Traffic-Aware and Memory-Aware Task Scheduling on Multi-Core Chips

Hongyu Meng
Institute of Automation, Chinese Academy of Sciences
University of Chinese Academy of Sciences
95 Zhongguancun East Road,100190, Beijing, China
menghongyu2014@ia.ac.cn

Yang Guo, Zijun Liu and Donglin Wang
Institute of Automation, Chinese Academy of Sciences
95 Zhongguancun East Road,100190, Beijing, China
{ guoyang2014 & zijun.liu & donglin.wang }@ia.ac.cn

Abstract—**With the development of semiconductor industry and integrated circuits, the performance of processors has been advanced steadily. More and more devices including cores, memories and peripherals are being integrated in chips to meet the requirements of high performance applications. The rapid increase in chip complexity makes it difficult for these devices to work efficiently. In order to facilitate efficient chips systems, we proposed a task scheduling algorithm for Chip Multi-Processors (CMP) which are called Homogeneous Earliest-Finish-Time (HoEFT) algorithm. We use this algorithm to finish two benchmarks on a chip system consisting of eight Processing Elements (PEs) and a 16MB shared memory. The results show that these PEs can reach reasonable utilization under HoEFT algorithm.**

Keywords-task scheduling; multi-core; shared memory; traffic-aware; memory-aware

I. INTRODUCTION

Rapid advancements in semiconductor technology over the past few decades make the computing systems larger and faster. By integrating more and more processors in computing systems, researchers can handle many high-performance applications. For example, a distributed computing system is a group of processors connected via a high speed network, which supports the execution of parallel applications. However, as the computing systems and applications become more and more complex, it is difficult for the systems to execute efficiently and achieve high performance. Task scheduling aims to allocate the tasks of an application to the set of available processors and arrange the execution of the tasks on each processor to minimize the total execution time [1]. The efficient scheduling of an application on the available resources is one of the key factors for achieving high performance [2]. Over the last several decades, considerable researches have been conducted for tasks scheduling and it has been proved that the task scheduling problem is an NP-complete problem in most cases [3].

In general, task scheduling is presented in two forms: static and dynamic [4]. In static scheduling algorithms, all information needed for scheduling must be known in advance while in dynamic scheduling, tasks are allocated to processors upon their arrival time and scheduling decisions are made during run time. In this paper, our main focus is on static scheduling. In addition, task scheduling algorithms can be classified into a variety of categories such as list-scheduling algorithms, duplication-based algorithm and guided random search methods. Though these algorithms have been proved with good results, they are usually not memory-aware or traffic-aware. Thus theses algorithms cannot be used for chip systems directly.

Like integrating many processors in a computing system, multi-core chip systems integrate many cores in one chip to improve the performance of each processor. Thus we consider that the tasks scheduling algorithms for computing systems apply equally for multi-core chip systems after improving. As the designs of multi-core chip systems become more and more application-specific, they derives two main chip systems: Chip Multi-Processors (CMPs) and Multi-Processors System on-Chip (MPSoC). CMPs usually consist of many homogenous cores and they are used for high-performance computing and cloud computing in workstations or servers. MPSoCs are generally used for real-time computing such as stream computing, communication processing and multimedia processing.

In order to improve the performance and utilization of CMPs, we proposed a task scheduling algorithm. Based on the original task scheduling algorithms for computing systems, we add traffic-aware and memory-aware patterns as they are important parts in chip systems. Like previous works, we represent applications by a directed acyclic graph (DAG) in which nodes represent application tasks and edges represent internal task data dependencies. We introduce traffic status to represent the ports of memory banks and cores, and memory status to represent the shared memory and cores' memory. Then we search the minimum time of tasks execution based on a certain scheduling order. In experiment, we use our algorithm to complete LU-decomposition and Double-precision General Matrix Multiplication (DGEMM) on an eight-PE chip system. Results show that the PEs can reach 16.8% and 95% utilization rate.

Following the above discussion, the remainder of this paper is organized as flows. We start by introducing related work in the next Section II and HoEFT algorithm in Section III. In section IV, we show the experimental results. Finally, section V is a conclusion.

II. Related Work

Static task scheduling algorithm can be roughly classified into two main groups: heuristic-based algorithms and guided random-search-based algorithms [2]. Heuristic-based algorithms can be further classified into three groups: list scheduling algorithms, clustering algorithms and task duplication algorithms. List scheduling algorithms maintain a list of all tasks of a given graph according to their priorities. It has two phase: task prioritizing phase for selecting the highest-priority ready task and processor selection phase for selecting a suitable processor that minimizes a predefined cost function. Different from the limited number of processors in list scheduling algorithms, tasks in clustering algorithms are mapped to an unlimited number of clusters and the selected tasks for clustering can be any tasks. Every iteration refines the previous clustering by merging some clusters. If two tasks are mapped to the same cluster, they will be executed on the same processor. Clustering algorithms require additional steps to generate a final schedule: a cluster merging step for merging the clusters so that the remaining number of clusters equals the number of processors, a cluster mapping step for mapping the clusters on the available processors, and a task ordering step for ordering the mapped tasks within each processor [5]. Task duplication algorithms complete tasks scheduling by mapping some of tasks redundantly, which reduces the inter-process communication overhead. Guided random-search-based algorithms mainly include scheduling algorithms based on genetic algorithms, simulated annealing algorithms and local search technique. In this paper, we focus on list scheduling algorithms and other traffic-aware and memory-aware algorithms.

Dynamic Critical-Path (DCP) scheduling algorithm [6] defines that every task has dynamic priority and every task except the entry task and exit task can be moved on the scheduled processors. In addition, when the step is in processor selection phase, DCP considers schedule the highest-priority successor task of the current task on this processor. DCP uses Absolute Earliest Start Time (AEST) and Absolute Latest Start Time (ALST) which can also be considered as lower bound and upper bound to determine the priority of each task. And it just selects the processors which contain the predecessor-tasks or do not contain any tasks. That means the processors should be homogenous otherwise DCP should search all the processors. DCP also contains an insert algorithm which is used to find the best processor and insert the current task between scheduled tasks. After inserting the current task, DCP will update the value of AEST and ALST of each tasks until the last task. Thus DCP constructs a dynamic priority list.

Heterogeneous Earliest Finish Time (HEFT) scheduling algorithm [2] is another typical list-scheduling algorithm. It uses a static priority list based on Upward Rank (URank). HEFT introduces two attributes: Earliest execution Start Time (EST) and Earliest execution Finish Time (EFT). When selecting processors, HEFT will search every processor and find the one which can minimize the EFT of the current task. Different from dynamic scheduling, when finding an insert slot, HEFT will not move any other tasks in current processor and will not check its highest-priority successor task. In addition, the URank value will not be update after each scheduling step in HEFT algorithm. Critical Path on a Processor (CPOP) scheduling algorithm [2] is another list-scheduling using dynamic priority based on HEFT algorithm. It uses the sum of URank and Downward Rank (DRank) to assign the priority of each task. In first, CPOP finds a critical path (CP) which is the highest sum of ranks and a critical-path processor which minimizes the value of the CP. Then they start to schedule the tasks based on the priority list. For the tasks in critical path, they are scheduled on the critical-path processor. For the other tasks, they are scheduled on the processors which can minimize the EFT. After each scheduling step, CPOP will update the priorities of reset tasks until the last task.

Longest Dynamic Critical Path (LDCP) algorithm [7] is a simplifying list-scheduling algorithm which has low complexity. LDCP just uses URank to determine the priority of each task. When selecting processors, it also chooses the one which minimize the finish time of selected task like HEFT. After each scheduling step, LDCP will update rank values of rest tasks. Like HEFT and CPOP, LDCP do not move scheduled tasks either when inserting the current task.

However, the above task scheduling algorithms do not consider the traffic ports and the storage capacity. In [8], authors propose Performance Driven Scheduling (PDS) algorithm, Homogeneous Workload Distribution (HWD) algorithm and Traffic Aware Scheduling (TAS). PDS bases on simulate annealing algorithm and performs the following: random scheduling and mapping of tasks, simulation and capturing, averaging and normalizing performance variables and finally calculate the performance. The other two algorithms performs to add workload and traffic restriction. In [9], authors propose Integer Linear Programming (ILP) algorithm which is a traffic-aware task scheduling algorithm. It contains the following steps: mapping each task to a processor, the scheduling of the tasks on corresponding processor and the priority of each message on an edge. In [10], authors propose a memory-aware task scheduling algorithm which is based on quantum evolutionary algorithm (QEA). Their main focus is energy consumption and they compare message passing and shared-memory communication using QEA.

III. HOEFT Algorithm

A scheduling system in CMPs consists of an application and the hardware architecture of chip systems. The application is represented by a DAG, **G = (V, E)**, where **V** is the set of **v** tasks and **E** is the set of **e** edges. Each edge represents the precedence constraint between two tasks.

We introduce two communication matrixes **UDATA** and **DDATA** to indicate the communication cost between two tasks. **UDATA** describes the communication from predecessor tasks to successor tasks while **DDATA** describes the communication from successor tasks to predecessor tasks. That means UDATA and DDATA are transposed each other.

We assume that there is a **p** ports shared memory which means **p** memory banks and a set **Q** of **q** homogeneous PEs in the target CMP. Each of the PEs and each of the memory bank has only one port and the PEs and shared memory are fully connected in the target system. In addition, the port can execute load and store at same time. The data communications may suffer from contentions and we use first-come-first-serve strategy to deal with these contentions. We introduce **PORTIN** and **PORTOUT** traffic matrixes to indicate the statues of PEs' and shared memory's ports. For every row vector in **PORTIN** or **PORTOUT**, it describe the idle time and usage time of its port.

We introduce memory matrixes **MEMORY (p + q)** to indicate the data storage by PEs and memory banks. **MEMORY** can also indicate the PEs or memory banks are full or have free space. As we assume there are only Scratch Pad Memories (SPMs) in the target system, we introduce indexes to determine which data can be overwritten when the memory is full. For the memory of PEs, we define it will be overwritten automatically when the memory is full. For the shared memory, we define the data can be overwritten if its index is above a certain value which is set by the type of the target application. When the status of shared memory is full and the index is lower than the certain value, the **PORTIN** of this memory bank cannot be used.

We introduce computation vendor **EXEC** to indicate the computation time of each task on PEs as our target system is homogenous. In addition, we introduce status matrix **AVAIL** to indicate the idle time and execution time for every PE.

Like HEFT algorithm, we also define the EST and Actual Finish Time (AFT) attributes which are derived from a given partial schedule. $EST(n_i, p_j)$ is the earliest execution start time of task n_i on PE p_j. For the entry task which has no predecessor task, the value of its EST is 0. For the other tasks in the graph, the $EST(n_i, p_j)$ values are computed recursively and starting from the entry task, as shown in (1).

$$\max\{ready[j], \max(AFT(n_m)+r(PE(n_m),p_j)(2c_{m,i}+w_m)\} \quad (1)$$

where n_m is one of the set of immediate predecessor tasks about task n_i and $ready[j]$ is the earliest time at which p_j is ready for task execution. The inner max block in (1) means the time when all data needed by n_i has arrived at p_j. AFT is determined when the task has been scheduled and its value is the EST add the computation time. $PE(n_m)$ is the PE which task n_m is scheduled on, and $r(PE(n_m),p_j)=1$ if $PE(n_m)\neq p_j$ and zero otherwise. As we use shared-memory in target system, we should double the communication time $c_{m,i}$ which means the communication time from task n_m to task n_i. As we add traffic-aware and memory-aware mechanism, we define the wait time as w_m which means the time should be waited before the data from task n_i can be sent. The calculation of w_m is described in PE Selection Phase which should update the statues of **PORTIN**, **PORTOUT** and **MEMORY** temporarily.

Based on previous insertion-based scheduling policy in [6], we add some restrictions on it. First, we define the scheduled tasks on one PEs cannot be moved. That means if it cannot find a slot on all PEs, the current task should be scheduled at last on one processor. Second, when finding a slot on one PE, it should check the traffic status and memory status. When both the traffic and memory can be available, the current task can be inserted on this PE.

Like most list scheduling algorithms, tasks in HoEFT are also ordered by scheduling priorities that are based on URank. The URank of a task n_i is defined by (2).

$$URank(n_i) = ex_i+\max(c_{i,j}+URank(n_j)) \quad (2)$$

where ex_i is the execution time of task n_i, n_j is one of the set of successor tasks about tasks n_i and $c_{i,j}$ is the communication time between task n_i and task n_j. The communication time is the value from **UDATA** which do not consider whether the two tasks are scheduled on the same PE. For the exit tasks, the values of URank are the computation times. Compared with other algorithms which use two ranks, we just use one to reduce the complexity as our focus is traffic-aware and memory-aware.

```
1. Set the computation vendor EXEC and communication matrix UDATA and DDATA with mean values.
2. Compute URank values of all tasks starting from the exit tasks.
3. List the tasks by decreasing order of URank values.
4. While there are unscheduled tasks in the list, do
5.     Select the current task from the list.
6.     Compute the EST value and update PORTIN, PORTOUT and MEMORY temporarily.
7.     Schedule the current task to the PE which minimizes EST value.
8.     Update matrixes including PORTIN, PORTOUT, MEMORY and AVAIL and update AFT values for scheduled tasks.
9. Endwhile
```

Figure 1. The HoEFT algorithm

Figure 1 shows the HoEFT algorithm for a bounded number of homogenous PEs. It also contains two major phases: task prioritizing phase and PE selection phase. As we want to research more details about tasks scheduling in CMPs, we add update phase which is used to restrict the rest of scheduling steps.

A. *Task Prioritizing Phase*

This phase is completed according to URank values shown in (3). We first compute the URank values of exit tasks which are equal to their computation times. Then we can compute the URank values of these exit tasks' predecessor tasks. Based on the DAG, we finish computing the URank values of all tasks in turn. At last, we prioritize the tasks based on URank values. The entry tasks have the highest priorities and the exit tasks have the lowest priorities. For those tasks which have the same URank value, we define the one which has more successor tasks has higher priority. If they still have the same number of successor tasks, we randomly prioritize them.

B. *PE Selection Phase*

From the **AVAIL** we can get the earliest available time of a PE. This time may not be the final value that a task can be executed as we can use the insertion-based algorithm to find a slot between the scheduled tasks. Although we can find a slot in some PEs or we get the time from the **AVAIL**, we still do not define the time as the maximum value between them. We take the communication time into account if the current task has predecessor tasks and the communication time should be doubled if the current task and its predecessor task are not on the same PE. Furthermore, we add wait time to allow the data needed by the current task to be ready before the tasks can be executed. The wait time is mainly caused by the contention in traffic and the limitation of the memory's capacity. We must wait the port of the destination memory and the port of current PE to be idle, and wait the memory bank until it can be written if it is full. No matter calculating the communication time or the wait time, we will update **PORTIN**, **PORTOUT** and **MEMORY** until all the predecessor tasks have been checked. When selecting PE$_j$, we get the minimum value from the results of **AVAIL** and insertion-based algorithm which is defined as ready[j]. Then we calculate the communication time and wait time as the above and add them together with the AFT values of the current task's predecessor tasks. After checking all the predecessor tasks, we choose the maximum sum. Then we select the maximum value as the EST from the sum and the ready[j]. After all the PEs have been checked, we select the PE which minimizes the EST.

C. Update Phase

After selecting the task and the PE, we should update the status of **PORTIN**, **PORTOUR**, **MEMORY** and **AVAIL**. It should be noted that the status of **MEMORY** should not be updated to the status which the current task has sent out its data. As the data needed by next selected task should not be sent before next update phase and can just be sent temporarily in next PE selection phase.

IV. Experimental Results and Discussion

We assume that our chip system contains eight PEs, a shared-memory and two Double Date Rate (DDR) controllers. Each PE consists of four 512-bit Floating-point Multiply-Add (FMA) units and 1 MB ping-pang memory. The shared memory contains 16 banks each of which is also 1 MB ping-pang memory. We assume that the Synchronous Dynamic Random Access Memory (SDRAM) can run at 2800 MHz with 64-bit data width and we don't care about the capacity of SDRAM. The PEs run at 1.4 GHz, and the shared-memory and DDRs run at 0.7 GHz. We assume that there is a 128-bit crossbar to connect them which also runs at 0.7 GHz. The one of the DDRs is used to store the original data and the other is used to store temporary data.

As LINPACK is now the most popular benchmark in the world for testing floating-point performance and it mainly consists of LU-decomposition and DGEMM, we test them separately on our chip system under the HoEFT algorithm. Our chip system is based on shared memory, so it will consume long time to write back the data after computing. We assume that each PE manages some fixed data and just change two vectors after each compare operation. Thus near the end of LU-decomposition, some PEs are idle. The other scheduling steps in LU-decomposition and the scheduling steps in DGEMM are as our algorithm.

Results show that in LU-decomposition, PEs can reach about 126.9 Gflops while in DGEMM they can reach 716.8 Gflops. Taking account of 5% error, we consider that PEs can reach 16.8% utilization rate in LU-decomposition and 95% utilization rate in DGEMM. As our algorithm contains more limitation, the utilization rate is lower than the results in distributed computing systems. Thus it can reflect the performance of chip systems more exactly.

V. Conclusion

In this paper, we proposed a traffic-aware and memory-aware task scheduling algorithm which can be used for task scheduling in CMPs. Based on previous list scheduling algorithms, we add traffic-status matrix and memory-status matrix which can be used to indicate more details in chip systems. We use this algorithm to finish two benchmarks in an eight-core systems. Results show that our algorithm can make the PEs run at a reasonable utilization rate.

In the future work, we consider that we can use this algorithm to finish more tasks scheduling in different chip systems. Then we can find the bottlenecks of hardware design and improve them.

References

[1] Daoud M I, Kharma N. A hybrid heuristic–genetic algorithm for task scheduling in heterogeneous processor networks[J]. Journal of Parallel & Distributed Computing, 2011, 71(11):1518-1531.

[2] Topcuoglu H, Hariri S, Wu M Y. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing[J]. IEEE Transactions on Parallel & Distributed Systems, 2002, 13(3):260-274.

[3] Bajaj R, Agrawal D P. Improving scheduling of tasks in a heterogeneous environment[J]. Parallel & Distributed Systems IEEE Transactions on, 2004, 15(2):107-118.

[4] Bansal S, Kumar P, Singh K. Dealing with heterogeneity through limited duplication for scheduling precedence constrained task graphs[J]. Journal of Parallel & Distributed Computing, 2005, 65(4):479-491.

[5] Liou J C, Palis M A. A comparison of general approaches to multiprocessor scheduling[C]// Parallel Processing Symposium, 1997. Proceedings. International. IEEE, 1997:152-156.

[6] Kwok Y K, Ahmad I. Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors[J]. IEEE Trans Parallel & Distributed Systems, 1996, 7(5):506-521.

[7] M.I. Daoud, N. Kharma, A high performance algorithm for static task scheduling in heterogeneous distributed computing systems, J. Parallel Distrib.Comput. 68 (2008) 399–409.

[8] Tafesse B, Raina A, Suseela J, et al. Efficient Scheduling Algorithms for MpSoC Systems[C]// Eighth International Conference on Information Technology: New Generations. IEEE, 2011:683-688.

[9] Yang L, Liu W, Jiang W, et al. Traffic-Aware Application Mapping for Network-on-Chip Based Multiprocessor System-on-Chip[C]// IEEE, International Conference on High PERFORMANCE Computing and Communications. IEEE, 2015:571-576.

[10] Lee J, Choi K. Memory-aware mapping and scheduling of tasks and communications on many-core SoC[C]// Design Automation Conference. IEEE, 2012:419-424.