International Journal of Reconfigurable and Embedded Systems (IJRES)
Vol. 10, No. 3, November 2021, pp. 157~167
ISSN: 2089-4864, DOI: 10.11591/ijres.v10.i3.pp157-167

Heuristic algorithms for dynamic scheduling of moldable tasks in multicore embedded systems

Takuma Hikida[1], Hiroki Nishikawa[2], Hiroyuki Tomiyama[3]
[1,2,3]Graduate School of Science and Engineering, Ritsumeikan University, Japan
[2]Research Fellow, Japan Society for the Promotion of Science, Japan

Article Info

Article history:

Received May 27, 2021
Revised Jun 18, 2021
Accepted Aug 2, 2021

Keywords:

Dynamic scheduling
Fixed number
Heuristic algorithm
Multicore
Parallel computing

ABSTRACT

Dynamic scheduling of parallel tasks is one of the efficient techniques to achieve high performance in multicore systems. Most existing algorithms for dynamic task scheduling assume that a task runs on one of the multiple cores or a fixed number of cores. Existing researches on dynamic task scheduling methods have evaluated their methods in different experimental environments and models. In this paper, the dynamic task scheduling methods are systematically rearranged and evaluated.

Corresponding Author:

Takuma Hikida
Graduate School of Science and Engineering
Ritsumeikan University
1-1-1 Nojihigashi, Kusatsu, Shiga, 525-8577, Japan
Email: takuma.hikida@tomiyama-lab.org

1. INTRODUCTION

Recent embedded systems have been equipped with a multicore system-on-a-chip to achieve high performance and power efficiency. In order to fully exploit the potential parallelism in tasks and multicore architectures, task scheduling is one of the most important techniques. Classical task scheduling techniques exploit task-level parallelism by assigning multiple tasks to different cores concurrently, but they assume that each task runs on a single core in a single-threaded manner. Modern techniques allow a multi-threaded task to run on multiple cores, which takes into account both task- and data-parallelisms.

Task scheduling can be classified from a variety of perspectives. One classification is whether cores are homogeneous or heterogeneous, and this work assumes homogeneous cores. Another classification is when a scheduling decision is made. If scheduling is decided at the design time and is unchanged at runtime, such scheduling is called static scheduling. Static scheduling is suited to small-scale embedded systems where release times (a.k.a. arrival times or activation times) of the tasks are known a priori. If a scheduling decision is made at runtime, such scheduling is called dynamic or online. Since the functionality of embedded systems is continually becoming more complex and dynamic, dynamic task scheduling has become more important than ever, and this paper addresses dynamic task scheduling.

Task scheduling can be classified based on the parallelism inside the tasks. Classical task scheduling assumes that tasks are not parallel. Each task is assumed to be single-threaded and run on a single core. Recently, parallel tasks (multi-threaded tasks) have become popular not only for supercomputing but also for embedded computing due to the increasing number of cores in embedded systems. Parallel tasks are further classified into gang tasks and fork-join tasks. Threads in a gang task are executed synchronously. All of the threads get started and completed at the same time. On the other hand, threads in a fork-join task are executed independently of one another. The timing behavior of gang tasks is more predictable than that of fork-join tasks, and therefore, gang tasks are suitable for real-time embedded systems. This paper assumes gang tasks. From another perspective, parallel tasks are classified into three types, i.e., rigid tasks, moldable tasks and malleable tasks, based on when the parallelism (the number of cores) is decided. The number of cores assigned to a rigid task, a moldable task and a malleable task is decided before task scheduling, during task scheduling and at runtime, respectively. This paper assumes moldable gang tasks.

Task scheduling is a kind of classical problem in the field of computer science and has extensively been studied for several decades. Very early work on multicore task scheduling assumes that tasks are single-threaded. Early algorithms try to execute as many tasks as possible simultaneously on different cores by exploiting inter-task parallelism. Drozdowski [1] provided an exhaustive survey of classic task scheduling methods in multicore architectures. Scheduling of tasks on multiple cores by executing multiple independent tasks on multiple cores in parallel. A list scheduling algorithm was proposed in which specific priorities are calculated and assigned to each task [2]-[5]. The list scheduling algorithm assigns the highest priority task to the free core until all tasks are scheduled. List scheduling is generally accepted as an attractive approach because it can combine low complexity and good results. Both approaches assume that each task is executed on a single core without consideration of data parallelism. The works in [1]-[5] are classified as static scheduling. It is not possible to make classifications related to parallelism tasks.

A list-based scheduling algorithm for data-parallel tasks was proposed [6]. Their work assumes that a set of dependent tasks is given in the form of a task graph and each task is assigned to a fixed number of cores to minimize the overall length of the schedule. Their scheduling is classified as static scheduling for rigid gang tasks. Yang and Ha [7], and Vydyanathan *et al.* [8] they dealt with static scheduling of gang and moldable tasks. Recent publications include [9] and [10] for moldable gang tasks. Shimada *et al.* [9] proposed ILP-based scheduling. The authors of [10] proposed a scheduling approach based on constraint programming. Researchers dealt with static scheduling of gang and malleable tasks [11]-[14]. An algorithm to determine the optimal solution for task allocation in a heterogeneous environment was proposed [15]-[18]. It should be noted that the scheduling approaches presented in [1]-[18] are not dynamic but static.

Ye *et al.* [19] attempted to develop an online scheduling model for moldable tasks. Yang *et al.* [20] developed a new algorithm for dynamic task scheduling in a heterogeneous multiprocessor system. The authors of [21]-[23] presented a new scheduling algorithm for heterogeneous distributed computing systems.

Dynamic scheduling has also been applied in the field of HPC. Ramezani [24] proposed CPA, a heuristic-based dynamic-critical path-aware scheduling method for scheduling task graphs on multi-FPGA systems, against the background of being plagued with the problem of computing resources to run HPC applications. Priya and Sahana, [25] implemented first come first served (FCFS) in HPC using message passing application programmer interface (MPI).

This paper studies dynamic scheduling of moldable gang tasks, where tasks' arrival time or period is unknown in advance. In this paper, dynamic task scheduling methods are systematically rearranged and proposed, and evaluate them in a common environment. Contributions of this paper are: i) nine scheduling algorithms based on task selection and parallelism decision are systematically proposed, and ii) the nine algorithms are quantitatively evaluated, and demonstrate that two algorithms are more effective than the other seven.

2. PROBLEM DEFINITION

In the scheduling problems that are tackled in this paper, a set of tasks is given, where the execution time of each task is known but the arrival time or period are unknown in advance, and the objective is to minimize the schedule length (a.k.a., makespan). The exectuon time of each task is dependant of the degree of paralellism. Table 1 represents an example of the executon time for the tasks in a task-graph. In the following example, the number of cores in the target system is assumed four, and the parallelism of each task is assumed to range from one to four. An increase in the number of cores used does not necessarily lead to a decrease in execution time. For example, in Table 1, the execution time of task 1 does not change even if two or more cores are allocated. In this case, assuming that the upper parallelism limit for task 1 is 2, the number of cores that exceed the upper parallelism limit for each task will not be allocated.

Task 1 takes 30 time-units when executed on one core. It takes 20 time-units if the task is executed on two cores. If it executed on three or four cores, the execution time is no shorter than 20 time-units. The reason why the execution times on three cores and four cores do not change is that the performance for task 1 can hardly be improved even if more cores are assigned to task 1 than two since the overhead for parallelization becomes large. For this reason, assignments of two cores to task 1 is the most effective selection in terms of performance and resources.

Table 1. Execution times of eight moldable tasks

	Number of cores				Release
	1	2	3	4	Time
Task 1	30	20	20	20	0
Task 2	30	30	30	30	0
Task 3	70	40	30	25	20
Task 4	30	25	20	15	50
Task 5	70	40	40	40	50
Task 6	90	50	45	45	50
Task 7	20	15	10	10	50
Task 8	10	10	10	10	50

Recall that this paper assumes dynamic scheduling of tasks where the tasks arrival time or the period is unknown beforehand. However, for comprehension, the example is shown with release time for each task. Task 1 and task 2 are assumed to be arrived at time 0 and become ready to run. Task 3 is assumed to be arrived at time 20.

Figure 1 shows one of scheduling results that the tasks in Table 1 are mapped on the four cores. At time 0, there are task 1 and task 2 that are arrived and become ready for the execution. The performance of task 1 can be improved if two cores are assigned. On the other hand, the performance of task 2 cannot be improved even if the cores are assigned. Therefore, two cores are assigned to task 1 and a single core is assigned to task 2, respectively. When task 3 becomes available, core 0 and core 1 are released and will be executed by 3 cores, including core 3. Not all cores are necessarily to run. In addition, at this point in time, executing task 3 with 4 cores after task 2 is completed can minimize the execution time for the 3 tasks, but because scheduling is based on the assumption that the task will be released at any time, task 3 execution do not wait for task 2 to complete execution.

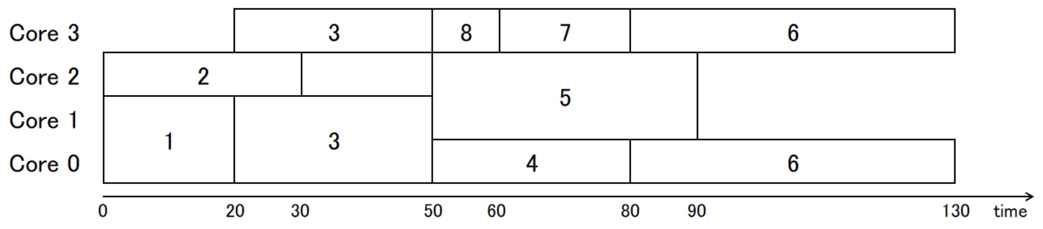


Figure 1. An example of a scheduling result

At time 50, tasks that can be executed is 4, 5, 6, 7, and 8. Although task 4 has the upper parallelism limit of 4, and task 5 also has the upper parallelism limit of 2. It can be considered that task 5 has less workload in parallelization. For this reason, a policy of distributing more cores to task 5 than to task 4 is adopted here. In this way, the way the distribution rules are defined affects the schedule length.

Rather than allocate another core to task 4, task 8 is allocated, which can be completed immediately by one core. Once task 8 is completed, task 7 is executed on that core. When task 4 is completed, task 6 is executed on the core. As a result, at the time when task 5 completes execution, task 6, which has the longest execution time on one of these cores and has the greatest benefit when executed on two cores, could be executed on two cores. In this way, which tasks are assigned to which cores in order of priority and by what rules also affect the schedule length.

3. THE PROPOSED HEURISTIC ALGORITHMS

As seen in the previous section, the moldable task scheduling problem can be split into two sub-problems: task selection and parallelism decision. In this paper, tasktask selection includes FCFS, SJF (Shortest Job First), and Ovh. In the FCFS policy, tasks are allocated in priority order of task arrival. In the SJF policy, tasks are allocated in priority order of burst duration of tasks on a core. In the Ovh policy, tasks are allocated in priority of small parallelization overhead. The Ovh policy reduces the increase in total workload when tasks are parallelized. Parallelism decision includes Single, Max, Fit, and Fair. In the Single policy, each task is allocated to a single core. In the Max policy, each task is allocated to a maximum number of cores. In the Fit policy, task selection priority task is allocated to maximum core. In the Fair policy, each task are given fair number of cores. To describe the details of the scheduling policies, another example is provided with execution time of each task as shown in Table 2.

Table 2. Execution time of tasks dependent of the number of cores

	Number of cores				Release
	1	2	3	4	Time
Task 1	40	25	15	15	0
Task 2	35	20	15	10	0
Task 3	45	25	25	25	10
Task 4	40	30	20	15	15
Task 5	55	40	20	20	20
Task 6	80	50	40	30	40
Task 7	70	40	30	25	50
Task 8	100	60	45	30	60

3.1. Single-FCFS scheduling

Single-FCFS scheduling is a simple technique that a task is executed on a single core in the FCFS policy. Therefore, the parallelism for each task is fixed as one and the tasks are executed in the order that they arrive in the queue. In Table 2, for example, the execution times of task 1 and task2 are 40 and 35 time-units due to the single-threaded fashion, respectively, and task 1 and task 2 are arrived at time 0 so that task 1 and task 2 are ready to be executed in the arriving order on a single core. Figure 2 represents gantt chart to show one of schedling results for the tasks in Table 2. The x-axis of the gantt chart shows time and the y-axis shows the cores. In the figure, each task is assigned to a single core from time 0, and the overall schedule length is represented as 155, when task 8 finished runing.

In general , the single-threaded execution minimizes the parallelization overhead. The performance cannot be scaled lineary with the number of cores since there occurs the parallelization overhead. For instance, the execution time of task 1 on dual cores represents 25 time-units for a thread. Therefore, the dual threads consume 50 time-units in total, which is longer than 40 time-units on a single. On the other hand, the single thread occupies a core until its end so that the long execution of a task may degrade the overall scheduling effectiveness. In the example of Figure 2, the execution of task 8 results in the performance degradation since the task runs for a long time.

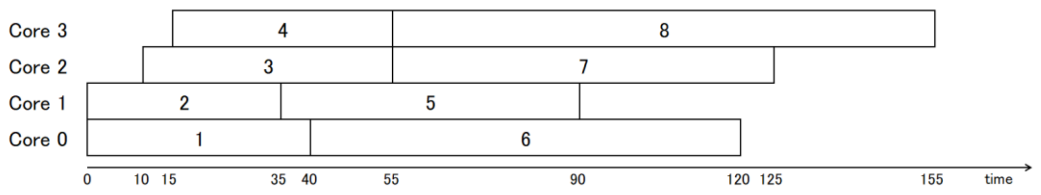


Figure 2. An example of Single-FCFS scheduling

3.2. Single-SJF scheduling

Single-SJF scheduling is a simple technique that a task is executed on a single core in the SJF policy. Therefore, the parallelism for each task is fixed as one and the tasks are executed in the rearranged queue that is sorted in the order of burst duration of tasks executed on a single core. In Figure 3, for example, the execution times of task 1 and task 2 are 40 and 35 time-units due to the single-threaded fashion, respectively. Since the execution time of task 2 is shorter than that of task 1, task 2 is allocated on Core 0 and task 1 is allocated on Core 1. In the example, the schedule lengths by Single-FCFS and Single-SJF are shown as 155, which is the same result.

3.3. Max-FCFS scheduling

Max-FCFS scheduling is a technique that a task is executed on a maximum number of cores in the FCFS policy. Therefore, the parallelism for each task is fixed as maximum and the tasks are executed in the order that they arrive in the queue. In this paper, the maximum degree of parallelism for each task is asssumed to set the number of cores on a target system. However, as shown in Table 3, if the execution time does not decrease as the number of cores increases, the smallest number of cores is regarded as the maximum parallelism. For example, the maximum parallelism of Task 3 is represented as two. In Table 3, for example, the execution times of task 1 and task 2 are 15 and 10 time-units in the Max policy, respectively. As the scheduling result, task 1 is executed on three cores, and task 2 is executed on four cores. The Max policy leads the maximum overhead for parallelization. In Figure 4, the schedule length by Max-FCFS is 170, which is longer than the Single policy.

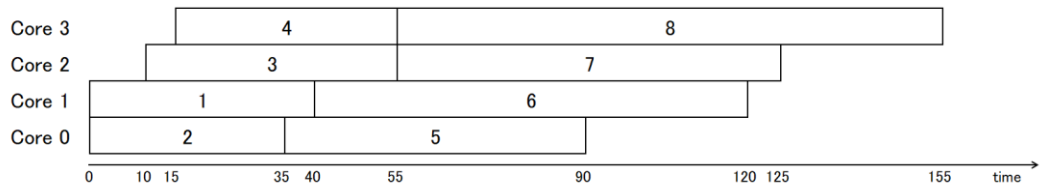


Figure 3. An example of Single-SJF scheduling

Table 3. Remarking the maximum degree of parallelism of Table 2

	Number of cores				Release
	1	2	3	4	Time
Task 1	40	25	15	15	0
Task 2	35	20	15	10	0
Task 3	45	25	25	25	10
Task 4	40	30	20	15	15
Task 5	55	40	20	20	20
Task 6	80	50	40	30	40
Task 7	70	40	30	25	50
Task 8	100	60	45	30	60

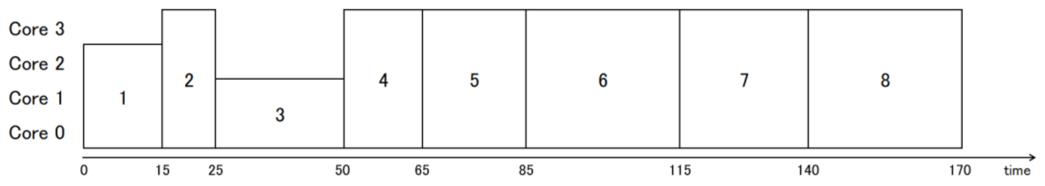


Figure 4. An example of Max-FCFS scheduling

3.4. Max-SJF scheduling

Max-SJF scheduling is a technique that a task is executed on a maximum number of cores in the SJF policy. Figure 5 shows the Max-SJF scheduling. At time 0, task 2 and task 1 are arrived in the order, and at time 25, task 3, 4, and 5 are arrived to ready, and the queue is sorted in the order of task 4, 3, and 5 in the SJF policy. At time 40, task 6 is arrived in the queue, and the queue sorted in the SJF policy is in the order of task 3, 5, and 6 since the execution time of task 6 is longer than those of task 3 and task 5. At time 65, task 7 and 8 are arrived and the queue is rearranged in the order of task 5, 7, 6, and 8, which is finally determined for scheduling.

3.5. Fit-FCFS scheduling

Fit policy is similar to the Max policy in that the Fit policy tries to execute tasks with their maximum parallelism. If the number of available cores is smaller than the maximum parallelism of the task, all the available cores are assigned to the task. Figure 6 shows the result of Fit-FCFS scheduling.

According to Table 3, task 1 and task 2 are arrived and available in the order at time 0. task 1 is executed on three cores at the maximum paralellism, and assigned to the cores. For task 2, there is no choice but to assign the rest of core, Core 3, at the time. At time 15, when Task 1 completes the execution, three cores are released and become available. The tasks that can be executed at the time are task 3, 4, and 5. Due to the FCFS policy, task 3 has the highest priority and is assigned to two cores. The remaining core executes the task 4, and task 5 waits until the cores are available. At time 35, when Task 2 completes the execution, the single core is assigned to task 5. At time 40, when Task 3 completes execution and task 6 is arrived, task 6 is assigned to two cores. At time 50, when task 4 completes the execution, task 7 and task 8 can be executed. Task 7 is available earlier thant task 8 in the FCFS policy; therefore, task 7 is assgined to the single core. taskAt time 90, when task 5 and ask 6 complete execution at the same time, 3 cores are allocated to task 8. The schedule length is represented as 135 as a result.

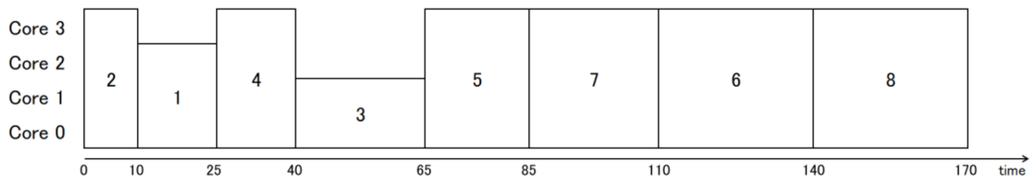


Figure 5. An example of Max-SJF scheduling

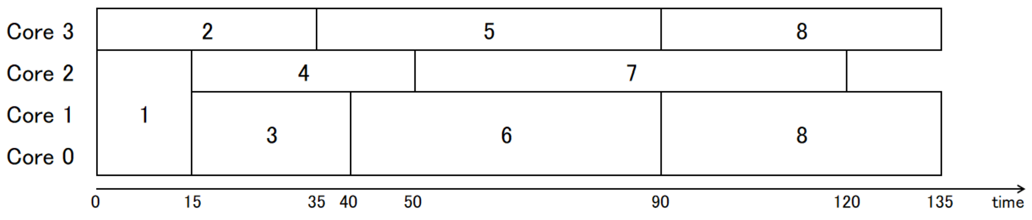


Figure 6. An example of Fit-FCFS scheduling

3.6. Fit-SJF scheduling

This technique is based on the Fit policy and the SJF policy, which are already presented in the paper. The queue is sorted in the order of shortest-job-first for task selection and the parallelism is determined through the Fit policy, which attemts to assigne as many cores as possible within the maximum degree of parallelism for tasks. Figure 7 shows a Fit-SJF scheduling result.

At time 0, both of task 1 and task 2 can be executed, but since task 2 has a shorter execution time on a single core than task 1, task 2 is assigned to four cores in the Fit policy due to the maximum parallelism in accordance with Table 3. At time 10, when task 2 completes the execution, task 1 and task 3 can be executed. task 1 is the predecessor task to task 3 since the shorter task than task 3, and 3 cores are assigned to task 1. The remaining core is to task 3. At time 25, when task 1 completes the execution, task 4 can be executed.

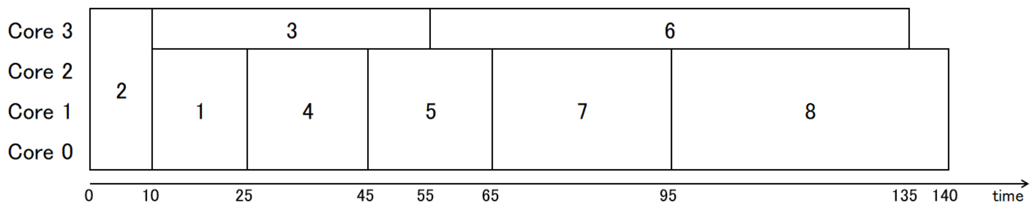


Figure 7. An example of Fit-SJF scheduling

3.7. Fair-FCFS scheduling

Fair tries to assign cores equally to the tasks in the queue. For example, if there exist two tasks in the queue and two cores are available, the Fair policy assigns one core to each task. If there exist three tasks in the queue and two cores are available, the first and second tasks in the queue are selected based on the FCFS.

Similarly, if there exist two tasks in the queue and three cores are available, the first task in the queue is assigned two cores, and the second task is assigned a core. Figure 8 shows a Fair-FCFS scheduling result.

At time 0, two cores out of four cores are assigned to each of task 1 and task 2, respectively in such a way that the cores are evenly distributed. At time 20, when task 2 completes the execution, task 3, 4, and 5 are arrived in the queue and become available. Due to the FCFS policy, task 3 and 4 are executable in this order. Thus, task 3 and task 4 are mapped on the remaining two cores so that a single core is assigned to each of them. At time 25, when task 1 completes the execution, task 5 is determined to run on the two cores since task 5 is alone in the queue at the time. At time 60, task 6 is mapped on a single core.. At time 65, 3 cores are released, and the tasks that can be executed are task 7 and task 8 in the order. Thus, task 7 is assigned to a single core, and task 8 is also assigned to a core in accordance with the Fair policy. The rest of the cores is allocated to task 8 due to the FCFS policy. As the result, the schedule length is represented as 165.

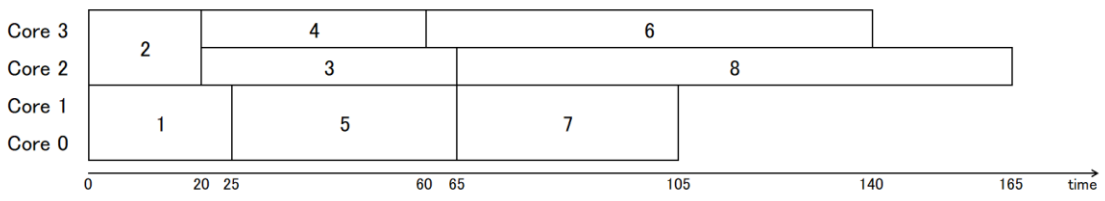


Figure 8. An example of Fair-FCFS scheduling

3.8. Fair-SJF scheduling

The technique is based on Fair and SJF policies. The detail of this technique is described with a scheduling example of Figure 9. At time 0, the queue is rearranged so that task 2 and task 1 are in the order due to the SJF policy. Based on the Fair policy, task 1 and task 2 are assigned to two cores, respectively. At time 20, task 3, 4, and 5 are arrived. As well as the previous assignement, the queue is sorted in the order of task 4, 3, and 5. At the time, the remaining number of cores is two and task 4 and task 3 are assigned to a single core based on the Fair policy, respectively. At time 25, task 5 is assigned to two cores. At time 60, when task 4 completes its execution, task 6, 7, and 8 are ready to run. The queue is rearranged into the order of task 7, 6 and 8 due to the SJF policy, according to Table 3. Thus, task 7 is assigned to a single core. At time 65, task 6 is assigned to two cores and task 8 is assigned to a core. As the result, The overall schedule length is 165 in the figure.

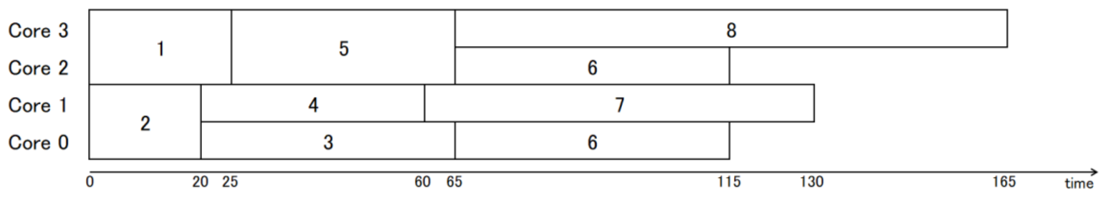


Figure 9. An example of Fair-SJF scheduling

3.9. Fair-Ovh scheduling

In Fair-Ohv scheduling, parallelism decision is based on the Fair policy. On the other hand, the Ovh policy is the task selection technique to assign cores where the task is selected based on the minimum overhead for parallelization of the tasks in the queue. In general, the increased degree of parallelism occurs large overheads for parallelization. Due to that, the efficiency of parallel computation of tasks may degrade so that the execution time can hardly become short in response to its parallelism. In this paper, the Ovh policy that evenly distributes a core to each task and assigne one of remaining cores to the task with minimum overhead is presented. Figure 10 shows an example of Fair-Ovh scheduling.

At time 0, task 1 and task 2 are evenly assigend a core. Then, one of the remaining two cores is assigned to a task with minimum overhead. The overhead can be calculated as follows. If task 1 is assigned two cores, the overhead is 10 ($=2 \times 25 - 40$). On the other hand, the overhead of task 2 is 5 ($=20 \times 2 - 35$). Therefore, the task with the minimum ovehead is determined as task 2, and a remaining core is additionally assigned to task 2. Now, task 1 is assigned a core and task 2 is assigned two cores and a core is still available. Then, if task 2 is assigned three cores, the overhead is 10 (=$15 \times 3 - 35$), according to Table 3. Compared the overhead of task 1 and task 2, task 1 is selected to assign the core. In this case, the overhead of task 1 is equal to that of task 2, but a core is assigned to task 1 likewise following the FCFS policy. Throughout the task selection, task 1 and task 2 are finally assigned two cores, respectively. At time 20, when task 2 completes the execution, task 3, 4, and 5 are arrived in the order. If one of the remaining cores is evenly assigned to each of task 3 and task 4, all the cores become occupied by the tasks. Therefore, additional assignment of remaining cores is not necessary at the time. At time 25, there is only task 5 to be executed in the queue. Thus two cores are assigned to task 5. At time 60, task 6, 7, and 8 are arrived and become ready in the queue, but there is only a core available at the time. In accordance with the Fair policy, task 6 is assigned a core, and any of cores cannot be assigned to tasks. At time 65, task 7 and task 8 can be started on the remaining three cores. A core is evenly assigned to each of them, and the overhead is calculated for additional assignment of the core. The overhead of task 7 is 10 (=$40 \times 2 - 70$). On the other hand, that of task 8 is calculated as 20 (=$60 \times 2 - 100$). Thus, the core is assigned to task 7 at the time, and two out of three cores are finally assigned to task 7 and a core is assigned to task 8, respectively. In this case, the schedule length as 165 is obtained, as shown in the Figure 10.

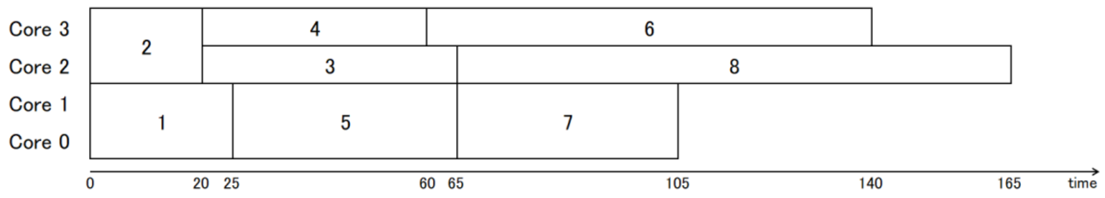


Figure 10. An example of Fair-Ovh scheduling

4. EXPERIMENTS

In the previous section, the scheduling problems that can be split into two section: task selection and parallelism decision are described. In task selection, FCFS, SJF, and Ovh have been presented. On the other hand, in paralallelism decision, Single, Max, Fit, and Fair have been presented. Moreover, nine scheduling techniques in combination with the scheduling policies are proposed.

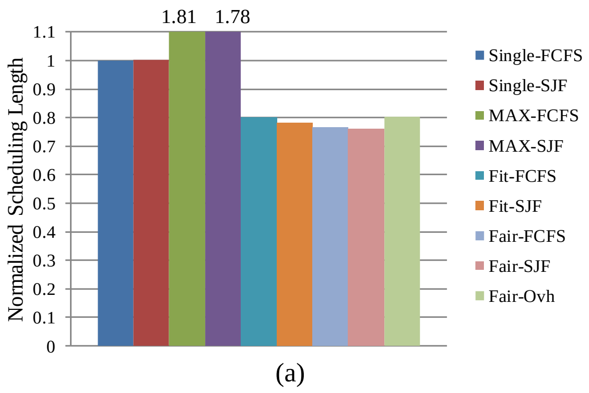
In order to evaluate the effectiveness of the presented scheduling techniques, experiments have been conducted. A metric of the experiments is in general the overall schedule length obtained through scheduling is supposed. In the experiments, a set of tasks with eight scenarios, each of which consists of 500 tasks or 1000 tasks derived from STG benchmark suite is used [26]. The maximum parallelism for each task is not included in the STG benchmark suite, which is therefore randomly generated and fixed in advance. Each task is assumed to have 10% of overhead to parallelize. The number of cores on the target system is set to eight and sixteen. The techniques employed in the experiments are those which are earlier presented in Section 3.

Figure 11(a) and Figure 11(b) shows the experimental results of nine scheduling algorithms with eight scenarios, each which conpose a set of 500 tasks, on 8 cores and 16 cores, respectively. Figure 11(a) shows that Fit and Fair achieve relatively good results, which reduce the schedule lengths by 21.7% on average. On the other hand, the Max policy obviously degrades the performance since tasks, which cannot be executed at its maximum parallelism, are waited until the cores are released. Therefore, the cores are not fully utilized in this policy. On 16 cores, the trend of the results is very similar to that on 8 cores, but the results by Fit and Fair policies are scaled for the better than the results on 8 cores.

Figure 12(a) and Figure 12(b) shows the experimental results on 8 cores and on 16 cores as well as Figure 11, but each of the task sets composes 1000 tasks in Figure 12. All the results of schedule lengths are normalized to the Single-FCFS algorithm and are shown as the average of the normalized schedule lengths for the scenarios. Compared the results among Fair-FCFS, Fair-SJF, and Fair-Ovh, the Ovh policy slightly degrades the performance. Fair-FCFS and Fair-SJF are better than the others, both of which can obtain the schedule lengths 25% shorter than those of Single-FCFS. Fair-Ovh obtains 20% shorter than Single-FCFS. Max-SJF recorded distinctly better results than Max-FCFS.

Figure 12(a) and Figure 12(b) shows the experimental results on 8 cores and on 16 cores as well as Figure 11, but each of the task sets composes 1000 tasks in Figure 12. All the results of schedule lengths are normalized to the Single-FCFS algorithm and are shown as the average of the normalized schedule lengths for the scenarios. Compared the results among Fair-FCFS, Fair-SJF, and Fair-Ovh, the Ovh policy slightly

degrades the performance. Fair-FCFS and Fair-SJF are better than the others, both of which can obtain the schedule lengths 25% shorter than those of Single-FCFS. Fair-Ovh obtains 20% shorter than Single-FCFS. Max-SJF recorded distinctly better results than Max-FCFS.



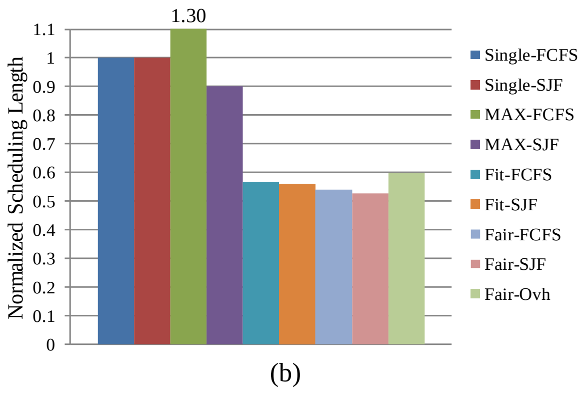
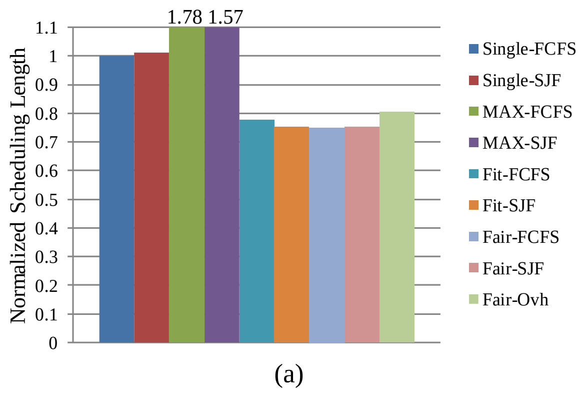


Figure 11. Normalized schedule length for 500 tasks: (a) 8 cores, (b) 16 cores



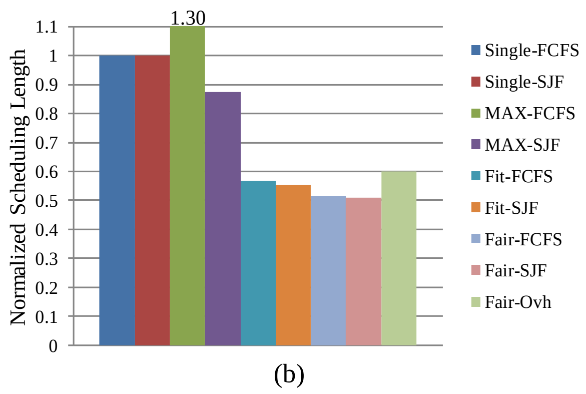


Figure 12. Normalized schedule length for 1000 tasks: (a) 8 cores, (b) 16 cores

5. CONCLUSIONS

The literature on dynamic scheduling algorithms for moldable tasks have been proposed yet have never been evaluated in a same environment. In this paper, dynamic task scheduling algorithms on the certain conditions have been systematically rearranged and evaluated. Dynamic scheduling of moldable tasks can be split into two sub-problems such as task selection and parallelism decision and have presented two policies for task selection and four policies for parallelism decision is assumed. In addition, the Ovh policy dedicated to the Fair task selection have been proposed. In the experimental scenarios, the effectiveness of Fair-FCFS and Fair-SJF derived shorter schedule lengths has been demonstrated. In future, more extensive experiments will be conducted. Also, more sophisticated algorithms will be developed.

ACKNOWLEDGEMENTS

This work is supported partly by KAKENHI 20H00590, 20H04160, and 20J21208.

REFERENCES

[1] M. Drozdowski, "Scheduling multiprocessor tasks: An overview," *European journal of operational research*, vol. 94, no. 22, pp. 215-230, 1996, doi: 10.1016/0377-2217(96)00123-3.
[2] S. Venugopalan and O. Sinnen, "Optimal linear programming solutions for multiprocessor scheduling with communication delays," *International Conference on Algorithms and Architectures for Parallel Processing*, 2012, pp. 129-138, doi: 10.1007/978-3-642-33078-0_10.

[3] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Commun. Ass. Comput. Mach.*, vol. 17, no. 12, pp. 685-690, 1974, doi: 10.1145/361604.361619.

[4] H. El-Rewini, H. Ali and T. Lewis, "Task scheduling in multiprocessing systems," *Computer*, vol. 28, no. 12, pp. 27-37, Dec. 1995, doi: 10.1109/2.476197.

[5] G. Sinevriotis and T. Stouraitis, "A novel list-scheduling algorithm for the low-energy program execution," *2002 IEEE International Symposium on Circuits and Systems. Proceedings (Cat. No.02CH37353)*, 2002, pp. IV-IV, doi: 10.1109/ISCAS.2002.1010398.

[6] Y. Liu, L. Meng, I. Taniguchi and H. Tomiyama, "Novel list scheduling strategies for data parallelism task graphs," *International Journal on Networking and Computing*, vol. 4, no. 2, pp. 279-290, 2014, doi: 10.15803/ijnc.4.2_279.

[7] H. Yang and S. Ha, "Pipelined data parallel task mapping/scheduling technique for MPSoC," *2009 Design, Automation & Test in Europe Conference & Exhibition*, 2009, pp. 69-74, doi: 10.1109/DATE.2009.5090635.

[8] N. Vydyanathan, *et al.*, "An integrated approach to locality-conscious processor allocation and scheduling of mixed-parallel applications," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 8, pp. 1158-1172, Aug. 2009, doi: 10.1109/TPDS.2008.219.

[9] K. Shimada, S. Kitano, I. Taniguchi, and H. Tomiyama, "ILP-based scheduling for parallelizable tasks," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Science*, vol. E100-A, no. 7, pp. 1503-1505, 2017, doi: 10.1587/transfun.E100.A.1503.

[10] H. Nishikawa, K. Shimada, I. Taniguchi and H. Tomiyama, "A constraint programming approach to scheduling of malleable tasks," *International Journal on Networking and Computing*, vol. 9, no. 2, pp. 131-146, 2019, doi: 10.15803/ijnc.9.2_131.

[11] A. Saifullah, K. Agrawal, C. Lu and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," *2011 IEEE 32nd Real-Time Systems Symposium*, 2011, pp. 217-226, doi: 10.1109/RTSS.2011.27.

[12] C. Chen and C. Chu, "A 3.42-Approximation algorithm for scheduling malleable tasks under precedence constraints," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 8, pp. 1479-1488, Aug. 2013, doi: 10.1109/TPDS.2012.258.

[13] K. Jansen and H. Zhang, "Scheduling malleable tasks with precedence constraints," *Journal of Computer and System Sciences*, vol. 78, no 1, pp. 245-259, 2012, doi: 10.1016/j.jcss.2011.04.003.

[14] J. Barbosa, C. Morais, R. Nobrega, and A.P. Monteiro "Static scheduling of dependent parallel tasks on heterogeneous clusters," *2005 IEEE International Conference on Cluster Computing*, 2005, pp. 1-8, doi: 10.1109/CLUSTR.2005.347024

[15] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *Journal on Computer*, vol. 38, no. 11, pp. 32-38, 2005, doi: 10.1109/MC.2005.379.

[16] J. Liu, K. Li, D. Zhu, J. Han, and L. Li, "Minimizing cost of scheduling tasks on heterogeneous multicore embedded systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 2, 2017, doi: 10.1145/2935749.

[17] H. Cheng, "A high efficient task scheduling algorithm based on heterogeneous multi-core processor," *2010 2nd International Workshop on Database Technology and Applications*, 2010, pp. 1-4, doi: 10.1109/DBTA.2010.5659041.

[18] T. Bridi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini "A constraint programming scheduler for heterogeneous highperformance computing machines," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 2781-2794, 1 Oct. 2016, doi: 10.1109/TPDS.2016.2516997.

[19] D. Ye, DZ. Chen, and G. Zhang, "Online scheduling of moldable parallel tasks," *Journal of Scheduling*, vol. 21, pp. 647-654, 2018, doi: 10.1007/s10951-018-0556-2.

[20] Y. Yang, S. Yu, and X. Bin, "A new dynamic scheduling algorithm for real-time heterogeneous multiprocessor systems," *Workshop on Intelligent Information Technology Application (IITA 2007)*, 2007, pp. 112-115, doi: 10.1109/IITA.2007.70.

[21] E. Amal, A. E. Nirmeen, and E. Ayman, "A new task scheduling algorithm for miximizing the distributed systems efficiency," *International Journal of Computer Applications*, vol. 110, no. 9, pp. 9-16, 2015, doi: 10.5120/19343-0279.

[22] A. George and A. Maria, "Dynamic task scheduling methods in heterogeneous systems- a survey," *International Journal of Computer Applications*, vol. 110, no. 6, pp. 12-18, 2015, doi: 10.5120/19318-0859.

[23] X. Ling, Q. Jiazhong, L. Shukuan, and Z. Wanting, "Dynamic task scheduling algorithm with deadline constraint in heterogeneous volunteer computing platforms," *Future Internet*, vol. 11, no. 6, p. 121, 2019, doi: 10.3390/fi11060121.

[24] R. Ramezani, "Dynamic scheduling of task graphs in multi-FPGA systems using critical path," *J Supercomput*, vol. 77, pp. 597-618, 2020, doi: 10.1007/s11227-020-03281-3.

[25] A. Priya and S. K. Sahana, "Processor scheduling in high-performance computing environment using MPI," in *Innovations in Soft Computing and Information Technology*, Singapore: Springer, 2019, pp. 43–54, doi: 10.1007/978-981-13-3185-5_5.

[26] T. Tobita and H. Kasahara, "A standard task graph set for fair evaluation of multiprocessor scheduling algorithms," Journal of Scheduling, vol. 5, no. 5, pp. 379-394,2002, doi: 10.1002/jos.116.

BIOGRAPHIES OF AUTHORS

Takuma Hikida received his BE and ME degrees from Ritsumeikan University in 2019 and 2021, respectively. At present, he works for Fujitsu Ltd. His research interests include task scheduling on multicore. He is a member of IEEE.

Hiroki Nishikawa received his BE and ME degrees from Ritsumeikan University in 2018 and 2020, respectively. At present, he is in the Ph. D program at Ritsumeikan University. His research interests include system-level design methodologies, design methodologies for cyber-physical systems, and so on. He is a member of IEEE.

Hiroyuki Tomiyama received his BE, ME, and DE degrees in computer science from Kyushu University in 1994, 1996, and 1999, respectively. He worked as a visiting researcher at UC Irvine, as a researcher at ISIT/Kyushu, and as an associate professor at Nagoya University. Since 2010, he has been a full professor with the College of Science and Engineering, Ritsumeikan University. He has served on program and organizing committees for a number of premier conferences including DAC, ICCAD, DATE, ASP-DAC, CODES+ISSS, CASES, ISLPED, RTCSA, FPL, and MPSoC. He has also served as an editor-in-chief for IPSJ TSLDM; an associate editor for ACM TODAES, IEEE ESL, and Springer DAEM; and a chair for the IEEE CS Kansai Chapter and IEEE CEDA Japan Chapter. His research interests include, but are not limited to, design methodologies for embedded and cyber-physical systems.