data structure. When choosing the next process to run on CPU, CFS picks the process that has the lowest virtual time. The virtual time of a process is calculated based on the CPU time allocated to the process in the past. By picking the process with the lowest virtual time, CFS maximizes the minimum share of CPU time for all processes. When a process is running, its virtual time is updated periodically every time the timer interrupt is triggered. When a process is blocked, suspended, or yields, its virtual time is also updated. In these cases, the virtual time is updated based on the CPU time consumed and the weight of the process after the previous update. When a process is unblocked, its virtual time is updated to a value slightly lower than the minimum virtual

time of the run queue, if this value is not lower than its original virtual time. The minimum virtual time is a monotonically increasing value maintained by the scheduler. It is updated when the minimum virtual time across all processes in the run queue increases. It is added to the virtual runtime of a process when it is assigned to a run queue and subtracted from a process that is being migrated from the run queue.

To balance the load among CPU cores, CFS calculates the load value for each process and each run queue. The load value of a process is based on its CPU time consumption in the past and its weight. The load value of a run queue is the summation of the load value of all processes in the queue. CFS compares the load value of different queues to determine if there is an imbalance. If there is a significant imbalance, CFS performs migration to balance the workload between the CPUs. This ensures processes do not get unfairly delayed because the CPU they are assigned to is overloaded.

Despite being the default scheduler of Linux for years, CFS suffers from some main weaknesses. First, it does not fully support latency-sensitive workloads [6]. Consider a latency-sensitive process **A**, that receives a packet once every 16 time slices and it takes 4 time slices to process the packet. Suppose that the application runs in a run queue with 3 other processes which are always ready or running. When a packet arrives, **A** is unblocked and starts running. **A** runs for a time slice each time it is scheduled to run. After that, it waits for the 3 other processes to each run for a time slice. As a result, after the packet arrives, **A** is scheduled to run in the 1^{st}, 5^{th}, 9^{th}, and 13^{th} time slices. While the processing time is 4 time slices in total, the response time of **A** is 13 time slices. If other processes are not latency-sensitive, the ideal scheduler should let A run for 4 time slices uninterrupted every 16 time slices. After that, **A** is blocked until the next packet arrives. The other 3 processes are scheduled to each run for 4 time slices from the 12 time slices left. As a result, in every 16 time slices, all 4 processes have run for 4 time slices. In this case, the response time of A is minimized, while CPU time is fairly allocated among processes over a longer time period (16 time slices).

Duda et al. propose the Borrowed-Virtual-Time (BVT) [6] scheduler to improve the performance of latency-sensitive workloads while preserving the fairness in CPU time allocation. BVT introduces the *effective* virtual time for latency-sensitive processes, replacing their actual virtual time. The effective virtual time of a latency-sensitive process is its actual virtual time subtracted by its *warp* value. The warp value of a process is the virtual time the process is configured to borrow. This gives latency-sensitive processes the advantage to be scheduled first and keep running on the CPU for a longer time period. However, the borrowed virtual time can be quickly consumed. If that happens, the process is blocked after which the borrowed virtual time is recharged. In the example mentioned previously, **A** can be scheduled ideally by BVT if its warp value is configured to be 3 time

slices or more.

The second problem with CFS is the overhead of implementing the multi-queue design in CFS. By using separate run queues for each CPU core, CFS minimizes the lock contention among CPU cores [12]. However, balancing the load among multiple queues introduces a significant overhead [20]. An alternative scheduler is the Brain Fuck Scheduler (BFS) [30]. In BFS, a single global run queue is shared among all the CPU cores. CPU cores can access the global run queue by acquiring a global lock. BFS uses virtual deadline to determine which process to schedule next. When a CPU becomes idle, it queries the global queue for the next process with the lowest virtual time. The virtual deadline for each process is calculated based on the current time and the weight of the process (which is determined by a *nice* value). BFS also supports different scheduling policies for real-time tasks created by root users and unprivileged users, normal tasks, and background tasks.

BFS handles CPU affinity by calculating the distance between cores. For each pair of CPU cores, a distance value is set during kernel initialization. The distance indicates the overheads when migrating a process from one CPU core to the other. If the memory access is uniform among all cores (i.e., non-uniform memory access (NUMA) is not enabled), BFS sets the distances between every pair of CPU cores to be the same value. When picking the next running process, the virtual time is multiplied by 2 to the power of such distance. This way, the migration would not happen unless the CPU is significantly backlogged.

Since the CPUs need to acquire a global lock to access the single run queue, lock contention becomes a problem for BFS when the number of CPUs increases. However, because of its simplicity, when the number of CPUs is small, especially when there are less than 16 CPUs, BFS shows better performance than CFS for many benchmarks [11].

Lastly, CFS is also found to have unexpected bugs in multi-core systems [20]. In modern multi-core systems, when NUMA is enabled, a set of CPU cores, called a NUMA node, shares the same local memory. The access time of non-local memory is longer than local memory. To adapt to NUMA, CFS first performs group load balancing between NUMA nodes, then balances the CPU cores in each NUMA node. This reduces the migration of processes between cores of different NUMA nodes. Lozi et al. [20] found that the group balancing algorithm in CFS is problematic. It leaves cores idle while other cores are busy in some special cases.

2.3 Memory Bandwidth Scheduling

The solutions to memory bandwidth scheduling can be categorized into two types: (i) source-based and (ii) target-based. In source-based solutions, the request generation rate is throttled at the source (e.g., CPU) to reduce or eliminate backlog at the target, i.e., the memory controller. In target-based solutions, when there is a backlog at the target, requests with higher priority are prioritized over others. The priority of memory requests from different sources in a target-based solution is usually based on the memory bandwidth usage by the source in the past or the progress of tasks that should be prioritized. To collect the data on memory bandwidth usage, most solutions rely on hardware features. We first review source-based solutions using existing hardware features. Then, we review target-based solutions, which usually need new hardware features. Finally, we review combined source-based and target-based solutions which provide better outcomes than source-based or target-based alone.

2.3.1 Source-based Solutions

Software-only Solutions

`cgroup` [24] is a kernel feature in Linux for resource management. It supports limitation, accounting, and isolation of resource consumption for many different resources, such as CPU, memory, disk I/O, and network. While `cgroup` does not manage memory bandwidth, it does manage disk I/O and network which are major sources of memory bandwidth usage other than CPU. `cgroup` relies on the Linux subsystem that manages the resource to collect resource consumption data and apply restrictions. For example, the support for disk I/O relies on the block I/O (`bio`) layer [15]. `cgroup` first attributes I/O requests to a process if it is not generated by the kernel. For non-buffered requests, e.g., all read requests and direct write requests, the requesting process is obvious. For buffered write requests, data written to the device is stored in a write buffer until a write-back is triggered. A write-back can be triggered by the requesting process, other processes, or the kernel. Thus, `cgroup` relies on the memory subsystem to identify the owner of the page for the requested data and counts it as the requesting process. The block I/O subsystem reports to `cgroup` when a block I/O request starts or finishes. `cgroup` can throttle requests from a group by limiting the group's block I/O queue depth or introducing artificial delays.

While `cgroup`'s management of disk I/O and network affects the memory consumption by external devices, it does so indirectly. This is insufficient to fairly allocate memory bandwidth among processes for four main reasons. First, it does not support accounting

and limiting normal memory bandwidth consumption generated by CPU reads and writes. Second, `cgroup` does not track the exact time of each memory operation. In the example of block I/O, `cgroup` only tracks the size of requests and the start and the end time of the request. The memory traffic associated with a request may be distributed unevenly during the period the request is being serviced by the hardware. As a result, there are inaccuracies when estimating memory bandwidth consumption, especially when the service time is long. Third, the size of requests may be different from the amount of memory traffic generated by the device. Common reasons for this are the uncertainty of hit or miss in the cache system (if I/O devices use cache) and the device may use the main memory to store intermediate results (e.g. FPGA accelerators). Lastly, `cgroup` manages different types of resources separately. If a group uses multiple devices, e.g., disk I/O and network, estimating the combined memory bandwidth consumption is almost impossible.

Prior work has explored using the existing Linux subsystems to manage per-process memory bandwidth consumption. Ewert et al. [7] propose solutions to estimate the memory bandwidth usage of processes by disk I/O and network. The authors explore both software and hardware solutions. The authors conclude that current hardware can provide accurate data for total memory bandwidth consumption by I/O devices, but it cannot provide memory bandwidth consumption by each device or each process. Following the implementation of `cgroup`, device drivers are modified to obtain memory bandwidth usage by each process from disk and network devices. Assuming that the actual memory bandwidth usage of I/O operation is proportional to the usage reported by device drivers, the authors use the proportion from the driver-reported memory bandwidth and the hardware-reported total memory bandwidth by all I/O devices to calculate the memory bandwidth usage by each process. However, the assumptions made in this work are completely accurate. The authors' design only updates the memory bandwidth usage of a process when it is dequeued. If the memory traffic between two updates distributes unevenly over time, then this method suffers from inaccuracies similar to `cgroup`.

Hardware-assisted Solutions

Existing hardware provides features that can be used to monitor the memory bandwidth usage of CPUs and I/O devices. In most modern CPUs, the Performance Monitoring Unit (PMU) provides a set of hardware counters that can be utilized to track system events. Usually, the PMU supports counting the number of retired instructions of each type, the cycles CPU spent on each status (e.g., memory stall), and the number of cache hits and misses. The PMU counters can also trigger interrupts. Prior works utilize the PMU counters to monitor system status and apply measures to manage memory interference.

Using the PMU, Kim et al. [17] propose memory-aware CFS (mCFS), a modification to Linux CFS to compensate processes for slowdowns due to memory contention. mCFS is implemented and tested on NVIDIA Xavier SoC, which is primarily used in embedded systems. Xavier integrates a high-performance GPU that can be used to run machine learning workloads. The authors demonstrate the performance impact of memory contention between CPU cores and between CPUs and GPUs. In mCFS, the PMU is used to measure the backend stall cycles and total CPU cycles. The authors define the *intrinsic* stall and the *memory-related* stall to distinguish the cause of stalls. The authors also define the *actualized scaled* CPU time to compensate for memory-related stall cycles when counting the CPU time consumed by the processes. The actualized scaled CPU time is used to calculate virtual time, which in turn is used to schedule processes.

Tang et al. [26] analyze the memory performance in datacenter applications running on multi-socket servers. When multiple threads run on hardware with non-uniform memory access and multi-level caches, modification of shared memory by one thread may cause other parts of the system to invalidate the caches storing the same address. The location of these threads on the CPUs has a significant impact on cache performance. The authors illustrate the destructive impact on cache performance caused by improper thread location and develop heuristic and adaptive approaches to co-locate threads properly.

In real-time applications, memory contention makes memory-related delays unpredictable. Thus, the operating system cannot predict if a real-time task can meet its deadline. MemGuard [32] is a solution addressing the problem. MemGuard regulates the total memory request rate to be lower than the DRAM controller's service rate and ensures a minimum memory bandwidth allocation for processes. The implementation of MemGuard is a patch to the Linux kernel. The kernel patch uses the PMU counters to count the number of last-level cache (LLC) misses. The memory traffic of processes is estimated by the number of cache misses on partitions of the cache mapped to the CPU core the process runs on. The authors use an empirical value for the DRAM controller's bandwidth, which is the total budget of all processes. When scheduling a process to run on CPU, MemGuard calculates the memory bandwidth budget for the process. The budget is converted to a threshold and used to configure the PMU counters according to the technical specifications. When the counter reaches the threshold, it indicates that the process has used up the memory traffic allocation in the current scheduling period. The PMU generates an overflow interrupt. MemGuard services the interrupt and stops the process from running. If there is unused memory bandwidth budget reclaimed from processes that have not consumed up all their allocated budget, the budget is transferred to the interrupt process. Then the interrupted process can run until it consumes its new budget.

MemGuard has been used to solve many problems caused by memory interference.

Schwäricke et al. [25] built a solution to real-time virtual machine (VM) communication using MemGuard. The authors argue that the increasing usage of GPUs and FPGAs for computing requires large data transfers to and from memory. They consider the transfer between applications running on different virtual machines that share devices like GPUs and FPGAs. The virtual machines use the `virtio` interface provided by the hypervisor to communicate with each other. Communication between virtual machines requires the copying of a large amount of data. In the target platform, QoS-regulated DMA engines are used to copy large chunks of data within the memory. Memory interference exists among the CPU cores and the DMA engines (The authors exclude other DMA-capable devices). To address this problem, the authors extend the hypervisor to control MemGuard and QoS DMA engines to regulate the memory usage by CPU and DMA engines. The hypervisor schedules the virtio packet used for VM communication. This way, since the memory bandwidth regulation minimizes memory interference, the VM communication time is predictable with the solution.

An extension of MemGuard, BWLOCK [31], addresses the performance degradation of applications that access memory in bursts, e.g. multimedia decoders. This type of application is sensitive to memory latency but has a low average memory bandwidth consumption over a long period. The authors define the memory-performance critical sections for these applications. BWLOCK provides a user-level API for user applications to acquire a lock on memory before entering such sections. Processes with the lock are allowed to use memory without restrictions, while others are regulated to use a small constant amount of memory bandwidth.

Aghilinasab et al. [1] propose a solution based on BWLOCK protecting GPU from memory interference by CPU while maximizing the memory bandwidth available to CPU and ensuring that tasks on GPU meet their deadline. Real-time tasks acquire the lock when using the GPU. The impact of memory interference on DMA operations of GPU is minimized by BWLOCK by restricting memory usage of CPU cores. To maximize the memory bandwidth available to CPU, the proposed solution estimates the finish time of GPU tasks based on the progress, the estimation of worst-case execution time (WCET), and the current time. Compared to BWLOCK which allocates a constant amount of memory bandwidth to CPU cores when the lock is acquired, the solution adjusts the amount based on the estimated finish time of the real-time GPU task. If the finish time is estimated to be earlier than the deadline, then the amount of memory bandwidth available to CPU is increased.

2.3.2 Target-based Solutions

In most existing systems, the memory controller schedules requests to maximize throughput. The status of DRAM when a request arrives affects the processing time of the request. A typical DRAM chip consists of multiple independent banks that allow requests destined for different banks to be processed in parallel. Each bank is organized as a two-dimensional array with multiple rows and columns and has one row buffer [21]. A memory address can be divided into bits storing the target bank, row, and column. To access data from a row, the row must be activated first and stored in the row buffer. Then, a read/write request is processed on the row buffer. This results in three different statuses when a request arrives, row hit (the row buffer has the requested row), row closed (the row buffer is not loaded with any row), or row conflict (the row buffer has another row that is different from the requested one), ordered by the resulting processing time from low to high. If row conflict happens, the row buffer always has to be written back. This is because the data is destroyed in the row after it is activated.

Resequencing the DRAM requests to have more row hits and reduce row conflicts can greatly improve the performance of DRAM. The memory controller often uses the first-ready first-come-first-serve (FR-FCFS) scheduling policy that maximizes row hits. It is an improvement of the first-come-first-serve (FCFS) scheduling policy. In FR-FCFS, the earliest row-hit request has the highest priority, while the latest row-conflict request has the lowest.

DRAM also has extra delays when switching between read and write commands. To reduce the overhead, the memory controller often minimizes the times of switching. If the request queue of the previous direction is not empty, the memory controller continues in the same direction. Only when there is no queued request in the previous direction, the memory controller switches the direction to process queued requests in the other direction.

To support fair scheduling of memory requests, prior solutions modify existing hardware. PARDIS [22] is a design of dedicated memory processors in the memory controller. The proposed memory controller consists of a request processor and a transaction processor. Both of them support running firmware code to control their behavior. The request processor receives request packets from the last level cache of CPUs and translates them into transactions. The transaction is the minimum unit of memory operations used as an intermediate representation in the processors. The transaction processor translates transactions into actual commands used by the memory. The authors evaluated the design by a software simulator for behavior and Verilog design for area and power consumption. The firmware used in the evaluation includes FCFS and FR-FCFS.

Usui et al. [27] propose a solution to make the scheduling of memory requests application-aware by categorizing memory requests. For CPUs, GPUs, and hardware accelerators, the latency and bandwidth requirements of memory access can be categorized into 6 types based on the application workloads and their stages. The memory requests are first prioritized based on their type, then with other properties, e.g., their deadline. The authors evaluated the design with a cycle-accurate simulator. Their results show that scheduling based on types outperforms scheduling policies that do not distinguish between the types of requests.

Fang et al. [8] address the problem of CPU-GPU memory contention with a solution dynamically recognizing the pattern of requests. The authors noted that the requests from GPU usually have high locality and high row-hit rate. The requests from CPU may be similar to GPU, but may also have low locality and low hit rate depending on the type of application. In their solution, the requests from CPU and GPU are isolated in different queues. Within the CPU queue, requests from each CPU core are distinguished based on locality. The authors propose dynamic bank partitioning, mapping requests with different characteristics to different bank sets, to eliminate memory interference of multiple CPU applications. Within the GPU queue, requests from each GPU core are assigned different criticality based on latency tolerance. A dynamic switching policy is also proposed to switch between criticality-based scheduling and locality-based scheduling (FR-FCFS). This balances fairness and performance. The authors evaluate their work on a simulator based on `gem5`, which is also used by other researchers studying CPU-GPU interference [16].

2.3.3 Combined Solutions

Hower et al. propose PABST [13]. The authors demonstrate that neither source-only nor target-only memory bandwidth management can provide sufficient fairness for different types of processes. Source-only solutions ignore the impact of latency on processes that sparsely traverse through a large range of memory. Target-only solutions cannot handle excess memory requests from processes that generate many requests in bursts. The authors propose a solution that consists of two parts: (i) a *governor* inside the L2 cache and (ii) a priority *arbiter* inside of the memory controller. When the arbiter reports saturation of the read queue in the memory controller, the governor throttles requests from each CPU core. The throttled request rate is proportional to the weight of each CPU core. This minimizes the overflow of memory requests at the memory controller and ensures fairness. The arbiter also tracks the memory bandwidth usage by each core. If the actual consumption of a CPU core is furthest behind its target consumption proportional to its weight, the requests from the CPU core are prioritized. As a result, processes that consume little memory bandwidth

but are sensitive to latency are prioritized. The solution is implemented and tested on an in-house, cycle-approximate simulator that models data center servers. Source-based only and target-based only solutions are also implemented for comparison. The results show that source-based only and target-based only solutions achieve target fairness in some different circumstances but fail in others. In comparison, PABST combines the benefits of both source-based and target-based solutions and achieves better fairness in most circumstances.

2.4 Multi-Resource Allocation

The definition of fairness and efficiency for single resource allocation is simple. For fairness, when all users have the same weight, no one should prefer the allocation of others. When they have different weights, their allocation should be proportional to their weight. For efficiency, all resources should be utilized if the users' demand reaches the system capacity. The system should not be able to allocate resources to a user without reducing the allocation of another user in such circumstances. When users demand more than one type of resource, it is hard to define fairness. The shares of allocation of different types of resources can be different. Multiple definitions are proposed to define the share of allocation of all resources from the shares of each resource. Based on these definitions, multiple scheduling policies are proposed.

We consider four properties when discussing multi-resource allocation [10]:

- **Sharing incentive**. All users are better off sharing resources than exclusively using a fair partition of the resources in isolation.
- **Strategy-proofness**. Users cannot increase their utility by misreporting their resource demands.
- **Envy-freeness**. Users always prefer their own allocation to the allocation of others.
- **Pareto efficiency**. The system utilization is maximized so that it is impossible to increase the utility of a user without decreasing the utility of another user.

Max-min allocation is commonly used in computer systems to achieve fairness and efficiency. Max-min guarantees a minimum share of resources for each user. Weighted max-min allocation guarantees each user a minimum share of resources proportional to the weight of the user. Max-min allocation can be applied to multiple resources. When doing multi-resource allocation, max-min guarantees a minimum share of each resource for each

user. However, the minimum share of each resource under max-min is the same. Such an allocation fails to consider the heterogeneity of demands. This reduces max-min allocation on multiple resources to a single-resource fair allocation on an abstract single resource that consists of slices of each resource.

Max-min allocation on multiple resources is not efficient. To see this, consider a system with 6 CPUs and 6GB of memory. Support that each task of user A's application requires 1 CPU and 2GB of memory and each task of user B's application requires 2 CPUs and 1GB of memory. The max-min allocation would give each of them 3 CPUs and 3GB of memory if they have the same weight. With such allocation, both users A and B can only run one task of their applications. As a result, only half of the system resources are utilized. If user A gets 2 CPUs and 4GB of memory, and user B gets 4 CPUs and 2GB of memory, both users can run 2 tasks of their applications. In this example, user applications have resource demand vectors that do not match the single resource abstraction. Using max-min allocation in such a case results in low performance for user applications and low utilization of system resources.

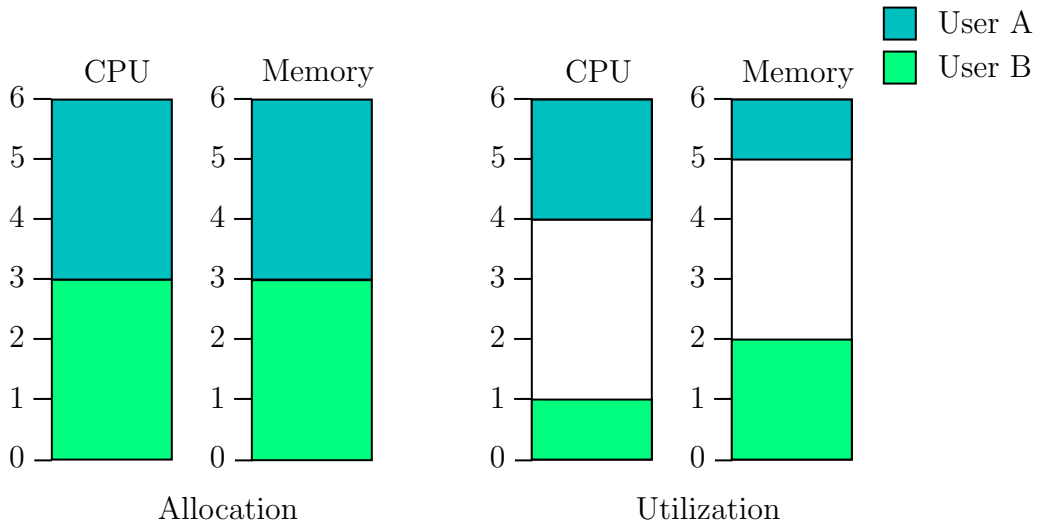


Figure 2.1: Allocation and Utilization of max-min

For time-shared resources, the difference in consumption of different resources also exists and can lead to similar problems with max-min allocation. Consider two time-shared resources, CPU time and memory bandwidth. If user A consumes mostly CPU time and negligible memory bandwidth, and user B consumes mostly memory bandwidth and negligible CPU time, the ideal allocation is to allocate a large share of memory bandwidth to user A and a large share of CPU time to user B. With max-min allocation, both users

get half of their desired type of resource, and the other half allocated to the other user remains unused.

2.4.1 Dominant Resource Fairness

Ghodsi, et al. [10] propose Dominant Resource Fairness Fair Allocation (DRF) that achieves multi-resource fairness. Unlike the max-min allocation, DRF calculates the share of different types of resources for each user. DRF equalizes and maximizes the share of dominant resource for all users. In a system with multiple resources, for each resource j, the capacity of that resource is denoted as r_j. The allocation of user i on resource j is denoted as $u_{i,j}$. The share of allocation of user i on resource j, $s_{i,j}$, is defined as:

$$s_{i,j} = \frac{u_{i,j}}{r_j} \tag{2.1}$$

The resource that a user has the highest share of is the dominant resource of the user. The share of the dominant resource of user i (the *dominant share* of user i), d_i, is defined as:

$$d_i = \frac{max_j\{s_{i,j}\}}{w_i}, j \in \text{resources} \tag{2.2}$$

In Equation 2.2, w_i is the weight of user. DRF allocates resources to users according to users' demands. To achieve the target multi-resource fair allocation, it picks the user with the lowest dominant share among those who have tasks ready to run. If the available resources can satisfy the demand of the next task of that user, the task is scheduled to run. The user's dominant share is also updated. DRF repeats this until the next task of the user with the lowest dominant share cannot be satisfied by available resources. At this point, the system achieves the target multi-resource fair allocation[1].

In the previous example, DRF would allocate 2 CPUs and 4GB of memory to user A and 4 CPUs and 2GB of memory to user B. The dominant share of both A and B is $\frac{2}{3}$ and the system is fully utilized. We explain how DRF meets the four desired properties using the previous example.

Sharing incentive. The users are better off sharing all system resources than exclusively using an equal partition of the resources. In the previous example, if user A and user B both use a fair partition exclusively, which is 3 CPUs and 3GB of memory, then both of

[1]Only guaranteed under the assumption of *progressive filling*, i.e., the dominant share of all users increases at the same rate after their tasks are scheduled

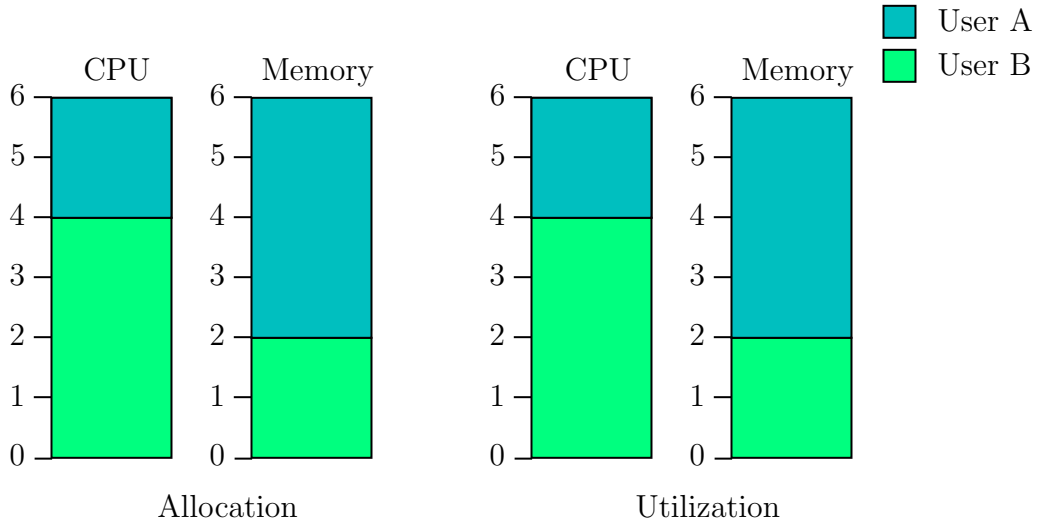


Figure 2.2: Allocation and Utilization of DRF

them can only run 1 task of their application. However, under DRF allocation, user A gets 2 CPUs and 4GB of memory, and user B gets 4 CPUs and 2GB of memory. This allows both of them to run 2 tasks. Therefore, both users would prefer the DRF allocation over the equal partition.

Strategy-proofness. The users cannot run more tasks by misreporting their resource demands. In the previous example, if A lies that it needs more memory, it does not change user A's dominant resource. Therefore, it does not change A's allocation as DRF makes share of dominant resource equal among all users. If A lies that it needs more CPU, DRF may reduce A's memory allocation. For example, if A increases its demand of CPU to 3, then CPU becomes A's dominant resource, and DRF will allocate 3 CPUs and 2GB of memory to A and 3 CPUs and 1.5GB of memory to B. This way, user A's allocation of memory reduces and gets penalized for misreporting.-

Envy-freeness. The users prefer their own allocation to others' allocation. In the previous example, if A and B exchange their DRF allocation, then both of them can only run 1 instance. As a result, both A and B would not prefer the DRF allocation of the other. In the previous example, both users have the same dominant resource. If the applications have the same dominant resource, then their allocations are the same. As a result, they would not prefer the other's allocation either.

Pareto efficiency. The system utilization is maximized so that it is impossible to increase the allocation of a user without decreasing the allocation of another user. In the

Notation	Explanation
i	Process i
w_i	The weight of process I
p_n^i	The n^{th} packet of process i
$R(p)$	The running time on CPU for packet p
c	The number of CPU in the system
$b(p)$	The memory traffic in bytes generated while processing packet p
m	The memory bandwidth of the system

Table 2.1: Notations used in the Dominant Resource Fair Queueing Scheduler

DRF allocation of the example, the system utilization is 100% when A gets 2 CPUs and 4GB of memory while B gets 4 CPUs and 2GB of memory. Obviously, there is no available resource so increasing the allocation of a user always results in a decreased allocation of another user.

2.4.2 Dominant Resource Fair Queueing

For packet scheduling, Ghodsi, et al. [9] propose the dominant resource fair queueing (DRFQ) to achieve DRF properties fro5 time-shared resources. The symbols used in DRFQ is listed in Table 2.1. DRFQ is an extension of start time fair queueing (STFQ). STFQ uses *virtual start time* and *virtual finish time* to schedule packets of different flows. When a packet of a flow arrives, if the flow is not backlogged, then the virtual start time of the packet is the same as the actual time. If the flow is backlogged, then the virtual start time of the packet is the same as the virtual finish time of the previous packet. The difference between the virtual start time and the virtual finish time of a packet is the packet's *processing time*. STFQ picks the packet with the lowest virtual start time to be processed next.

Based on STFQ, three variants of DRFQ are proposed: (i) memory-less DRFQ, (ii) dove-tailing DRFQ, and (iii) Δ-bounded DRFQ. In memory-less DRFQ, the processing time of a packet is defined as the *dominant resource time* of the packet. The dominant resource time of a packet is the largest processing time on any resource. For the n^{th} packet of flow i, p_n^i, its processing time on resource j, $t_{n,j}^i$, is defined as:

$$t_{n,j}^i = \frac{s_{n,j}^i}{r_j} \tag{2.3}$$

In Equation 2.3, r_j is the total capacity of resource j and $s^i_{n,j}$ is the time consumed by p^i_n on resource j. The processing time of p^i_n on resource j is the time consumed by p on resource j normalized by the total capacity of resource j. The dominant resource time of p^i_n, $D(p^i_n)$, is defined as:

$$D(p^i_n) = \frac{max_j\{t^i_{n,j}s\}}{w_i}, j \in \text{resources} \tag{2.4}$$

In Equation 2.4, w_i is the weight of flow i. The calculation of virtual start time and virtual finish time is similar to that of STFQ:

$$S(p^i_n) = max(V(a(p^i_n)), F(p^i_{n-1})), \tag{2.5}$$

and

$$F(p^i_n) = S(p^i_n) + D(p^i_n) \tag{2.6}$$

In Equation 2.5, $a(p^i_n)$ is the actual time when p^i_n arrives. $V(a(p^i_n))$ is the system virtual time at actual time $a(p^i_n)$. $S(p^i_n)$ is the virtual start time of packet p^i_n and $F(p^i_n)$ is its virtual finish time. The virtual start time of p^i_n is the larger one of the system virtual time when it arrives (flow i is not backlogged) and the virtual finish time of the previous packet p^i_{n-1} (flow i is backlogged).

Memory-less DRFQ is fair when the dominant resource of packets of the same flow is the same. However, if the packets of a flow have different dominant resources, it is unfairly penalized. Dove-tailing DRFQ addresses the problem of fairness.

In dove-tailing DRFQ, the system virtual time is maintained for each resource. A packet also has a virtual start time and a virtual finish time on each resource. The virtual start time of a packet on a resource is the larger one of the system virtual time on that resource and the virtual finish time of the previous packet on that resource. The virtual finish time of a packet on a resource is the virtual start time of the packet on that resource plus the processing time on that resource. The virtual start time of a packet is the largest virtual start time on all resources of the packet. When the dominant resource of flow I switches from resource A to resource B, later packets cause less increment on the virtual start time than that of memory-less DRFQ. This is because the packets have higher virtual times on A which is accumulated when the dominant resource of I is A. The increment of virtual times on the new dominant resource, B, does not affect the virtual time of the packet until the gap of virtual times between A and B is filled.

Dove-tailing DRFQ may cause starvation when there is a big gap between virtual start

times on different resources. Δ-bounded DRFQ addresses the problem of starvation. Δ-bounded DRFQ sets the upper bound of the gap between virtual start times on different resources of a packet. The minimum virtual start time on any resource is the maximum virtual start time across all resources minus Δ.

The authors implemented memory-less DRFQ and evaluated it with dominant-resource monotonic workloads. Results show that memory-less DRFQ achieves DRF for packet flows.

Chapter 3

Tracking Per-Process Memory Bandwidth Usage

In our proposed design, a process's memory bandwidth usage in the past affects how it is scheduled in the future. However, memory controllers do not usually track memory bandwidth usage per process. Therefore, we proposed simple modifications to the memory controller to track per-process memory bandwidth usage. We modify `gem5` to implement our proposed modification. In this chapter, we first present a quick overview of the memory subsystem. Then, we provide a detailed description of our modifications to the memory controller. Finally, we describe our implementation in `gem5` and illustrate how we validate our implementation through experiments.

3.1 Characteristics of Memory Requests

`gem5` treats memory requests from CPU and DMA-capable devices differently. In this section, we introduce how memory requests from these sources are generated and treated by the rest of the system.

3.1.1 Memory Requests from CPU

Memory requests originating from CPU are generated by the instruction bus, the data bus, or the Memory Management Unit (MMU). The instruction bus fetches instructions. When memory instructions execute, the data bus generates requests to read from or write

to the memory. If a virtual address is used, the MMU fetches the page table to translate virtual addresses to physical addresses. Compared to the data bus, the total number of memory requests generated by the instruction bus and the MMU is negligible.

Read and write requests have different impacts on the memory subsystem and the performance of the application. Each CPU core in the target system has a private L1 cache and a shared L2 cache. When the CPU generates a memory read request, it is first handled by the L1 cache. If the data is found in L1, there is a cache hit, and L1 responds with the data. If it misses, the L1 cache sends a request to L2 and waits for the response. When the response arrives, L1 tries to find an available cache line. If all cache lines are allocated, then L1 performs a cache line replacement. In write-back caches, if the cache has to replace a dirty cache line, a writeback request is sent to the L2. The L2 cache handles the requests similarly, but sends a request to and waits for responses from the memory controller. Read requests can trigger prefetching and generate additional read requests. Therefore, a read instruction can trigger additional memory requests, i.e., writeback requests and prefetching requests.

When the CPU generates a memory write request, the cache handles it as a read-modify request. This is because the cache operates at the granularity of cache lines (e.g., 64 bytes) and the size of a write request is smaller than the size of a cache line. The request either hits in the cache or the cache line is brought to the cache on a cache miss. Then, a partial write is performed on the cache line. The cache line is marked *dirty* after being written. The cache line is not written to the memory until a replacement happens.

In Linux, one process runs on a CPU core at any time. Therefore, each core can be mapped to a process. The memory requests generated by the core can be attributed to the process.

3.1.2 Memory Requests from DMA-capable Devices

In `gem5`, memory requests from DMA-capable devices first go through the I/O bus and a small I/O cache. The I/O cache in `gem5` is added to handle coherency problems based on snooping requests. The default size of the I/O cache is 1KB, making it negligible compared to the size of DMA requests. The I/O cache handles the requests similarly to the L2 cache. After the requests enter the rest of the cache system, they are treated similarly to requests from CPU. However, attributing a request to a process is different for DMA memory requests.

Different from CPU, processes can share a DMA-capable device at the same time. Therefore, a request generated by a DMA-capable device cannot be simply attributed to

a process by identifying the device that generated the request. A possible solution for this problem is to store the process ID (PID) in memory requests.

Existing hardware with IOMMU [1] supports storing PID in memory requests. IOMMU [14] is introduced to support translation from virtual address to physical address for I/O devices. Before IOMMU, I/O devices use contiguous physical addresses to access memory. This would give them full access to all memory, creating security problems. Allocating contiguous physical addresses for large DMA is also challenging. IOMMU translates virtual addresses based on page tables provided by the kernel. The IOMMU locates the page tables to be used for a request by the Requester ID (RID) identifying the device and the Process Address Space ID (PASID) in PCI-e packets identifying the user process. The fields `streamID` and `substreamID` in ARM SMMU packets are identical with RID and PASID. IOMMU provides performance monitoring counters that count the number of memory requests sorted by PASID, which can be used to count I/O memory traffic generated by each user.

Ideally, if IOMMU is enabled and PASID is supported by the operating system and the DMA-capable device, when a process starts an operation on the device, the device driver provides the process ID and the virtual address of the data buffer to the hardware. The data buffer is allocated in user space memory. When the device needs to use DMA to read from or write to the data buffer to service the request, it uses the process ID provided by the driver as PASID. The kernel configures the IOMMU to translate the virtual address using the process's page table. As a result, the device performs DMA read and write on the user space memory of the process. If the device does not support PASID, the kernel can use I/O Virtual Address (IOVA). IOVA creates an address space for one or more devices, separating data buffers from the rest of the memory. If the kernel does not support IOMMU, IOMMU operates in pass-through mode. All requests in the system use physical address and IOMMU does not perform any translation on the addresses.

Many existing DMA-capable devices, including GPU, NIC, and SSD, do not support PASID. A key reason for this is that the protocols used by these types of devices do not have a definition of PASID. For example, NVMe command used by NVMe protocol does not have a field for PASID in its data structure.

The operating system also needs to determine which process is responsible for DMA memory traffic. However, this can be hard to implement. For example, considering disk I/O, the virtual file system (VFS) and the block I/O layer in Linux obfuscate the process responsible for DMA operations. In VFS, non-direct write operations write to the page cache. After that, the data is written back to the disk when the utilization of the page cache

[1] Our target system also has SMMU, which is the name of IOMMU in ARM systems.

Name	Number of Registers	R/W	Description
traffic[max_pid]	8192	RO	The traffic attributed to each process
virtualTime_pid[max_pid]	8192	RW	The virtual time of each PID set by the kernel
virtualTime_coreId[n_cpus]	4	RW	The virtual time of the process running on each CPU core
pid_coreId[n_cpus]	4	RW	The PID of process running on each CPU core

Table 3.1: Registers in the Memory Controller in `gem5`

reaches a threshold or after a time threshold. Such mechanisms make it hard to determine the original process responsible for the write command sent to the disk. However, non-direct I/O is usually used for small amounts of data transfer. If the size of the data to be written exceeds the size of the page cache, the overhead of copying data to and from the page cache degrades the performance. To address this problem, for large disk I/O, a common practice followed by I/O-intensive applications (e.g., MySQL InnoDB engine) is to use direct asynchronous I/O provided by the `libaio` library in Linux to bypass the page cache. Such applications manage I/O cache by themselves, use direct I/O to avoid the overheads of page cache, and use asynchronous I/O to reduce CPU usage.

3.2 Monitoring Memory Bandwidth

To track per-process memory usage, we propose modifications to the memory controller. In particular, we propose adding an interface which is connected to the peripheral bus. The interface allows the kernel running on CPU to access a group of memory-mapped registers used for memory bandwidth monitoring and memory packet scheduling. The definition of the registers is shown in Table 3.1.

`traffic` and `pid_coreId` registers are used to track memory traffic of processes. `traffic` registers count the number of bytes read and written by each process. They are read-only and clear to 0 after each read by the kernel running on CPU. When the kernel assigns a

PID to a newly created process, it reads the traffic register associated to the PID to clear the value of the register to 0. After that, the kernel read the traffic registers to get the amount of traffic generated by each process since the last read. `pid_coreid` registers store the mapping from CPU core ID to PID. It is used to find the PID of memory requests generated by CPU. We set the maximum number of PIDs in Linux to 8192 (32768 by default). In our experiments, the maximum PID never exceeds 1024.

Implementing our proposed design in real hardware raises some practical challenges that need to be addressed. First, the cost is correlated to the number of each set of registers, which is the maximum number of processes or the number of CPU cores. While the number of CPU cores is usually limited, the number of processes can be a large value. A possible solution is to distinguish processes that need memory bandwidth tracking and memory request scheduling. The operating system can assign them another unique ID with a limited maximum value to be used in the memory controller. Second, accessing the registers introduces extra delay depending on the hardware implementation. The delay can be reduced by using a faster hardware design or limiting the times of access to these registers in the kernel.

3.3 `gem5` Simulator

To implement our proposed modifications to the memory controller, we use the `gem5` simulator. The `gem5` simulator is a cycle-accurate full-system simulator widely used in computer architecture studies [8][16]. The `gem5` simulator supports multiple simulation models for CPUs, cache systems, system buses, and memory. Custom implementations of simulated network cards, GPUs, and SSDs can be integrated into `gem5` simulation.

In `gem5`, all simulation objects inherit the `SimObject`. Some types of simulation objects including cache, buses, and CPUs share other superclasses which provide the basic function of the object. Simulation objects usually have configurable parameters. Instances of simulation objects are created and configured by the starter script that starts running the simulation. In this section, we introduce important components in `gem5`.

3.3.1 Simulation Mode

`gem5` supports different simulation modes, providing different levels of speed and accuracy for different use cases. Implementation of simulated hardware components may support one or more modes. The fastest *system emulation* (SE) mode only provides the

emulation of most system-level services including system calls. The *atomic* mode simulates a full system without memory and system bus delays, which runs slower. The *timing* mode is based on the atomic mode but simulates all memory and system bus delays, which runs slowest. While some detailed hardware behavior is not fully consistent with real hardware, the timing mode is considered accurate enough for system research [4]. In this thesis, we use `gem5` to simulate memory subsystem with contention and delays. Thus, we only use timing mode in later discussions.

3.3.2 CPUs

`gem5` provides different CPU models. In atomic mode, a simple instruction-per-cycle (IPC) CPU model is used. In timing mode, the simplest CPU model is the `TimingSimpleCPU`, which is based on the IPC CPU used in atomic mode but also simulates the delays caused by memory instructions. A more complex CPU model is the in-order, pipelined CPU model. The most sophisticated CPU model is the out-of-order CPU model based on the Alpha 21264 RISC CPU that is used in real world. Complex CPU models provide better in-simulation performance but lower simulation speed. CPU models in `gem5` are also Instruction Set Architecture (ISA) independent, as the ISA layer is separated from the CPU models. Thus, multiple ISAs are implemented using the CPU models, and developers can modify the ISA design. For simplicity and simulation performance ,we use ARMv8 `TimingSimpleCPU` in our experiments.

3.3.3 Ports and Buses

`gem5` models the bus requests and responses within the components such as CPU, cache system, external devices, and memory controller. Simulation objects have one or more ports to connect with other objects. For example, the CPU object has an instruction bus and a data bus, both have a memory-side port connected to the L1 I-cache or D-cache. The memory controller has a CPU-side port connected to the memory bus interconnect. Simulation objects between the source (e.g., CPU) and the target (e.g., memory controller) have both a CPU-side port and a memory-side port.

In timing mode, the memory-side port provides an interface with the `sendTimingReq` function and the `recvTimingResp` function, and the CPU-side port provides an interface with the `recvTimingReq` function and the `sendTimingResp` function. The `sendTimingReq` function is called to send a request. It then invokes the `recvTimingReq` function of the corresponding CPU-side port. The `sendTimingResp` function is called to send a response

for a request. It then invokes the `recvTimingResp` function of the simulation object which sent the request. Requests are modeled as packets in the `Packet` class.

3.4 Monitoring CPU Memory Usage

In `gem5`, the memory requests are modeled in the `Packet` class. A packet stores the information of a memory request. To identify the originating core of a packet, we added `coreId` field in the class, which has a default value of -1, meaning it is not initialized and invalid. When a packet is created by the CPU data bus, the core ID is written to the `coreId` field. To attribute a packet generated by a CPU core to a process, the memory controller uses the `coreId` and the mapping stored in the `pid_coreid` registers. When the kernel switches to a new non-idle process, it writes the PID of the new process to the `pid_coreid` register corresponding to the CPU core it is performing the context switching.

`coreId` is propagated when a packet triggers the creation of prefetching and writeback packets. Prefetching packets are created in the `QueuedPrefetcher::insert` function. In `QueuedPrefetcher::insert`, prefetching packets are created and inserted into the outgoing packet queue of the cache. `coreId` from the original packet is propagated to the newly creatffffed packet, attributing prefetching packets to the process that created the original packet.

Writeback packets are created in the `BaseCache:recvTimingResp` function, which is invoked when the response of a packet arrives at a cache. If the response is for a read packet that requires a cache block to be allocated, a writeback is triggered. `BaseCache:recvTimingResp` calls other functions to find the block and evict it if necessary. It generates a list of writeback packets. We modify `BaseCache:recvTimingResp` to propagate `coreId` from the original packet of the response to the generated writeback packets. Although the data being written back may not belong to the process that triggers the writeback, we still attribute the writeback to the triggering process that generated the original packet. This is because the triggered writebacks are the results of the execution of the triggering process. Attributing them to the triggering process makes the counting directly correlate with the impact on system resources by the process. Writeback can also be triggered by a writeback packet generated by the upper-level cache to the lower-level cache if the lower-level cache needs to evict a block. In this case, `coreId` is propagated from the original writeback packet to the generated writeback packets.

When a packet arrives at the memory controller, the `DRAMCtrl::recvTimingReq` function is invoked. `DRAMCtrl::recvTimingReq` adds packets to DRAM queues if the queues

are not full. We modify `DRAMCtrl::recvTimingReq` to update the `traffic` registers which store the per-process memory traffic tracking. If `coreId` of the incoming packet is valid, the memory controller reads the PID from the corresponding `pid_coreid` register. Then, it adds the size of the Packet to the `traffic` register corresponding to the PID.

3.5 Monitoring DMA Memory Usage

To monitor per-process memory traffic generated by DMA-capable devices, we utilize the existing implementation which supports SMMU. The `gem5` Packet class already provides the `subStreamID` field that can be used to store PID. We add support for `subStreamID` in the DMA-capable device, the driver, and the memory controller.

We first add the definition of PID in NVMe commands. In the NVMe protocol, operations including read and write are controlled by NVMe commands. An NVMe command consists of multiple little-endian 64-bit Command Dwords. In the current definition of NVMe commands, several Command Dwords are unused. We pick Command Dword 1, which is unused in both read command and write command, to store the PID value. Both Linux and SimpleSSD implement the data structure of NVMe command. In Linux kernel, NVMe command is implemented in `struct nvme_command_rw`. Command Dword 1 is defined as the `u64 rsvd2` field. In SimpleSSD, NVMe command is implemented in `SimpleSSD::HIL::NVMe::SQEntry`. Command Dword 1 is defined as two `uint_32t` fields `reserved1` and `reserved2`.

The NVMe driver needs to determine which PID it should write to NVMe commands. In the NVMe driver in Linux, the `nvme_setup_rw` function sets up NVMe commands. For I/O operations initiated using the `libaio` library, the `nvme_setup_rw` is called during the system call invoked by the initiating process. Thus, the PID can be obtained from the system context. In the Linux kernel, `current` global variable stores the pointer to the descriptor of the process currently running on the current CPU core. The driver writes the PID from the `current` global variable to the `u64 rsvd2` field in `struct nvme_command_rw`, so SimpleSSD can get the PID from fields `reserved1` and `reserved2` in `SimpleSSD::HIL::NVMe::SQEntry`.

After receiving a command, SimpleSSD creates an `SGL` or `PRP` object to store the command. NVMe commands use Physical Region Page (PRP) or Scatter/Gather List (SGL) for addressing memory. The type of addressing of an NVMe command is specified in the NVMe command. NVMe driver in Linux uses PRP. To support tracking the PID of NVMe commands, we add a `subStreamID` field in both `SGL` class and `PRP`

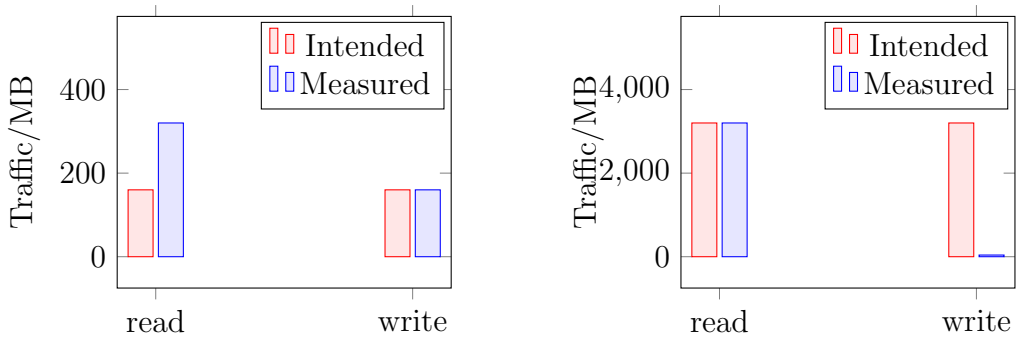
class. `subStreamID` is initialized with the PID in the NVMe command (`reserved2` in `SimpleSSD::HIL::NVMe::SQEntry`).

When the execution of the command needs to access the memory, the `dmaRead` or `dmaWrite` function of the `NVMeInterface` class is called to create DMA entries (`DMAEntry` class). We add the `subStreamID` fields in `DMAEntry`, which is initialized with the value from `subStreamID` in `SGL` or `PRP`. DMA entries are stored in the DMA queue, waiting to be processed. When DMA entries are processed, packets are created and sent to the I/O cache. This is done in `NVMeInterface::submitDMARead` and `NVMeInterface::submitDMAWrite`. We modify `NVMeInterface::submitDMARead` and `NVMeInterface::submitDMAWrite` to use `subStreamID` from `DMAEntry` when calling functions `DmaDevice::dmaRead` and `Dmadevice::dmaWrite` to create DMA packets. After entering the I/O cache, the packets can trigger writeback and prefetch. In these cases, the `subStreamID` of the original packet is propagated to the generated packets similarly to `coreId` as mentioned in previous section. Finally, when a DMA packet arrives at the memory controller, the memory controller gets the PID from `subStreamID` and adds the `traffic` register corresponding to the PID with the size of the packet.

3.6 Verification

Amount of Traffic. We first verify that the total amount of traffic counted by the memory controller matches the traffic generated by processes. We run `mbw` benchmark that does array copy, which reads from one address and then writes to another address. With `mbw`, the generated memory traffic is 50% higher than the memory traffic intended by the code. This is mainly because the cache subsystem treats write requests as read-modify-writeback. Thus, writing to the memory generates additional read requests equal to the size written. We also run `fio` to read from and write to the SSD using asynchronous direct I/O by `libaio`. While reading from the SSD generates memory traffic slightly more (less than 1%) than the intended size of read, writing to the SSD generates much lower memory traffic when the block size is small. This is because DMA requests may hit in the L2 cache. `fio` writes the same buffer in the memory to the SSD repeatedly, so the buffer is cached in L2 and not replaced if it is small enough.

Memory Packets from All Sources. We also validate that our implementation can capture requests created by different hardware components. We run `lbm`, `Gems`, and `mcf` from SPEC2006 benchmark to measure the distribution of source of requests. Fig. 3.2 depicts the source of requests. Requests are categorized into accounted requests generated by the cache writeback, the L1 prefetcher, the L2 prefetcher, and the CPU data bus, and



(a) `mbw` (b) `fio`

Figure 3.1: Intended and Measured Memory Traffic of `mbw` and `fio`

unaccounted requests. All accounted write requests are categorized as cache writeback requests. Percentages of write requests for `lbm`, `Gems`, and `mcf` is 43.9%, 37.2%, 50.3%. Accounted read requests are categorized as requests generated by the L1 prefetcher, the L2 prefetcher, and the CPU data bus. Percentages of requests generated by the L1 prefetcher for `lbm`, `Gems`, and `mcf` is 32.9%, 32.0%, 27%. Percentages of requests generated by the L2 prefetcher for `lbm`, `Gems`, and `mcf` is 23.2%, 30.8%, 22.7%.

Accounted Read requests generated by the data bus for all three benchmarks are less than 1%. This indicates that most read requests generated by CPU data bus are hit in the cache because of prefetching. The results also show that requests that are not accounted for any process generated when the benchmarks run is less than 1% of total requests. In summary, the results show that our implementation correctly reports the memory traffic generated by processes.

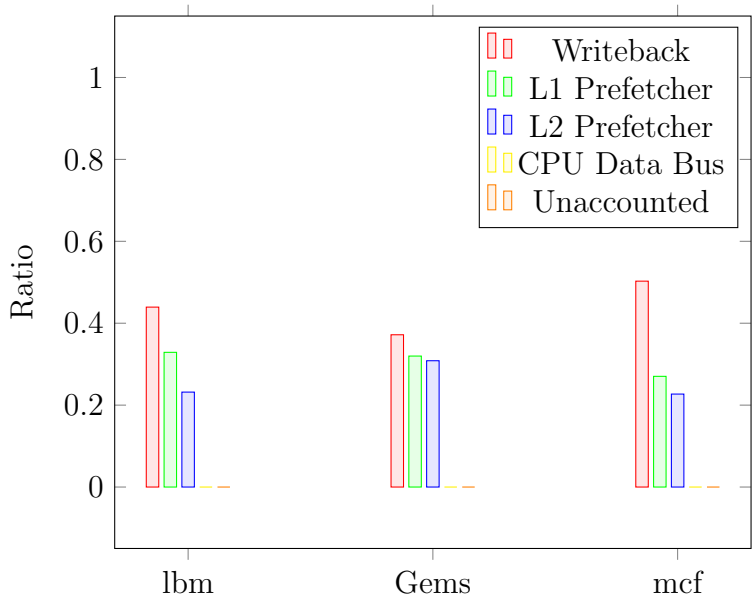


Figure 3.2: Distribution of Source of Memory Requests

Chapter 4

Dominant Resource Fair Queueing Scheduler

We propose Dominant Resource Fair Queueing Scheduler (DRFQS) to fairly allocate CPU time and memory bandwidth. DRFQS allocates CPU time and memory bandwidth fairly according to our proposed model based on DRFQ (Section 2.4.2. CPU time allocation is enforced by the kernel scheduler. Memory bandwidth allocation is enforced by the kernel scheduler (as source throttler) and the memory request scheduler (as target arbitrator).

In this chapter, we first illustrate our model of processing scheduling in operating systems. Then, we overview our Linux scheduler design. Next, we describe our proposed modifications to the memory controller. Finally, we discuss challenges that arise because of the differences between the DRFQ model and process scheduling in the OS. We end this section by proposing solutions to these challenges to achieve multi-resource fairness in process scheduling.

4.1 Process Scheduling Model in Operating System

We use memory-less DRFQ (see Section 2.4.2) to model the process scheduling in Linux. The notation used is listed in Table 4.1. The design of the run queue in CFS mostly follows the STFQ. DRFQ reduces to STFQ when there is only one type of resource. We keep the properties of CFS so our model reduces to CFS when the processes only have contention on CPU time.

Notation	Explanation
i	Process i
w_i	The weight of process I
p_n^i	The n^{th} packet of process i
$R(p)$	The running time on CPU for packet p
c	The number of CPU in the system
$b(p)$	The memory traffic in bytes generated while processing packet p
m	The memory bandwidth of the system
$V_{current}$	Current system virtual time

Table 4.1: Notations used in the Dominant Resource Fair Queueing Scheduler

In DRFQS, processes are considered similar to packet flows in DRFQ. DRFQS divides the execution of a process into "packets." A packet describes the period in which a process is running on CPU or using memory bandwidth through DMA-capable devices. A new packet is created when the status of a process changes (e.g., when a process is preempted, blocked, unblocked, or runs out of its current time slice). The previous packet finishes when the new packet is created. When a process is ready but not running on CPU, it is considered to have a backlogged packet and no packet currently being processed. When a process is running on CPU or is blocked, it is considered to have a packet currently being processed and no packet backlogged.

In DRFQ, the processing time of a packet should be calculated based on the time it consumes on different resources. However, this cannot be determined before it runs. Therefore, in DRFQS, the processing time of a packet is calculated based on its resource consumption measured after it finishes. DRFQS relies on only the virtual start time to schedule packets from processes. The processing time of a packet only affects the virtual finish time of the current packet and the virtual start time of future packets. We define the processing time on CPU to represent the consumption of CPU time and processing time on memory controller to represent the consumption of memory bandwidth.

Consider a system with c CPU cores and m GB/s memory bandwidth. For a packet that describes the period when the process is running on CPU, the processing time on CPU is calculated as:

$$T(p_n^i, CPU) = \frac{1}{c} \times R(p_n^i), \tag{4.1}$$

where p_n^i is the n^{th} packet of process i and $R(p_n^i)$ is p_n^i's running time on CPU. A process uses 1 of c CPUs, so the share of CPU consumed during $R(p_n^i)$ is $\frac{1}{c}$.

The processing time on memory bus is calculated as:

$$T(p_n^i, mem) = \frac{b(p_n^i)}{m}, \tag{4.2}$$

where $b(p_n^i)$ is the amount of memory traffic in gigabytes generated by p_n^i after the packet is created. This includes the memory traffic generated by CPU instructions or DMA-capable devices, whether the process runs on CPU or not. Therefore, the dominant resource time of the packet p_n^i is calculated as the larger one of the processing time on CPU and the processing time on memory bus, divided by the weight of the process:

$$D(p_n^i) = \frac{max(T(p_n^i, CPU), T(p_n^i, mem))}{w_i}. \tag{4.3}$$

The virtual finish time of p_n^i is calculated as the virtual start time plus the dominant resource time:

$$F(p_n^i) = S(p_n^i) + D(p_n^i). \tag{4.4}$$

In CFS, recently unblocked processes are not charged for the memory bandwidth they consume when they are blocked. However, when a process is blocked, it still can consume memory bandwidth. For example, with asynchronous disk I/Os, a process can start a disk I/O operation before it is blocked. When the hardware finishes the operation, it triggers a system interrupt to unblock the process. The period that the process is blocked is treated as a packet in DRFQS, denoted as p_k^i. p_k^i does not consume CPU. The dominant resource time of p_k^i is calculated as:

$$D(p_k^i) = \frac{b(p_k^i)}{m \times w_i}. \tag{4.5}$$

A blocked process may consume little to no memory bandwidth (e.g., a process waiting for events other than the finishing of DMA operation). In CFS, a system virtual time is maintained for each queue and used to assign the virtual time of recently unblocked processes. In DRFQS, we maintain and use a system virtual time system similar to CFS's:

$$V(t) = max(V(t'), min_p(S(p))),\ \forall p \in \text{packets currently processing}, \tag{4.6}$$

where $V(t)$ is the system virtual time at actual time t, and t' is the actual time of the last time the system virtual time was updated. The system virtual time is updated to the minimum virtual start time of all currently processing packets if it does not decrease.

If a process consumes little to no memory bandwidth when it is blocked, it is considered not to be backlogged. We apply this to the calculation of the virtual start time of p^i_{k+1}:

$$S(p^i_{k+1}) = max(V(a(p^i_{k+1})) - C, F(p_k)), \tag{4.7}$$

where $V(a(p^i_{k+1}))$ is the system virtual time when p^i_{k+1} arrives (when the process is unblocked), and C is a positive constant. By subtracting C from the system virtual time before assigning it to a non-backlogged packet, DRFQS gives recently unblocked processes a higher priority than other processes similarly to CFS.

Lastly, newly created processes are considered to have a non-backlogged packet. The virtual start time of the packet is assigned to the system virtual time when the process is created:

$$S(p^i_0) = V(a(p^i_0)). \tag{4.8}$$

4.2 Linux Scheduler Design

The process scheduler of DRFQS is based on the BFS scheduler of Linux [30]. In this section, we first explain the reason we choose BFS over CFS. Then, we discuss our modifications to the data structures of the original BFS. Next, we explain how the virtual time is updated and used in the kernel scheduler. Finally, we discuss how the kernel controls the memory scheduler with the input of virtual time.

4.3 The Linux Kernel and the BFS Scheduler

CFS scheduler uses a separate run queue for each CPU core. The virtual time of processes in different run queues is not comparable. This is because each run queue maintains its own minimum virtual time (`min_vruntime`). `min_vruntime` increases when the virtual time of the process with minimum virtual time in the queue increases. CFS does not balance the value of `min_vruntime` of multiple run queues. If `min_vruntime` of different queues increase at different paces, e.g., when the number of processes in the queues are different so the CPU time allocated to each process is different, the value of `min_vruntime` of the

queues becomes significantly different after running for some time. `min_vruntime` is used to initialize new processes, subtracted from processes migrated out from the queue, and added to processes migrated into the queue. As a result, the imbalance of `min_vruntime` makes the virtual time of the processes in different queues incomparable. CFS balances loads of different queues only if their load value is significantly different. This makes the unfairness between queues persists for a long period. Incomparable virtual time makes it hard to determine the priority of memory packets in the memory controller. Different from CFS, BFS uses only a single global run queue. We adopt the single-queue design of BFS and implement our scheduler based on BFS code.

In BFS, each process has a virtual deadline. The scheduler updates this value and uses it to pick the process to run. The deadline is updated when the following functions are called:

1. `time_slice_expired()` resets the timeslice and set the new deadline for a process that just used up its time slice when a scheduler timer interrupt is triggered. The deadline is set to the current system time plus the priority offset. The priority offset value is calculated based on the process's priority so that high priority process has a lower deadline.

2. `adjust_deadline()` adjusts the deadline of a process when its priority changes while it is running. The deadline is subtracted by the priority offset based on its previous priority, then added with the priority offset based on the new priority.

3. `wake_up_new_task()` finishes the initialization of a new process, then adds it to the run queue and makes it ready to run. The deadline of the newly created process is set to the deadline of the process that created it.

In our target system, NUMA is not enabled. BFS initializes the distance between each pair of CPU cores to be 3. If a process was previously assigned to another CPU, its deadline is multiplied by 8 before comparing it with other processes. This ensures CPU affinity, reducing times of migration and migration overheads.

4.3.1 Data Structure Modifications

BFS uses the `task_struct` data structure for process descriptors and the `global_rq` data structure for global run queue. Each process has a `task_struct`. Pointers of `task_struct` are stored in lists. A global `global_rq` variable, `grq`, is used for the global run queue. The kernel acquires a lock before accessing or modifying `grq`. We add necessary fields to `task_struct` and `global_rq` as shown in Table 4.2 and Table 4.3, respectively.

Name	Type	Description
vruntime	uint64	The virtual time of the process
last_update	uint64	The actual time indicating the last time the virtual time of the process was updated
nice	int32	The nice value of the process used to calculate the weight
is_unblocked	bool	The flag indicating if the process has benn unblocked

Table 4.2: Added Fields in the `task_struct` Data Structure

Name	Type	Description
global_vruntime	uint64	The system virtual time $V_{current}$

Table 4.3: Added Fields in the `global_rq` Data Structure

In the process descriptor `task_struct`, we replace the `deadline` field used by BFS with `vruntime`. `vruntime` stores the virtual time of the process as in CFS. `last_update` stores the actual time (scheduler clock time) when the last time `vruntime` is updated. `nice` is added to store the weight of the process. `is_unblocked` is added to mark if a process is a recently unblocked process. It is set when the process is unblocked and cleared when the process starts to run on CPU. It is used to delay the wake-up of recently unblocked processes in some circumstances to achieve fairness (see Section 4.5.1 for details). In the global variable `grq`, the `global_vruntime` field is added to store the system virtual time $V_{current}$.

Addition of Virtual Time to BFS

Original BFS uses virtual deadline to schedule processes. We replace the deadline with virtual time. The virtual time of a process is a monotonically increasing value that is updated on three events:

1. when the scheduler timer interrupt is triggered, the interrupted process runs out its time slice, and
2. when a blocked process is unblocked, and
3. when a new process is created.

Before updating the virtual time, the kernel calculates the processing time on CPU and the processing time on memory bus. The processing time on CPU is calculated with the actual time passed since the time recorded by `last_update` in `task_ struct`. `last_update` is updated when the CPU switches to process i or when `update_vruntime(i)` was called on the process. The current time is read from the `grq.niffies` field in the global run queue. The actual time passed is calculated by subtracting `last_update` from `grq.niffies`. The processing time on CPU is the actual time passed divided by the number of CPU cores, according to Equation 4.1.

The processing time on memory bus is calculated according to Equation 4.2. It is calculated by dividing the traffic generated since the last time `update_vruntime(i)` was called on process i by the maximum memory bandwidth. The traffic is read from the `traffic` register corresponding to the PID of the process (as shown in Table 3.1). For a system with single-channel 1066MHz 32-bit LPDDR2 memory, the theoretical speed is 4.264 bytes/ns. The theoretical speed is almost impossible to reach. This is because the overheads of row buffer miss and eviction and read/write direction turning. Our experiment shows that the worst-case maximum memory bandwidth of such a DRAM module can be as low as 3.5 bytes/ns. For such a system, 4 bytes/ns can be used as an approximate value of maximum memory bandwidth for simplicity. The choice of such value affects the calculated share of memory bandwidth. Therefore, it affects the calculation of dominant resource time. However, it has an effect on allocation only if processes in the system have different types of dominant resources and the share of CPU time and the share of memory bandwidth consumed by some processes are close. The maximum memory bandwidth used in the scheduler can be set higher to benefit processes with memory bandwidth as their dominant resource, or set lower to benefit processes with CPU time as their dominant resource.

We also track the system virtual time, $V_{current}$, according to Equation 4.6. It is used to assign the virtual time of recently unblocked processes (according to Equation 4.7) and newly created processes (according to Equation). We designate a function, `update_global_vruntime()` to update the system virtual time, as shown in Algorithm 1. The function first finds the lowest virtual time of all processes currently running on CPUs. If the value is larger than the previous $V_{current}$, then `global_runtime` is updated to the new lowest virtual time value.

When a new process is created, `update_global_vruntime()` is called first to update $V_{current}$, and then the virtual time of the newly created process is set to $V_{current}$.

When a process is running and the scheduler timer interrupt is atriggered, the `schedule()` function is called. `schedule()` is the main scheduler function. If a pro-

Algorithm 1 Updating System Virtual Time

$T \leftarrow$ minimum virtual time of running proceses
if $T >$ system virtual time **then**
  system virtual time $\leftarrow T$
end if

cess, denoted as i, runs on the interrupted CPU core before the interrupt is triggered, `schedule()` updates the status of i and picks the next process to run on the interrupted CPU core. We designate a `update_vruntime(`i`)` to update the virtual time of i based on Equation 4.4, as shown in Algorithm 2.

`update_vruntime` also updates the virtual time of processes running on other cores. This prevents the inconsistency of virtual time. If other running processes are not updated, then their virtual time is likely to be lower than the newly updated process. Since virtual time is also used by the memory scheduler, this avoids unfairness in the memory scheduler. While blocked processes may consume memory bandwidth, their virtual time cannot be updated in the same way. This is because the number of blocked processes is large and most of them do not consume memory bandwidth. Updating virtual time for all of them creates significant overheads. If the virtual time of blocked processes is not updated, memory requests generated by DMA-capable devices for these processes would have a temporal advantage over other requests. However, the virtual time is updated fairly when the blocked process is unblocked.

Simply adding the virtual time with the dominant resource time can cause starvation. Consider the situation when there are two processes and more than two CPU cores. As there is no contention on CPU, one of the processes, denoted as A, always runs on the CPU and generates a large amount of memory traffic, making memory bandwidth its dominant resource. The other, denoted as B, also always runs on the CPU and generates little memory traffic, making CPU time its dominant resource. The maximum share of CPU allocated to a process is $\frac{1}{c}$, where c is the number of CPU cores. The share of memory bandwidth can be higher than that. As a result, A's virtual time grows faster than B's. Over time, A's virtual time becomes much larger than B's. The system virtual time tracks the lowest virtual time across all running processes, which is B's. If new processes are created in this situation, their virtual time is much lower than A's. When the number of processes is more than the number of CPU cores, A is starved until other processes catch up with it on virtual time. To prevent such starvation, DRFQS limits the difference between the virtual time of a running process and the system virtual time. If the limit is too small, the increment of virtual time becomes unfair if a process consumes a large

share of resources between two updates. If the limit is too large, the starvation can be too long when it happens. We use the length of 10 time slices as the limit. Therefore, in `update_vruntime`, after updating the virtual time of all processes, `schedule()` calls `update_global_vruntime()` to update $V_{current}$. Then, it compares the virtual time of i with the $V_{current}$ and applies the limit.

Algorithm 2 Updating Virtual Time of Running Process i

for each running process j (includes i) **do**
  $T \leftarrow$ current time $-$ `last_update`$_j$
  `last_update`$_j \leftarrow$ current time
  $b \leftarrow$ `traffic`[`pid`$_j$]
  `vruntime`$_j$ $+= max(\frac{T}{c}, \frac{b}{m})/w_j$
end for
`update_global_vruntime()`
`vruntime`$_i \leftarrow min(V_{current} + 10 \times timeslice,$ `vruntime`$_i)$

When a blocked process is unblocked, the `activate_task()` function is called. The function first updates the virtual time of all running processes and $V_{current}$ similarly to `schedule()` and calls `update_global_vruntime()` to update the global virtual time. The function calculates the virtual time of the unblocked process as Equation 4.7, as shown in Algorithm 3. The constant C in Equation 4.7 is set to the length of 10 time slices. If a process consumes little to no memory bandwidth so its virtual time is lower than $V_{current} - C$ after adding its dominant resource time, it is considered "sleeping" when it is blocked. In this case, its virtual time is set to $V_{current} - C$, which is similar to CFS. If not, it either "sleeps" for a short time or is actually "running" when it is blocked. Different from running processes, the increment of virtual time of recently unblocked processes has no upper bound. This is because a process can be blocked for a long time and consume a large share of memory bandwidth during the period. A limit on the increment of virtual time can cause an unfair advantage for recently unblocked processes. The scheduler marks the process as a recently unblocked process by setting its `is_unblocked` field to true, in order to support delayed wake-up in Section 4.5.1.

We replace the definition of *nice* value. We modified the `setpriority()` system call. The system call takes an integer as a parameter and writes the *nice* value of the process, stored in the `nice` field of the calling process's descriptor, with the input integer. When updating `vruntime`, the *nice* value is converted to the weight as shown in Equation 4.9.

Algorithm 3 Updating Virtual Time of Recently Unblocked Process i

`update_vruntime()`
$b \leftarrow$ `traffic`$[pid_i]$
`vruntime`$_i$ $+= \frac{b}{m}/w_i$
`vruntime`$_i \leftarrow max(V_{current} + 10 \times timeslice,$ `vruntime`$_i)$

$$w_i = \begin{cases} 1 + nice_i, & \text{when } nice_i >= 0, \\ \frac{1}{1-nice_i}, & \text{when } nice_i < 0. \end{cases} \tag{4.9}$$

Controlling the Memory Scheduler

The kernel writes the `virtualTime_pid`, `virtualTime_coreId`, `pid_coreId` registers (in Table 3.1) to provide input to the memory scheduler. When switching the process on a CPU core, the kernel writes the PID of the process that starts to run on the CPU core to the corresponding `pid_coreId` register. This provides the mapping from CPU core to PID, enabling the memory tracking module in the memory controller to identify the process responsible for memory requests generated by CPU cores and caches. When the virtual time of a process is updated, the kernel writes the `vruntime` field of the process descriptor to the corresponding `virtualTime_pid` register. This updates the priority of the requests generated by the process. The `virtualTime_pid` register is only used to provide the priority by DMA memory requests. For memory requests generated by CPU cores and caches, the `virtualTime_coreId` registers are used to provide the priority of these requests. On both occasions (when switching the process and when updating the virtual time), the corresponding `virtualTime_coreId` register is updated.

4.4 Design of Memory Scheduler

We use a first-ready, lowest-virtual-time-first policy in the memory scheduler of the memory controller. With such a policy, the memory scheduler first finds the request that hits the row buffer. If such a request does not exist in the queue, it picks the request with the lowest virtual time. If a request can be attributed to a process, its virtual time is set to the virtual time of the process responsible for it `coreId` and `subStreamId` fields of the memory request packet and the registers shown in Table 3.1. For packets generated by the data bus of CPU cores and additional packets triggered by these packets, the memory

controller gets the virtual time from the `virtualTime_coreId` register corresponding to the core ID provided by the `coreId` field of the packet. For packets generated by DMA-capable devices and additional packets triggered by these packets, the memory controller gets the virtual time from the `virtualTime_pid` register corresponding to the PID provided by the subStreamId field of the packet.

For packets generated by the instruction bus of CPU cores and packets generated by DMA-capable devices for a DMA operation started by routines other than direct disk I/O, the `coreId` field and the `subStreamId` field are not valid. The virtual time of these memory requests is assigned to 0, giving them the highest priority. However, if these requests need to be counted, requests generated by the instruction bus can be attributed similarly to the requests generated by the data bus. Requests generated by non-direct disk I/O can be attributed to the process that triggers the I/O with modifications to the VFS layer of Linux. Requests generated by the kernel should avoid causing memory contention, or they should be given the highest priority when necessary.

After assigning virtual time to a packet, the memory scheduler then determines whether to accept the packet and insert it into the queue. The memory controller has a read queue for read requests and a write queue for write requests. Accepted packets are inserted into the corresponding queue, sorted first by virtual time and then by actual arrival time, i.e., the hardware time when the packet arrives. When the arrival rate of packets is high, excessive packets are rejected. Our proposed design rejects a packet on two occasions: (i) when the corresponding queue is full, and (ii) when the utilization of the corresponding queue is higher than 90% and the virtual time of the packet is not lower than the lowest virtual time across all packets in the queue (the head of the queue). This way, 10% of the queue is preserved to packets with lower virtual time (higher priority), allowing them to be prioritized when the packet rate is high.

When the DRAM bus becomes ready, the memory scheduler picks a packet from the queues to be processed. The memory scheduler first determines which queue to pick from, i.e., the next direction (read or write) of the DRAM bus. Our proposed design chooses the queue whose head has lower virtual time. This may result in frequent switching of the direction of the DRAM bus. Switching the direction of DRAM bus introduces extra delay. The delay when switching from read to write and from write to read is provided in the tRTW and tWTR parameter of the DRAM, respectively. Therefore, frequent switching causes degradation of memory performance. This is an intrinsic trade-off between performance and fairness. The performance degradation is negligible in the system and for two main reasons. First, bursty requests are mostly generated by the prefetcher or DMA-capable devices. Requests of the same burst have the same virtual time and direction and do not cause frequent switches of direction. Second, the virtual time used in the memory

scheduler does not change frequently. It only changes when the kernel scheduler updates the registers, which has an interval long enough compared to the switching time.

After the queue is determined, with the first-ready, lowest-virtual-time-first policy, the memory scheduler iterates through the queue, starting from the head, to find a row hit. If it exists, the first row-hit packet is picked. Otherwise, the scheduler picks the head of the queue which has the lowest virtual time.

4.5 Supplemental Implementation of DRFQS

In the previous section, we followed the model provided by DRFQ to build DRFQS. However, there are some main differences between the packet scheduling problem in DRFQ and the process scheduling in operating systems, as well as between memory bandwidth and other resources. In this section, we discuss the challenges raised by these differences and the solutions we propose to address them.

4.5.1 Delayed Wake-up for DMA Process

Consider the situation when the system has only an I/O process, denoted as A, that generates DMA memory requests, and a memory-intensive process, denoted as B, that generates normal memory requests. The dominant resource of both A and B is memory bandwidth. If the number of CPU cores is more than the number of actively running processes, then A and B both can get CPU when they are ready. Theoretically, if A consumes more memory bandwidth than B, its virtual time increases faster than B's, and A's virtual time will eventually become higher than B's. When the virtual time of A becomes higher than B, the memory scheduler delays the packets of A and prioritizes the packets of B, allowing B to consume more bandwidth. This way, the memory bandwidth allocation eventually becomes fair. However, in practice, A gets unfair advantages over B. The first reason is that the virtual time of a blocked process is not updated until it is unblocked. When A is blocked, the DMA-capable device generates memory requests for the DMA operation started by A. These requests are prioritized over requests generated by B because the virtual time of A lower. The second reason is that the volume of bursts of requests generated by DMA-capable devices is usually larger than that of requests generated by CPU. When the request rate is high, the memory controller starts rejecting packets when its queue becomes full. As mentioned in Section 2.3.3, the memory scheduler is a target-based solution that cannot enforce memory allocation solely by itself when the memory request

rate is higher than the capacity of the memory bus. Such a situation should be managed by a combination of source-based throttling and target-based scheduling. However, even though A has a higher virtual time than B, because there is no contention on CPU, the source throttling does not apply to this case.

To address this challenge, we throttle the CPU even if it results in one or more CPUs idling. We introduce *delayed wake-up* for recently unblocked processes. When a blocked process uses excessive memory bandwidth (more than $\frac{1}{c}$), it borrows time from the future. When the process is unblocked, it is penalized as its virtual time increases due to its consumption of excessive memory bandwidth. If the virtual time of a recently unblocked process is larger than the system virtual time plus a constant, it would not be picked to run even if there is an available CPU, and no other process that can be scheduled. A smaller value of the constant provides better fairness, while a larger value of the constant provides better performance. In our implementation, the constant is set to 10 time slices, considering the block size of our I/O operations. In other words, a process, denoted as i, would not be picked by the scheduler if it is a recently unblocked process and:

$$\texttt{vruntime}_i > V_{current} + 10 \times timeslice. \tag{4.10}$$

When other processes run and the system virtual time increases as a result of that, the penalized process will eventually get a chance to run on the CPU. When there is no other process ready and all CPUs are idling, the system virtual time will not be updated. In this case, we allow the DMA process to run on CPU since there is no contention.

However, such a solution is based on the assumption that a process continues to consume memory bandwidth after it is unblocked. If the process only needs CPU after it is unblocked, delayed wake-up introduces unnecessary delay. The trade-off between performance and fairness on this challenge is based on the ability of the memory scheduler to enforce memory allocation and the tolerance of unfairness on memory bandwidth allocation. If the hardware can enforce memory bandwidth allocation or performance is preferred over fairness in such extreme cases, then delayed wake-up is not required.

4.5.2 Best-Effortness for DMA Process

Delayed wake-up introduces a problem when there is no memory bandwidth contention. In the example of the previous section, if process B consumes only CPU time and does not create memory contention, A and B can run concurrently without interfering with each other. This is what happens in CFS since it does not consider memory bandwidth contention. However, with the delayed wake-up design, A is not allowed to consume a larger

share of memory bandwidth than the share of CPU consumed by B (i.e., $\frac{1}{c}$). This degrades the performance of A and leaves a significant portion of memory bandwidth unused.

Such a problem is caused by the property of DRFQ allocation. To ensure fairness, DRFQ allocations only guarantee the minimum share of the dominant resource for each process. However, for efficiency, the unused memory bandwidth should be allocated to the process that needs it. The difficulty to achieve such a goal is that unlike detecting idle CPU, it is hard to detect if there is spare memory bandwidth. The maximum bandwidth of memory depends on the pattern of requests. The memory can be saturated when the observed bandwidth consumption is much lower than the theoretical bandwidth. In prior works [13, 32], solutions to detect memory bandwidth saturation have been proposed. In [13], memory bandwidth saturation is detected by measuring the utilization of the read queue in the memory controller. In [32], a worst-case maximum DRAM bandwidth obtained by experiments is used to ensure that the total memory bandwidth budget does not exceed the capacity in any case. A solution that mitigates the problem without relying on the memory controller is to "allocate" idle CPU time to running processes when updating their virtual time. This increases the dominant share of processes whose dominant share is CPU. In the previous example, as DRF equalizes the dominant share, such a solution also increases the share of memory bandwidth from $\frac{1}{c}$ to $\frac{1}{n\_processes}$. $n\_processes$ is the number of processes running on CPU which is less than c (the number of CPU cores). However, neither solution is perfect and the trade-off between fairness and efficiency still needs to be considered.

4.5.3 High Variance in Memory Bandwidth Allocation

Our proposed memory scheduler uses the virtual time of processes to schedule memory requests. The design of the scheduler makes the virtual time an ordinal value. The priority of processes at the memory scheduler is only determined by the order of their virtual time. No matter how large the difference between the virtual time of two processes is, the memory scheduler behaves the same. This results in sharp changes in allocations over time as small random differences in virtual time happen frequently on processes with the same priority that should be treated the same. The randomness is introduced by the randomness in the execution of processes. For example, if a daemon process in the system is unblocked, preempts a running process, and runs for a short time, the process that is preempted has a slightly lower virtual time that the processes that are not preempted. Our experiments show that the variance in memory bandwidth allocation can reach 10%.

We propose a simple solution to smoothen the priorities. We divide the virtual time by

the length of a time slice when the kernel writes it to the registers (`virtualTime_coreId` and `virtualTime_pid`) storing the virtual time of processes in the memory controller. The virtual time stored in the `vruntime` field of `task_struct` is in nanoseconds. The length of a time slice is 4ms, or 4000000. We use an approximate calculation by left shifting the virtual time by 22. As a result, the priorities of processes become different only if there is a significant difference in virtual time. When the difference is insignificant, such as the random difference caused by a short preemption, the priorities do not change.

Chapter 5

Evaluation

In this chapter, we first describe the hardware platform we use to evaluate our proposed design. Then, we evaluate the fairness of DRFQS allocations by comparing the measured resource allocation with the theoretical fair allocations. Next, we compare the running time of workloads under DRFQS to their running time on a baseline hardware and under default kernel to evaluate the impact on performance. The results show that our proposed design achieves approximate DRF allocations for CPU time and memory bandwidth and significantly improves the performance of prioritized processes under memory bandwidth contention.

5.1 Evaluation Platform

We use full-system simulation in `gem5` to evaluate our software-hardware co-design. The hardware specifications of the evaluation platform are listed in Table 5.1. The simulated system has 4 CPU cores and each core has an L1 cache. The L2 cache is shared among all cores. We use both the original memory controller that uses the FR-FCFS scheduling policy and the modified memory controller that uses the first-ready lowest-virtual-time-first scheduling policy for comparison. The DRAM in the target system is a 1066MHz 32-bit LPDDR2 DRAM, which has a theoretical maximum bandwidth of 3.971GB/s (4.264 bytes/ns).

Component	Specification
CPU	4× ARMv8 IPC CPU (`TimingSimpleCPU`) @ 3GHz
L1 Cache	32KB I-cache, 64KB D-cache each core, 4-way set associative, 64-byte block size, 2-cycle latency
L2 Cache	2MB, 8-way set associative, 64-byte block size, 8-cycle latency
Memory Controller	One 128-packet read queue and one 128-packet write queue, Original FR-FCFS scheduling and DRFQS scheduling
DRAM Bandwidth	LPDDR2 1066MHz 32-bit, 3.971 GB/s theoretical bandwidth

Table 5.1: Evaluation Platform

5.2 Benchmarks

We run a series of benchmarks to measure the fairness and efficiency of DRFQS. The list of benchmark sets is shown in Table 5.2. We define the benchmark processes running in Linux as foreground processes and background processes. Background processes start first. When background processes finish their initialization stage, we start running the foreground processes. We make sure that background processes do not finish earlier than foreground processes by setting the parameters of both processes. As a result, the foreground processes always run with the interference created by the background processes. After the foreground processes finish, we measure the running time and the average resource allocation of the foreground processes. For sets 1 to 3, we exchange the position of processes so both of them run as a foreground and a background process.

The dominant resource of a process may change during its run time. Usually, after a process is started, it is in the initialization stage. After the initialization is finished, it enters the actual running stage. We use three basic benchmarks that have mostly constant behavior in their running stage, i.e., `mbw`, `fio`,`sjeng`.

`mbw` is a simple memory bandwidth micro-benchmark. In our experiments, we configure `mbw` to repeatedly copy from one array to another. The size of each array is 16MB. It is large enough so the caches do not affect performance. `mbw` generates 48MB of memory traffic for each copy, as write instructions are handled by the cache as read-modify-writeback. Writing to the target array causes the caches to generate additional memory read requests

No.	Foreground Process	Background Process
1	`mbw`	`mbw`
2	`mbw`	`fio`
3	`fio`	`mbw`
4	`mbw`	4× `sjeng`
5	`fio`	4× `sjeng`
6	`lbm`	None
7	`lbm`	`fio`
8	`Gems`	None
9	`Gems`	`fio`
10	`mcf`	None
11	`mcf`	`fio`

Table 5.2: Benchmark Process Sets

with a size equal to the array. `mbw` reports the time elapsed and speed for each run of the copy. When running alone, `mbw` consumes about 2700MB/s of memory bandwidth. Divided by the total memory bandwidth value used in DRFQS (4 bytes/ns, or 3815 MB/s), the share of memory bandwidth consumed by `mbw` is about 0.7. The system has 4 CPUs, so the share of CPU time consumed by `mbw` is about $\frac{1}{4}$. This makes memory bandwidth its dominant resource. However, the memory interference by other processes in the system may reduce the memory bandwidth allocated to `mbw`. When the share of memory bandwidth allocated is lower than the share of CPU time allocated, CPU time becomes `mbw`'s dominant resource. This happens when other processes consume memory bandwidth and create memory bandwidth contention. The share of memory bandwidth consumed by `mbw` decreases. However, the share of CPU time consumed by `mbw` does not decrease when there is no contention on CPU. Instead, `mbw` spends time on CPU waiting for memory requests to complete. When the memory bandwidth contention causes the share of memory bandwidth consumed by `mbw` to be lower than $\frac{1}{4}$, CPU time becomes the dominant resource of `mbw`. In our experiments, we make sure this does not happen unintentionally.

`fio` is a disk I/O micro-benchmark. In our experiment, we configure `fio` to sequentially read a 32MB file repeatedly with a block size of 1MB using direct asynchronous I/O provided by `libaio` library. `fio` reports its CPU usage when performing disk I/O operations. Since it uses asynchronous I/O, the SSD consumes memory bandwidth when `fio` is blocked. In DRFQS, the dominant resource of `fio` is CPU time when it runs on CPU, or memory bandwidth when it is blocked. Therefore, the dominant share of `fio` over a long period is the sum of the share of CPU time and the share of memory bandwidth.

When running alone, `fio` consumes about 2000MB/s of memory bandwidth. As a result, `fio` creates memory bandwidth contention if running with another process such as `mbw`.

`sjeng` is a chess engine. After initialization, it only consumes CPU time. As a result, the dominant resource of `sjeng` is always CPU in its running stage. When running alone, `sjeng` consumes 100% of one CPU core and never blocks. Therefore, when running with other processes, `sjeng` consumes as much CPU time as available. We use this to calculate `sjeng`'s CPU usage when running under contention.

Other than these basic micro-benchmarks, we run 3 benchmarks from SPEC2006 with different memory intensiveness to represent real-world workloads: `lbm`, `Gems`, and `mcf`, ordered by their memory intensiveness from high to low.

When calculating the share of memory bandwidth allocation, we need to consider the fact that the theoretical maximum bandwidth is impossible to reach with most workloads. In DRFQS, the maximum bandwidth is used as the total available memory bandwidth. It is used to: (i) identify the dominant resource and (ii) calculate the virtual time. DRFQS uses 4 bytes/ns (3815 MB/s) to simplify calculations. When analyzing the allocation results, we choose the total available memory bandwidth with the following principle. When there is no memory bandwidth contention, we use 3.725 GB/s which is used by DRFQS as the total available memory bandwidth. When there is memory bandwidth contention, it means that the memory bandwidth is saturated. Therefore, the total available memory bandwidth is the sum of the memory bandwidth consumed by all processes. We determine the existence of memory bandwidth contention by comparing the memory bandwidth allocated to processes to their demand. The demand for memory bandwidth of a process is the memory bandwidth consumption measured when it runs alone. If the memory bandwidth allocated to a process is lower than its demand and is not caused by the delayed wakeup mechanism, there is a memory bandwidth contention.

5.3 Evaluation of Resource Allocation

In this section, we evaluate the resource allocation by DRFQS. We run benchmarks and measure the dominant share of each process. The goal of DRFQS is to make the dominant share of each process proportional to its weight. We compare the ratio between the dominant share of processes and compare it with the ratio between their weights. If DRFQS achieves DRF allocation, then the ratio between the dominant share of any two processes is the same as the ratio between the weight of the two processes.

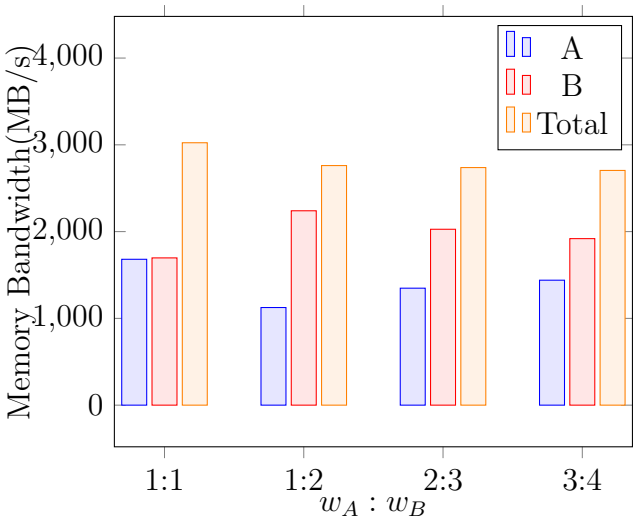


Figure 5.1: Memory Bandwidth Consumption of two `mbw` processes

5.3.1 Allocation of Memory Bandwidth

Workloads without DMA

We run set 1 with two `mbw` processes, A and B, of different weights. We change the weight of A and B and measure the memory bandwidth allocated to them. We make sure that the difference between the weights does not make CPU time becomes the dominant resource of any of them by setting their weights close to each other. The share of CPU time consumed by A and B is always $\frac{1}{4}$. The result of memory bandwidth consumption is shown in Fig. 5.1.

Fig. 5.1 shows that the memory bandwidth allocated to A and B differs when the ratio between their weights changes. The total memory bandwidth consumed by both A and B also slightly differs. In all cases, the memory bandwidth allocated to both A and B is more than 1000 MB/s. Compared to the maximum memory bandwidth used by DRFQS (3815 MB/s), the share of memory bandwidth is larger than the share of CPU time ($\frac{1}{4}$). Therefore, we calculate the ratio between A's dominant share and B's dominant share by calculating the ratio between the memory bandwidth allocated to A and B. We compare it to the ratio between their weights, as shown in Fig. 5.2.

Fig. 5.2 shows that the ratio between the dominant share of the two `mbw` processes is almost identical to the ratio between their weights. The actual allocation by DRFQS is almost the same as the theoretical DRF allocation. When the weight of A and B is 1:1,

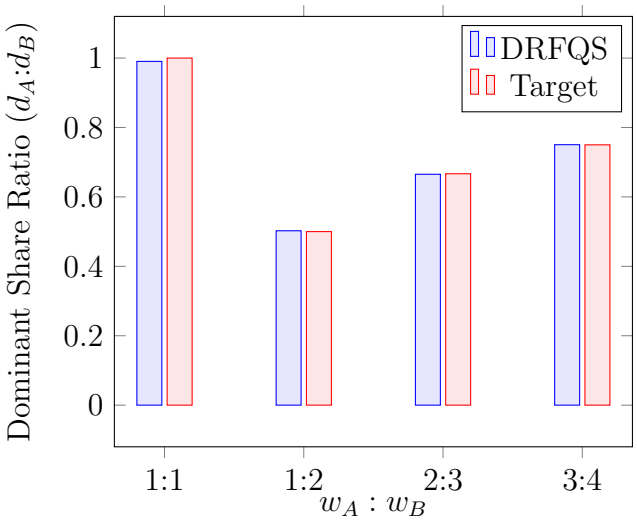


Figure 5.2: Ratio of Dominant Share between Two `mbw` Processes

1:2, 2:3, and 3:4, the error is -0.96%, 0.48%, -0.22%, and 0.05%, respectively. In these experiments, `mbw` always runs on CPU. The memory bandwidth allocation is enforced only by target-based arbitration at the memory controller based on the input from the kernel scheduler. The result shows that for processes that consume memory bandwidth by CPU read and write, DRFQS allocates memory bandwidth proportionally to their weights.

Workloads with DMA

In this experiment, we run set 2 and set 3 with different weights for `mbw` and `fio`. We measure the CPU time and memory bandwidth consumption for each combination of weights. The CPU time allocation is shown in Fig. 5.3 and the memory bandwidth consumption is shown in Fig. 5.4.

The dominant share of `mbw` is its share of memory bandwidth. The dominant share of `fio` is its share of memory bandwidth plus its share of CPU. We calculate the ratio of dominant share and compare it with the target DRF allocation, as shown in Fig. 5.5.

Fig. 5.5 shows that the ratio between the dominant share of `mbw` and `fio` closely follows the ratio between their weights. However, the difference is larger than in the previous experiment of two `mbw` processes. When the weight of `mbw` and `fio` is 1:1, 2:1, 3:1, 4:1, 5:1, the error is -3.2%, 6.9%, 8.1%, 7.0%, 9.0%. In this experiment, the memory bandwidth allocation is enforced by both source-based throttling at the kernel scheduler (the delayed

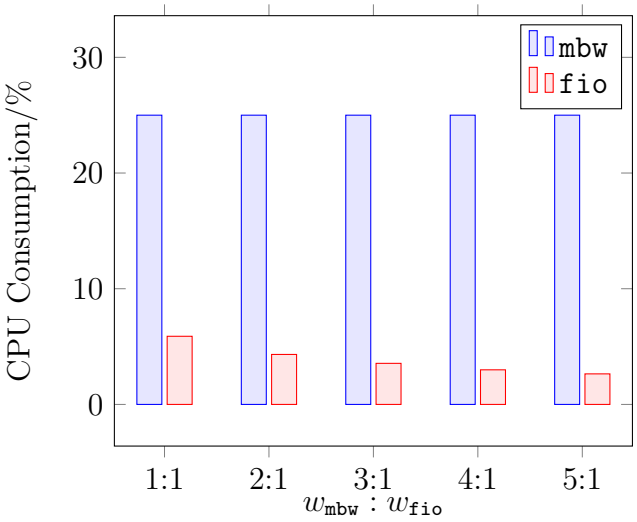


Figure 5.3: CPU Consumption of `mbw` and `fio`

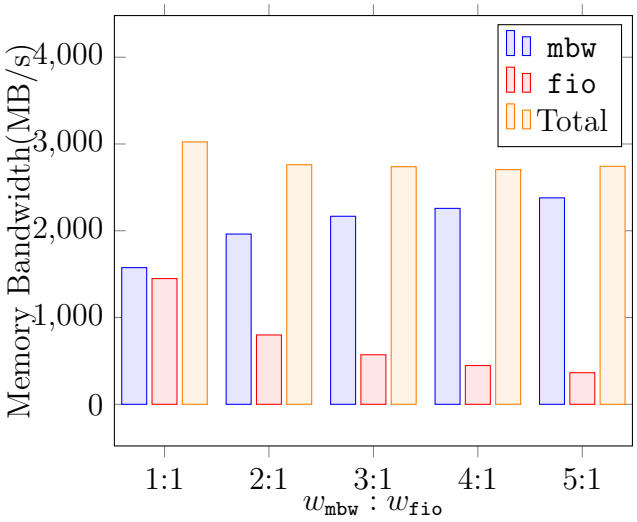


Figure 5.4: Memory Bandwidth Consumption of `mbw` and `fio`

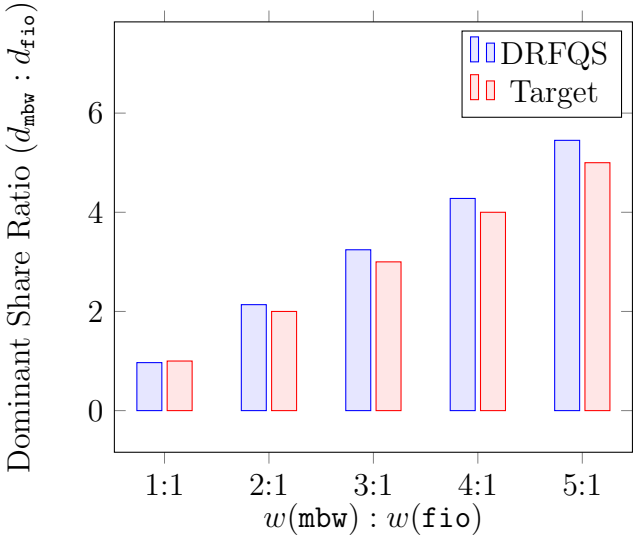


Figure 5.5: Ratio of Dominant Share between `mbw` and `fio` and Target Ratio

wake-up applied to `fio`) and the target-based arbitration at the memory controller. The larger error is caused by the difference in memory request pattern between `mbw` and `fio` and delayed wake-up which only applies to `fio`.

5.3.2 Allocation of Different Dominant Resources

We run set 4 and set 5 with the same weight for all processes. We use the assumption that the CPU time not allocated to `mbw` or `fio` is equally shared by the 4 `sjeng` processes. The share of allocation on CPU time and memory bandwidth and the dominant share of each `mbw` and each `sjeng` process when running set 4 is shown in Fig. 5.6. The same shares for `fio` and each `sjeng` process when running set 5 is shown in Fig. 5.7.

The result shows that DRFQS equalizes the dominant share of processes. In set 4, the dominant share of `mbw` is only 1.1% more than that of `sjeng`. In set 5, the dominant share of `fio` is only 0.9% more than that of `sjeng`. The allocation also achieves higher utilization of system resources. In set 4 and set 5, there are 5 processes in total. With max-min allocation (see Section for details), each process can only get 20% of all system resources. With DRFQS, the dominant share of all of them is more than 20%.

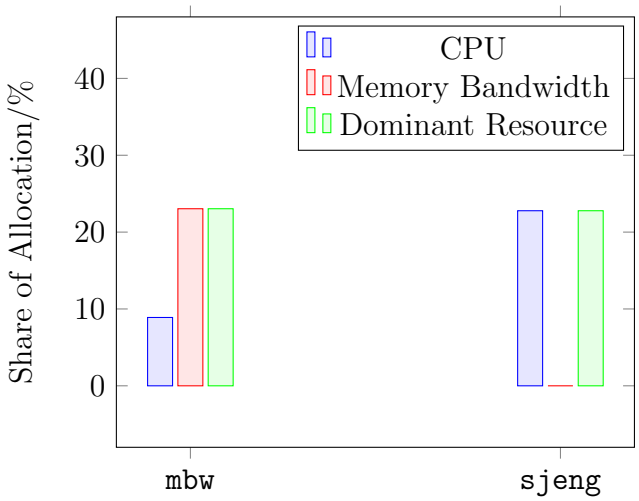


Figure 5.6: Share of Allocation of `mbw` and `sjeng`

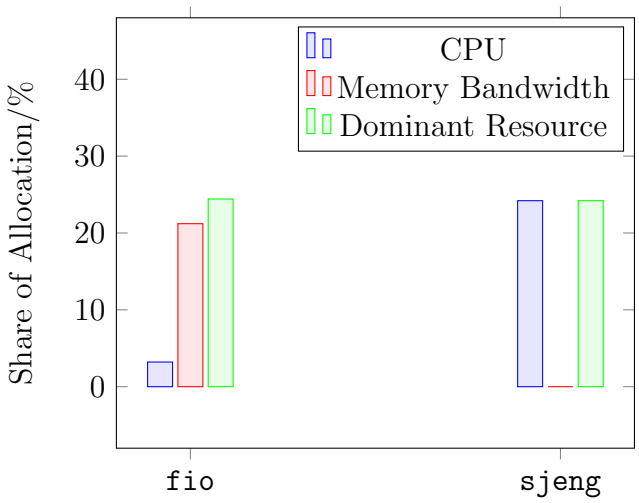


Figure 5.7: Share of Allocation of `fio` and `sjeng`

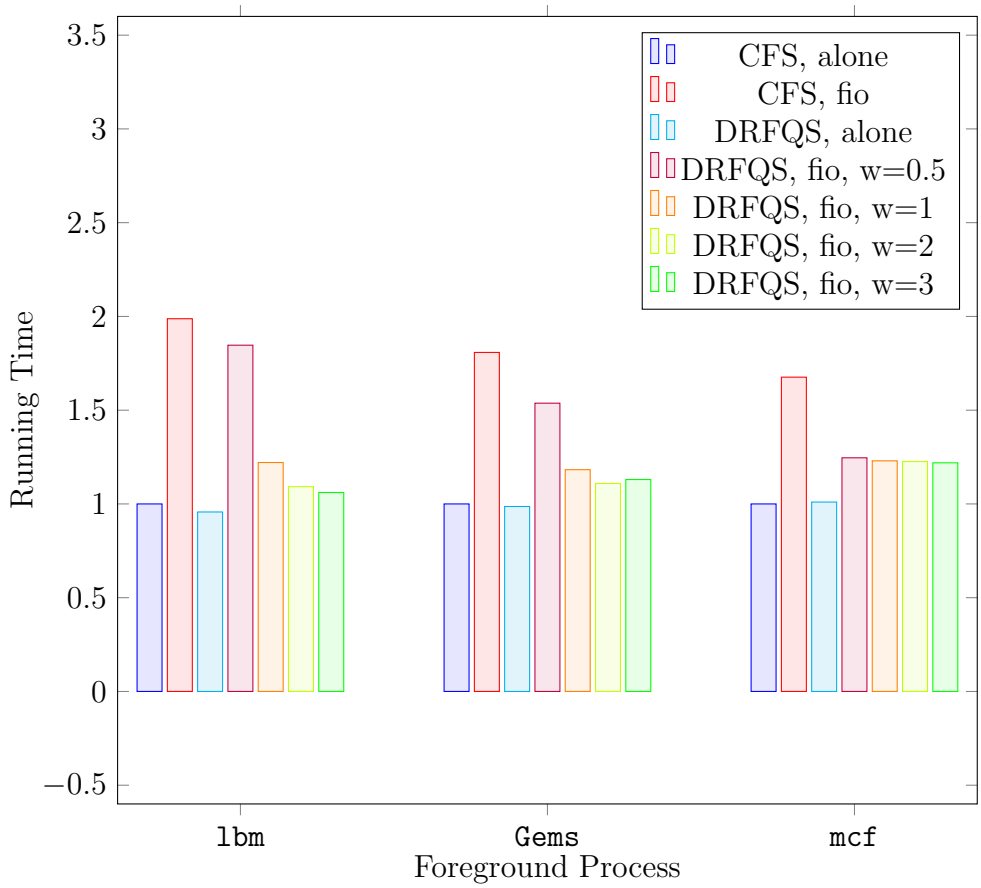


Figure 5.8: Running Time of SPEC2006 Benchmarks

5.4 Comparison of Running Time

In this experiment, we run sets 6-12 in both CFS and DRFQS. In sets 6, 8, and 10, the SPEC2006 benchmark runs without resource contention. In set 7, 9, 11, the SPEC2006 benchmark runs with `fio`. In DRFQS, We adjust the weight of the benchmark and fix the weight of `fio` to 1. In CFS, as there is no CPU contention, the weight of processes does not affect performance. We only run the processes with default weight in CFS. We measure the running time of the benchmark and compare the result of DRFQS to that of CFS. The result is shown in Fig. 5.8. All running times are normalized to the running time in CFS without resource contention.

Fig. 5.8 shows that DRFQS can significantly reduce the performance impact of memory

bandwidth contention by implementing multi-resource fair allocation of CPU and memory bandwidth on processes. In CFS, the benchmark is 1.8-2 times slower when `fio` creates memory contention. In DRFQS, the slowdown of the benchmark is controlled by the weight of the processes. When the weight of the benchmark process is the default value 1, the running time is less than 25% longer than without memory bandwidth contention. If the memory-intensiveness of the benchmark is high (e.g., `lbm`), increasing the weight of the benchmark can reduce the running time even more. This is achieved by increasing the benchmark's allocation of memory bandwidth to satisfy the high demand when increasing the benchmark's weight. However, for benchmarks less bounded by memory bandwidth, the effectiveness of DRFQS decreases.

5.5 Conclusion

Based on our evaluation, we conclude that our proposed design, DRFQS, can achieve dominant resource fairness on CPU time and memory bandwidth in the target system. DRFQS makes the share of the dominant resource of each process proportional to the weight of the process. With such fairness, the performance impact of memory bandwidth contention is addressed. The running time of processes under memory bandwidth contention can be controlled by the weight of processes.

Chapter 6

Conclusions and Future Work

In operating systems, memory bandwidth contention between processes has a significant impact on performance. It is necessary for operating systems to manage memory bandwidth allocation for processes. To extend operating systems from allocating CPU time to allocating CPU time and memory bandwidth, two major problems need to be resolved. First, the operating system needs information about memory bandwidth consumption per process. Second, the operating system needs to allocate memory bandwidth to processes and effectively enforce the allocations.

Our proposed solution supports per-process memory traffic tracking by simple modifications to existing hardware and software. In our verification, our proposed solution captures most memory traffic and attributes them to the responsible process. Our proposed solution allocates CPU time and memory bandwidth, achieving dominant resource fairness (DRF). The memory bandwidth allocation is enforced by source throttling in the CPU scheduler and target arbitration in the memory controller. In our evaluation, we show that our implementation achieves DRF allocations. Our results also show that such allocations effectively manage the performance impact made by memory bandwidth contention, significantly reducing performance degradation for prioritized processes.

For the future work, our proposed design can be extended in several dimensions. First, simulation results could be improved by boosting the memory performance of the simulated CPU, replacing the DRAM with higher performance models (e.g., DDR4 DRAM) and adding other DMA-capable devices (e.g., GPU, NIC, and FPGA).

Second, the memory traffic tracking and memory packet scheduling we proposed can be implemented in HDL simulations or ASICs. The software simulator is less accurate than HDL simulations or ASICs, and it does not provide any insight into the cost of area and

power supply when implementing them in ASICs. To implement these features on real hardware, practical challenges, such as delays, need to be considered.

Third, the CPU scheduler we implemented has problems inherited from the original BFS scheduler. When running on a system with a large number of CPU cores, lock contention may degrade the performance. In future work, a per-CPU run queue design should be used while also supporting the DRF allocation of CPU time and memory bandwidth.

Lastly, the multi-resource fair allocation we proposed in our model is based on single-threaded process. For multi-threaded processes, Linux treats each thread as a process (light-weight process, LWP). A possible solution to address this challenge is to split the weight of the process to its threads. A process can assign different weights to its threads.

References

[1] Homa Aghilinasab, Waqar Ali, Heechul Yun, and Rodolfo Pellizzoni. Dynamic memory bandwidth allocation for real-time gpu-based soc platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3348–3360, 2020.

[2] Sungyong Ahn, Kwanghyun La, and Jihong Kim. Improving I/O resource sharing of linux cgroup for NVMe SSDs on multi-core systems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, June 2016. USENIX Association.

[3] Andy Bavier and Larry Peterson. The power of virtual time for multimedia scheduling. 07 2000.

[4] Anastasiia Butko, Rafael Garibotti, Luciano Ost, and Gilles Sassatelli. Accuracy evaluation of gem5 simulator system. In *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–7, 2012.

[5] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. volume 41, pages 77–88, 05 2013.

[6] Kenneth Duda and David Cheriton. Borrowed-virtual-time (BVT) scheduling. *ACM SIGOPS Operating Systems Review*, 34:27–28, 04 2000.

[7] Jos Ewert. Estimating I/O Memory Bandwidth. 2016.

[8] Juan Fang, Wang Mengxuan, and Zelin Wei. A memory scheduling strategy for eliminating memory access interference in heterogeneous system. *The Journal of Supercomputing*, 76, 04 2020.

[9] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. *ACM SIGCOMM Computer Communication Review*, 42, 09 2012.

[10] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andrew Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of heterogeneous resources in datacenters. 04 2010.

[11] graysky. Cpu schedulers compared. Technical report, 2012.

[12] Taylor L. Groves and Eric Schulte. Bfs vs. cfs scheduler comparison. 2009.

[13] Derek Hower, Harold Cain, and Carl Waldspurger. Pabst: Proportionally allocated bandwidth at the source and target. pages 505–516, 02 2017.

[14] Adrian Huang. *An Introduction to the IOMMU Infrastructure in the Linux Kernel*. Lenovo, 2021.

[15] Facebook Inc. Io controller - cgroup v2. Technical report, 2019.

[16] Min Kyu Jeong, Mattan Erez, Chander Sudanthi, and Nigel Paver. A qos-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 850–855, New York, NY, USA, 2012. Association for Computing Machinery.

[17] Jungho Kim, Philkyue Shin, Myungsun Kim, and Seongsoo Hong. Memory-aware fair-share scheduling for improved performance isolation in the linux kernel. *IEEE Access*, 8:98874–98886, 2020.

[18] Johannes Langguth, Xing Cai, and Mohammed Sourouri. Memory bandwidth contention: Communication vs computation tradeoffs in supercomputers with multicore architectures. pages 497–506, 12 2018.

[19] Tong Li, Dan Baumberger, and Scott Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, page 65–74, New York, NY, USA, 2009. Association for Computing Machinery.

[20] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: A decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[21] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. pages 146–160, 12 2007.

[22] Mahdi Nazm Bojnordi and Engin Ipek. Pardis: A programmable memory controller for the ddrx interfacing standards. volume 31, pages 13–24, 06 2012.

[23] Yue Qian, Zhonghai Lu, and Qiang Dou. Qos scheduling for nocs: Strict priority queueing versus weighted round robin. In *2010 IEEE International Conference on Computer Design*, pages 52–59, 2010.

[24] Rami Rosen. Resource management: Linux kernel namespaces and cgroups. *Haifux, May*, 186:70, 2013.

[25] Gero Schwäricke, Rohan Tabish, Rodolfo Pellizzoni, Renato Mancuso, Andrea Bastoni, Alexander Zuepke, and Marco Caccamo. A real-time virtio-based framework for predictable inter-vm communication. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 27–40, 2021.

[26] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Soffa. The impact of memory subsystem resource sharing on datacenter applications. pages 283–294, 07 2011.

[27] Hiroyuki Usui, Lavanya Subramanian, Kevin Chang, and Onur Mutlu. Squash: Simple qos-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators. *ACM Transactions on Architecture and Code Optimization*, 12, 05 2015.

[28] C. A. Waldspurger and E. Weihl. W. Stride scheduling: Deterministic proportional-share resource management. Technical report, USA, 1995.

[29] Carl Waldspurger and William Weihl. Lottery scheduling: Flexible proportional-share resource management. 11 2001.

[30] Wikipedia. Brain Fuck Scheduler — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Brain%20Fuck%20Scheduler&oldid=1089307402, 2022. [Online; accessed 19-July-2022].

[31] Heechul Yun, Waqar Ali, Santosh Gondi, and Siddhartha Biswas. Bwlock: A dynamic memory access control framework for soft real-time applications on multicore platforms. *IEEE Transactions on Computers*, 66(7):1247–1252, 2017.

[32] Heechul Yun, Gang Yao, R. Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. pages 55–64, 04 2013.

[33] Zhi-Hui Zhan, Xiao-Fang Liu, Yue-Jiao Gong, Jun Zhang, Henry Shu-Hung Chung, and Yun Li. Cloud computing resource scheduling and a survey of its evolutionary approaches. *ACM Comput. Surv.*, 47(4), jul 2015.