A task-based multi-core allocation mechanism for packet acceleration

Zerun Li, Yang Guo, Sheng Liu, and Junyang Zhang
College of Computer, National University of Defense Technology, 109 Deya Road, Kaifu District, Changsha, Hunan, China
a) li_zr@foxmail.com

Abstract: Task allocation is a traditional and complicated problem for a system with many peripherals. Normal allocation methodology focuses on fuzzy data transfer but is unable to analyze some specific experimental statistics and does not improve system performance. Data Director (DD) is an innovative packet-based manager, is comprised of Data Index (DI) and Packet Direct Memory Access (PDMA) and realizes the enhanced data transfer in a complex system. This paper presents an improved task allocation model to enhance data transfer in a task-based model with three different allocation methodologies. The experimental results for the optimization allocations show that the proposed methodologies reduce the delay of transfer differently due to the schedule costs in these cases.

Keywords: data transfer, queue optimization, task scheduling, DMA

Classification: Integrated circuits

References
[1] I. Benacer, et al.: “A high-speed traffic manager architecture for flow-based networking,” New Circuits and Systems Conference (2017) 161 (DOI: 10.1109/newcas.2017.8010130).
[2] Q. Zhang, et al.: “A scalable and programmable modular queue manager architecture,” Asilomar Conference on Signals, Systems & Computers (2011) 1124 (DOI: 10.1109/acssc.2010.5757579).
[3] H. Kim, et al.: “A memory-efficient heterogeneous parallel pattern matching scheme in deep packet inspection,” IEICE Trans. Electron. 7 (2010) 377 (DOI: 10.1587/elex.7.377).
[4] Y. Xu, et al.: “A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues,” Inf. Sci. 270 (2014) 255 (DOI: 10.1016/j.ins.2014.02.122).
[5] M. Akbari, et al.: “An enhanced genetic algorithm with new operators for task scheduling in heterogeneous computing systems,” Eng. Appl. Artif. Intell. 61 (2017) 35 (DOI: 10.1016/j.engappai.2017.02.013).
[6] N. Lynn and P. N. Suganthan: “Ensemble particle swarm optimizer,” Appl. Soft Comput. 55 (2017) 533 (DOI: 10.1016/j.asoc.2017.02.007).
[7] A. M. El-Nagar and M. El-Bardini: “Parallel realization for self-tuning interval type-2 fuzzy controller,” Eng. Appl. Artif. Intell. 61 (2017) 8 (DOI: 10.1016/j.engappai.2017.02.011).
[8] A. Jindal, et al.: “Network design and optimization for multi-product, multi-time, multi-echelon closed-loop supply chain under uncertainty,” Procedia
1 Introduction

It is a challenge for a complex system with different peripherals to implement task scheduling algorithms that handle the subtasks of an application [1, 2]. The heterogeneity model of a data processing system can be divided into two categories, i.e., the Processor-Based Heterogeneity Model (PHM) and the Task-Based Heterogeneity Model (THM). In a PHM computing system, a processor executes all tasks at the same speed, regardless of the type of each task. On the other hand, in a THM computing system, how fast a processor executes a task depends on whether the heterogeneous processor architecture matches the task requirements and memory features [3]. We use a THM computing system to balance the tasks of a multi-core processor. The goal of a task scheduling algorithm is typically to schedule all the subtasks on a given number of available processors in order to minimize runtime without violating precedence constraints.

The genetic algorithm, a popular heuristic allocation methodology, assigns a priority to each subtask while using a heuristic earliest finish time (EFT) approach to search for a solution for the task-to-processor mapping. Many factors can affect the efficiency of the genetic algorithm, such as the initial population size and the method by which its modifications are chosen and the particular crossover and mutation operators. This meta-heuristic algorithm is simple, flexible, and ergodic [4]. Besides generating schedules with lowered repetitions, which allows for maximum parallelization for improved performance of the tasks, it also reduces execution time and helps to avoid conflicts with local optimums [5]. Although genetic allocation is improved to guarantee variety and consistent coverage of problem space, it is laborious to complete the overall configuration at the beginning of data transfer to make the system operate under supervision. Furthermore, an ensemble particle swarm optimizer uses inertia weight to adjust the balance in the search progress in order to update the positions and the velocity of the particles. However, this methodology is too expensive for the self-adaptation when transferring the data instead of configuring the transmission order at the beginning of data transfer [6]. Besides, while a tuning mechanism generates the updating factors to tune the output scaling factors, it is also efficient for initializing the transfer configuration before the start of the transmission system [7].

This paper proposes an innovative data transfer scheme to configure data transfer at the beginning of data transfer and to serve the peripherals with data demands simultaneously. Three possible cases, based on a system with eight or four
homogeneity cores, are used to evaluate the performance of this allocation scheme. The outline of this paper is as follows. Section 2 will explain the basic mode of the data transfer and the related firmware. Section 3 will present three cases to evaluate the scheme. Section 4 will explain the experiment results and will compare the differences between the allocation methodologies. Section 5 will conclude the conclusion of the research.

2 Allocation model

In a multi-core processor, different cores and peripherals may initiate data transfer when they need to communicate with other devices in the system. Data Director (DD), comprised of several Packet Direct Memory Accesses (PDMAs) and a Queue Controller (QC), can describe the features of information transferred across different modules to facilitate data transfer and to offer a consistent Application Programming Interface (API) to the host software. The transferred feature information is collectively referred to as DI (Data Index).

Most tasks are often offloaded from the peripheral hardware to the central processor to enhance calculation performance. Significant performance improvement may result from the comprehensive design of the host software and the peripheral modules interface. In a multi-functional network, the transmission and reception of the packet are crucial tasks. The DD can maximize the efficiency of interaction between the API and the peripherals’ modules.

2.1 Description for the transfer behaviors

DD can function like a direct memory access and serve the packet-based data transfer. The demand for transfer data is defined in DI, which can be popped from or pushed to related queues to realize the data movement.

A DI includes a small space for describing a series of data that needs to be transferred between cores and peripherals. In a multi-core structure of packet data structure processors, the scheduler of the system requires effective arbitration methodology based on sufficient priorities [8]. Fig. 1 illustrates the basic transfer flow of DIs via DD. A DI can indicate the data address by a pointer to a data buffer located in the memory and does not include any data that needs to be processed at the destination. In order to deal with the access conflict when a peripheral receives several DIs from other transmitting peripherals, a DI supports a chain transfer mode to transfer DIs with the same destination. The first DI in a series of DIs is called a
Data Index Front (DIF), and the remaining DIs are called Data Index Buffers (DIBs). A DIF has fixed DI room which includes a pointer to a data buffer and, optionally, a pointer to link some DIBs. DIBs are flexible when their sizes are configured, but it is necessary to link the DIB to the DIF rather than allocating the DIB at the beginning of the transfer queue.

A packet is the logical grouping of a DI and the payload data attached to it. The payload data may be referred to as a data buffer, and depending on the DI type, it may be contiguous with the DI fields or it may be somewhere else in the memory, with a pointer stored in the DI.

The sub-module required to transfer the DI is called a PDMA. The PDMA only works in receiving mode and transmitting mode. In receiving mode, the PDMA fetches a free DI, traverses the DI to find the data referred to, transfers the payload into the buffer, and puts the DI into the destination queue. In transmitting mode, the PDMA pops the DI from the transfer queue, traverses the DI, reads the load data onto the buffer, and transfers the data to the transmit port.

In order to manage the queues transferred in the PDMA of different modules, a manager module is useful for monitoring the queues and realizing the allocation and accumulation for DIs. This is called a QC. The QC can configure the source address and destination peripheral of a DI. It can also accelerate the management of the packet queues. Packets are added to a packet queue by writing the 32-bit DI address to a specific sub-module in the QC. Packets are de-queued by reading the same location for that particular queue. It is noteworthy that the QC can only transfer the DI allocated from the DI regions in the memory-associated QC. Fig. 2 presents the firmware related to the data transfer of DD.

Effective configuration is essential for transferring the DI and realizing the high-speed transport among cores and different peripherals. It is necessary to allocate the superfluous tasks to the idle cores and to balance the loads of the cores. The key configuration for the queue is the linking memory, which contains the next DI of the current DI. When dealing with a complex mission, it is useful to dispatch appropriate DIs to different cores in order to guarantee the efficient work of the cores or the peripherals.

![Fig. 2. Data transfer illustration of DD](image-url)
2.2 Basic operations for queues comprised of packets

Basic push operations for DIs are described as follows:

1. A master device in the system writes the address of a DI to a register for a queue and generates a push request for the QC.
2. The QC reads the tail pointer to the queue to get the last DI in the queue.
3. The QC modifies the linking information from the last DI to make it refer to the new DI that is pushed in.
4. The QC modifies the tail pointer to the queue and make it point to the new DI that is pushed in.
5. The QC modifies the linking entry of the new descriptor, making it NULL.

Basic pop operations for a DI are as follows:

1. A master device in the system reads a register for a queue, and the register generates a PUSH request for the QC.
2. The QC reads the header pointer to the queue to get the first DI in the queue and the address of the first descriptor is returned to the master.
3. The QC reads the linking entry of the first DI to find the second DI in the queue.
4. The QC modifies the header pointer to the queue and makes it point to the second DI in the queue.

The push operation usually operates in a few cycles on a master device because it is normally a write operation of the master and it does not stall the master. Meanwhile, the pop operation often stalls the master because it is actually a read operation, and the master needs the response from the QC.

According to the user’s configuration, the QC monitors one or more specific queues. Once there is a DI in the queue, the QC pops it from the queue and writes the pointer to the descriptor to a buffer, which is called an accumulation buffer; normally it is in the local L2 of a core. When the QC accumulates a specific number of DIs for the accumulation buffer, it triggers an interruption to the core, and then the core reads the DIs from the core’s local L2 instead of from the QC, which saves time for each DI read operation.

The master writes the pointer to a DI into the internal PDMA inside the QC that indicates the transmit queues for the desired channel. Channels may provide more than one queue to transfer DIs and it is necessary to provide a particular prioritization policy of the queues. This behavior is configured by the application program interface and is controlled by the QC.

When several DIs are queued together, it is clear that the DIF must rank first in a series of DIFs and DIBs with several data buffer packets. In other words, the DIF is supposed to be at the start of a packet, followed by zero or more DIBs. When all the DIs in a packet are allocated, the data transfer action is configured and ready, and it is then feasible to trigger the transport of the data onto the devices demanding the data. In terms of a peripheral access to a DIF or DIB, there is no need to wait for conveying the data buffer. It is practical to access the next DI immediately without catching the data buffer. This methodology can save much time in linking all the DIs in a packet.

Usually, the push operation is continuous, so the wait time of the push operations may be long in terms of recalling the push operation. Meanwhile, a
pop operation needs more machine cycles to read the DI from a transfer queue because the QC needs more time to react to the request for data than is the case in a push operation. If multiple DIs need to be popped, an accumulator may be used to accumulate multiple DIs from the QC into core’s local memory. Then core can read descriptors in its local memory, and the time required to read a DI from local memory is quite short. Fig. 3 shows the linking relationship for two queues.

![Diagram showing data flow for packets in two queues](image)

**Fig. 3.** Data flow for packets in two queues

### 3 Different methods for evaluating the data transfer

The data transfer between the devices in a multi-core system relies on the data blocks’ characteristic of adaptation to a different environment. The transfer tasks usually rely on the allocation speed of the bus and the processing ability of the destinations. However, it is essential to establish whether the data blocks are indivisible and whether the destination peripherals are in demand of the data transfer [9]. Three allocation cases will now be presented to test the performance of the allocation methodology based on the eight or four processing cores.

#### 3.1 Average allocation

When the system needs to transfer eight data blocks between the devices, we assume that the data lengths are $l_1, l_2 \ldots l_8$ respectively. Furthermore, each data block needs to be transferred to eight cores with their own PDMAs. The average transfer length is $\sum_{i=1}^{8} l_i / 8$. Each core can take $\sum_{i=1}^{8} l_i / 8$ to balance the tradeoff on transferring the data blocks. In this case, we transfer a $1 \times 128$ matrix, a $2 \times 128$ matrix, a $3 \times 128$ matrix, a $4 \times 128$ matrix, a $5 \times 128$ matrix, a $6 \times 128$ matrix, a $7 \times 128$ matrix, and an $8 \times 128$ matrix to eight cores and the longest delay is determined by the transfer of the $8 \times 128$ matrix. However, if the matrices are divisible, the input matrices can be allocated to each core with the same size to balance the load of each core. Fig. 4 offers the time required for calculating different matrices with eight cores using the Average Allocation (AA) methodology.
When four cores calculate the same size of the matrices, the average transfer length is $\sum_{i=1}^{4} l_i / 4$. we transfer a $3 \times 128$ matrix, a $7 \times 128$ matrix, an $11 \times 128$ matrix, and a $15 \times 128$ matrix to four cores and the longest delay is determined by the transfer of the $8128$ matrix. Fig. 5 illustrates the time required for calculating different matrices with four cores.

### 3.2 Long data first allocation
In some cases, the long data block is indivisible. We assume that $l_1$ is the block that needs to be transferred to a single core. Therefore, core1 can process the data block $l_1$ with a single PDMA. The other data blocks can be transferred using the methodology of AA. The most obvious dispatching rule that should be used involves orders with the highest positive lateness being the first to be delivered (since we consider that orders with negative lateness are not delivered) [10]. This allocation methodology is applicable to the fixed demand for an indivisible data block.

In this way, an $8 \times 128$ matrix is transferred to each core before all eight cores receive a $1 \times 128$ matrix, a $2 \times 128$ matrix, a $3 \times 128$ matrix, a $4 \times 128$ matrix, a $5 \times 128$ matrix, a $6 \times 128$ matrix, a $7 \times 128$ matrix, and an $8 \times 128$ matrix respectively. These eight matrices are divisible so that we can optimize the load balance to divide the latter eight matrices. Fig. 4 compares the original delays in calculating the matrices with the optimized delay using the Long Data First Allocation (LDFA) methodology by eight cores.

When implementing four cores, an $15 \times 128$ matrix is transferred to each core before four cores receive a $3 \times 128$ matrix, a $7 \times 128$ matrix, an $11 \times 128$ matrix, and a $15 \times 128$ matrix respectively. Fig. 5 compares the original delays in calculating the matrices with the optimized delay using the Long Data First Allocation (LDFA) methodology by four cores.

### 3.3 Irregular allocation
This algorithm uses segmentation, representing every data block, to divide scheduling into several segments, and every segment is indivisible. Each segment should cover independent tasks that allow for parallel execution on different processors. When the lengths of segmentations are visible ahead of time, it is feasible to allocate the subtasks with a sequence in a descending order. The calculation loads in one area may cause rising allocation demands in the areas nearby [11].

We use one core to calculate the $8 \times 128$ and $7 \times 128$ matrices, the original calculation operation, and eight cores to calculate the $7 \times 128$ and $1 \times 128$ matrices, the $6 \times 128$ and $2 \times 128$ matrices, the $5 \times 128$ and $3 \times 128$ matrices, the $4 \times 128$ and $4 \times 128$ matrices, the $3 \times 128$ and $5 \times 128$ matrices, the $2 \times 128$ and $6 \times 128$ matrices, and the $1 \times 128$ and $7 \times 128$ matrices respectively. Fig. 4 illustrates the delays by eight cores in calculating the matrices using Irregular Allocation (IA).

For four cores computing the matrices with the same length, they receive the $3 \times 128$ and $15 \times 128$ matrices, the $7 \times 128$ and $11 \times 128$ matrices, the $7 \times 128$ and $11 \times 128$ matrices, and the $15 \times 128$ and $3 \times 128$ matrices respectively. Fig. 5 offers the delays by four cores using Irregular Allocation (IA) together with the original delays.
4 Results and analysis

In this paper, the data that needs to be transferred is called the event. Fig. 4 shows the experimental results of the three allocations. Furthermore, the matrix columns are prolonged with 256, 384, 512, and 640 to compare the optimization effects using different matrix sizes. The necessary initial information about the event includes the data length, the address of the data in the memory space, the destination peripheral for the data, and the priority of the data in the transfer queue. As the data block length increases, the optimization for the transfer delay is inconspicuous. The reason for this phenomenon may be that the cost of scheduling makes up most of the transfer delay. The optimization focuses on the allocation of the different tasks, except the transfer delay for scheduling.
According to the statistics from Table I, AA and LDFA have better optimization results using four cores than eight cores, but IA performs more efficiently with eight cores than four cores. We can also find that the improvement to AA fluctuates from 14.27% to 29.65%. The scheduling cost of the eight destinations is quite high, so the improvement cannot achieve 12.5% (1/8) for each core. The AA methodology is not very practical most of the time. The average dispatch mode is not applicable to some common conditions. The enhancement of LDFA is from 7.14% to 14.83%. The indivisible data block takes up the majority of the data transfer, so the performance improvement here is less than is achieved by using AA. The performance improvement using IA can reach 26.85%. This is because IA can balance the data load to every core favorably. This methodology can save scheduling costs to a great extent because the initial configuration allocates a large number of resources with small resources and realizes the load balance across the cores.

From the results above, AA is the most beneficial because it can efficiently transfer every data block to realize the sufficient implementation of the resources. LDFA has the fixed transfer delay time and its time delay is inevitable; therefore, its optimization cannot be compared with AA. However, IA can transfer the data block to achieve improved performance.

### Table I. Optimization results of different allocations (%)

| Columns of Matrices | 128   | 256   | 384   | 512   | 640   |
|---------------------|-------|-------|-------|-------|-------|
| AA (Four Cores)     | 17.12 | 22.23 | 26.12 | 28.06 | 29.25 |
| AA (Eight Cores)    | 14.27 | 20.01 | 25.06 | 27.83 | 29.65 |
| LDFA (Four Cores)   | 8.56  | 11.12 | 13.06 | 14.03 | 14.62 |
| LDFA (Eight Cores)  | 7.14  | 10.01 | 12.53 | 13.91 | 14.83 |
| IA (Four Cores)     | 12.11 | 16.01 | 19.08 | 20.63 | 21.68 |
| IA (Eight Cores)    | 12.74 | 17.96 | 22.66 | 25.31 | 26.85 |

5 Conclusions

This paper presented a novel allocation model for transferring data across devices. The three allocation methodologies were applied to different scenarios. The improvement in the task schedule was based on the matrix calculation, and we compared the results of the matrix calculations. AA can deal with similar tasks in a system. LDFA can transfer some tasks with indivisible data blocks. IA can calculate tasks whose lengths are different and unchangeable. However, the allocation methodologies are not relevant in some common applications. In the future, we plan to expand the task allocation methodologies to different sources and destinations in the system and compare the time delay improvements.