Transactions on Machine Intelligence 1(1) (2018) 39–48

ISSN Online: 2821-1693

Transactions on Machine Intelligence

Journal Homepage: https://tmachineintelligence.ir/

On-Line Reusing-Based Scheduling Algorithm for 2-Dimensional Tasks in Reconfigurable Hardware

A. Parisay[1], H. Shahriar Shahhoseini[2], S. M. Mohtavipour[3,*]

[1,2,3] Department of Electrical Engineering, University of Science and Technology (IUST), Tehran, Iran

ARTICLE INFO

Article History:
Received 19 January 2018
Received in revised form
22 February 2018
Accepted 11 March 2018
Available online 11 March 2018

Keywords:
Dynamically Reconfigurable System, Reconfiguration overhead, On-line Scheduling, Hardware partition

ABSTRACT

Reducing reconfiguration overhead is critical to improving the runtime performance of dynamically reconfigurable Field-Programmable Gate Arrays (FPGAs). In this paper, we introduce a novel task-reuse strategy tailored for two-dimensional FPGA hardware layouts. The key idea is to identify and exploit repetitive computations by reusing already-configured hardware modules rather than incurring costly bitstream reloads. First, incoming tasks are classified into two categories significant (high-impact or frequently appearing) and non-significant based on metrics such as execution frequency, resource intensity, and temporal locality. Each category is assigned to its own hardware partition. Within the significant partition, when a new significant task arrives, the system either replaces an existing module whose future utility is low or, if sufficient empty regions exist in the non-significant partition, temporarily maps the task there. If neither option is feasible, the partition boundary is extended to accommodate the new module, up to predefined physical limits. We evaluated our approach on a suite of benchmark applications exhibiting high task repetition. Compared to leading dynamic-reconfiguration algorithms, our method reduced overall makespan by 20.3% on average. Moreover, the task-placement decision algorithm operates in polynomial time, achieving placement decisions over three times faster than competing strategies. These results demonstrate that intelligent partitioning combined with selective reuse and partition-border extension can substantially lower reconfiguration overhead and accelerate FPGA-based computation pipelines.

1. INTRODUCTION

Reconfigurable Computing (RC) provides both benefits of Application Specific Integrated Circuit (ASIC) devices' high performance and General Purpose Processor (GPP) devices' flexibility to make it an excellent option for most applications, especially those with parallel processes, such as cryptography [1], video applications [2] and image processing [3]. Most RC systems include one Central Processing Unit (CPU) for management of data

* Corresponding author: mehdi_mohtavipour@elec.iust.ac.ir
Department of Electrical Engineering, University of Science and Technology (IUST), Tehran, Iran

http://dx.doi.org/10.47176/TMI.2018.39

communication and software implementation, and one or several Reconfigurable Processing Units (RPU) for hardware implementation of tasks and instructions [4, 5]. Because of their block structure and re-configurability features, FPGAs are considered as the main RPU platform in RC field. Also re-configurability is categorized in two methods in FPGAs, static and dynamic. In static method whole device's process is stopped and new configuration data is injected into the device, this is also referred to single context reconfiguration. Similar to single context, multi-context models exist too. In multi-context several parallel plates for the FPGA is developed and each time one of these plates is needed, context is switched rapidly. In this model switching only happens between plates and each of them can be considered as a single context FPGA. On the other hand in dynamic method which is called Run-Time Reconfiguration (RTR), each part of every FPGA context can be reconfigured during works of some other parts. This is very beneficial because no stopping is needed for reconfiguration if the data is on another context, and more physical area can operate simultaneously while other contexts are being reconfigured. It should be noted that [6].

Each FPGA consists of several rows and columns of Configurable Logic Block (CLB) and routing resources is used to connect these CLBs for data communication. Tasks of an application use some of CLBs to implement their function. For each task hardware for implementation of tasks is always smaller than sum of all task sizes. Also dynamic approach made it possible to better satisfy application performance and power requirements by enabling customization of the limited logic resources one rectangular shape size is considered which covers the number of needed CLBs. Scheduling and placement of tasks are the most important challenges that prevents reconfigurable systems to achieve their maximum potential performance. Because of tasks' rectangular shape, placing them in RPU surface can lead to fragmentation, hence growth of free unusable space and utilization problems is inevitable. Task temporal placement methods manage free spaces to find the best place for each task and reduce fragmentation of RPU surface. Placement algorithms can perform in two ways, offline and online. In offline placement, all information about tasks' characteristics such as arrival time or dimensions are known before the program starts, and many time consuming computational methods like evolutionary algorithms are used to find an optimum answer. For example, a hybrid placement strategy based on genetic and greedy algorithms is proposed in [7] to efficiently place a set of tasks before system starts to work. On the other hand online algorithms decide with no information about incoming tasks in the future. When a new task arrives, one appropriate place is chosen for it on the surface of device, based on the state of the system.

Online algorithms' design is very complex but more realistic, because details of most applications cannot be predicted and calculated before running, while they play an important role in offline scheduling of RC tasks. Computation and decision time of online algorithms is a critical metric that should be small enough to make the algorithm able to decide before next task is arrived. In addition to finding an appropriate place, reconfiguration overhead is one of the main obstacles that should be tackled. Communication between memory and reconfigurable device has a large latency and even in some cases is larger than execution time of task. In order to improve the performance of such systems, placement and scheduling algorithms can be constructed based on these considerations. In this paper, by assuming that when program consists of repetitive tasks, reusing of existing hardware configuration can reduce the overhead, a novel strategy in design of scheduling and placement algorithms for reconfigurable systems is proposed. This strategy's main contributions can be mentioned as follows:

- Detecting of repetitive tasks and their characteristics
- Management of repetitive and non-repetitive tasks and allocating area of a 2-Dimentional RPU
- Introducing preserving and replacement policies for repetitive tasks

The rest of this paper is organized as follows, in section 2 previous works on reconfigurable computing focusing on reconfiguration overhead are introduced, in section 3 problem is further explained with the assumed model of reconfigurable hardware and tasks, and the novel strategy for the placement of repetitive and non-repetitive tasks is proposed in section 4. In section 5 simulation model and conditions are described and then results for the proposed algorithm in several scenarios are presented and conclusion and future works form section 6 of this paper.

2. RELATED WORKS

In this section some efforts on mitigating the effect of reconfiguration overhead and placement fragmentation is reviewed. In [8-9] authors presented online scheduling examples which in [8], with use of multiple sizes for each task, common border with other tasks and device boundaries in space and time are calculated for each task, and the

best place is chosen, causing low surface fragmentation and in [9] authors proposed two methods in scheduling and placement. First method which is named horizon, places the tasks by investigating only the free spaces which will not be occupied in future. But the second method which is called stuffing, computes termination time of existing tasks and predicts future situations of the incoming tasks in order to consider more free regions than the first method missed, leading to a better performance improvement.

Authors in [10] developed a cache based strategy to reduce the reconfiguration time for several models of FPGA device such as single context or multi-context. Some challenges like non-uniform configuration size for caching or large size of data which is led to small number of configuration are investigated in this research. In [11] configuration prefetching has been introduced and issues according to single or multi context FPGA have been investigated. In single context model one configuration is pre-fetched into the device when the current configuration is running on it. In multi-context model several configurations are loaded into the device and one dynamic algorithm for reducing the prefetching penalty based on the Markov prefetching method is proposed. When dependent tasks are modeled for reconfigurable hardware, reconfiguration time plays an important role for overlapping with execution time. Works in [12] inspects this fact that reconfiguration part of each task can be done before it needs to be executed. So based on this assumption, their algorithm pre-fetches tasks as soon as possible and two metrics are calculated for it, priority of the tasks and placement decision to avoid conflicts between tasks.

Some memory hierarchy models also are introduced to reduce the configuration overhead which in [13] two on-chip memories with different access time are discussed and efficient mapping algorithms with respect to energy consumption and access latency features are proposed. [14] is recent scheduling method based on our works which mitigated the configuration overhead. In this work, tasks are separated into two groups, called significant and non-significant and one 1-Dimentional FPGA is divided into two partitions for each and tasks with high repetition probability are persevered in significant partition for future use. One simple Boolean equation for making a decision whether a task is significant or not, and another simple equation for resizing significant and non-significant partitions were two aspects of our works. Although it was shown that this approach can notably improve the performance of reconfigurable systems with repetitive tasks, but still some further works on reusing based scheduling algorithm are necessary which is discussed in this paper.

3. PROBLEM DEFINITION AND MODEL

It is mandatory to have a system for managing hardware tasks, since in most of reconfigurable systems, the FPGA device is used to implement hardware tasks with finite number of CLBs and Interconnections. The ideal form of operation is using all of the FPGAs resources, causing a %100 Utilization Factor (UF) of FPGA surface, but it cannot be achieved and preserved due to fragmentation. Equation (1) demonstrates how U.F is calculated, in which n is number of tasks which were executed in the program, w_i and h_i are width and height of task number i consecutively, lt_i is lifetime of task number i, O_i is the omitted reconfiguration overhead of task number i, W_{FPGA} and H_{FPGA} are width and height of our FPGA consecutively and T_f is the time that program is finished. If the task is reused, O_i is its reconfiguration overhead and if not, O_i will be zero for task number i.

$$U.F. = \frac{\sum_{i=1}^{n} w_i \times h_i \times (lt_i - O_i)}{W_{FPGA} \times H_{FPGA} \times T_f} \tag{1}$$

It should be kept in mind that task replacement is time consuming due to task's reconfiguration overhead. So it is ideal to have a task on chip's surface upon its arrival, but also this cannot happen all the time, because in online placement algorithm, as mentioned in section 2, the placement decision unit has no information about the task which is coming in the future and it recognizes the task at its time of arrival. Deciding on which task is the most proper task to be removed to make space for new task is sometimes tough and can go wrong, because maybe the next new task which is about to come, is the same as the task which we eliminated from the chip. So having a good decision algorithm can hugely improve performance in some cases. If a task is repeated in time, it rises the probability that it is going to be repeated in the future too, so it can be a wise decision to keep a task with high probability of occurrence on the chip's surface after its work is done. A parameter called repetition ratio (RR) determines how repetitious a program is. By preserving a task on chip's surface, if the new task is the same, the previously configured task starts running instantly and reconfiguration overhead is omitted. So predicting whether a task is going to be needed again

correctly, have a major effect on reducing Task Rejection Ratio (TRR) and consequently, reducing runtime. Poisson probability distribution is used to calculate the possibility that a certain task is coming n times in the next Δt time, using λ which is mean of its occurrence in the past Δt time. Equation (2) is the Poisson probability function in general form.

$$P_{\Delta t}(n) = \frac{\lambda^n e^{-\lambda}}{n!} \tag{2}$$

$$P_{i,1}(1) = \frac{{\lambda_i}^1 e^{-\lambda_i}}{1!} = \lambda_i e^{-\lambda_i} \tag{3}$$

Sometimes two or more implementations of a hardware task is available which are different in size, shape and running time, but they all do the same task and produce the same results. Deciding on which implementation is the best choice for a certain situation is critical. If an implementation is both smaller and faster, the other version has no superiority and it should not be used. In a realistic and simple perception, most of tasks have two versions, one faster task which has bigger dimensions, and one version which is smaller, but takes more time to finish. Deciding which one is more appropriate can be an aspect of this problem. It should be noted that tasks cannot wait more than a certain deadline. Sometimes the newly arrived task can wait for a similar task to finish and start immediately, causing elimination of reconfiguration overhead and no deadline is reached. If waiting for the previous instant of task to finish working causes passing the deadline, another place for executing the task should be found. If a task cannot be placed on the surface of chip, it will be rejected. If a task is rejected, it is going to be run on the main CPU using a software algorithm which is always slower than executing the same task on a hardware implementation. We consider this difference as a rejection penalty. The algorithm should prevent rejection to reduce makespan.

In this paper, a simple method, called grouping of tasks is used to model task dependency. If some tasks are in a group together, it means that the next group, needs the data produced by this task group. It means that if all of the tasks in a group are not finished executing, the next group could not start. In the real world this situation happens when a section of a code, like a loop, is computing data which is needed to start execution of the next section of the code. We wait for a certain amount of time called group's arrival time for the next group to come. If the next group arrives before the previous groups is finished completely, it has to wait. The main CPU is assumed to be a single core CPU and if too many tasks are rejected, it will create a queue of tasks waiting to be executed by software on the main CPU. We add the time needed for the main CPU to finish all of its tasks to the deadline of our tasks. It is because when a task has a deadline, it means that if we have to wait more than that deadline, it is better to run the task by the main CPU using software. If the main CPU is busy doing previous rejected tasks, it means that it is reasonable to wait for the task to be done on the FPGA.

In our proposed model, tasks are sent to the reconfigurable processor to be executed using hardware implementation one after another. It is easily concluded that if tasks come rapidly after each other, it puts more tasks in risk of being rejected by the reconfigurable processor and if we have a proper amount of delay after each task is received, processor has decent time to finish every task and start the next one. In a program, all tasks come after each other after certain amount of time is passed.

Now that we know the model made to simulate the algorithms, we can introduce our proposed algorithm in the next section.

4. PROPOSED ALGORITHM

In this paper, we propose a new 2-Dimensional task scheduling algorithm, in which we try to re-use every task after it is finished executing. So first of all we have to decide whether a task is a significant task and should be preserved after its execution is finished or not. As it was described in section 3, we use Poisson probability distribution function to decide how much a task is likely to come in the future. Another condition also plays a role in our decision, and it is the ratio of overhead to the lifetime of a task. If a huge portion of a task's lifetime is reconfiguration overhead, it means that if we reuse this task, more overhead is eliminated. If a task's Poisson probability is greater than threshold th_1 and its ratio of reconfiguration overhead to execution time is greater than threshold th_2 (4),we call it a significant task and we treat it differently compared to non-significant tasks.

$$P_i > th_1 \; and \; \frac{o_i}{lt_i - o_i} > th_2 \rightarrow T_i \; is \; significant \tag{4}$$

We divide chip's area into two partitions. One partition is for significant tasks and one is for non-significant tasks. Partitions can change their size if certain conditions occur. Non-significant tasks are eliminated just after their execution is finished, but significant tasks are preserved after their execution is finished, waiting for another instance to come in the future. We do this because it prevents fragmentation to spread all over chip's surface and it stays in the non-significant partition only. When a task is non-significant, time needed for the task to finish executing in a certain place is calculated, and repeated for all the partition's area. If two or more places have the same time of finish, the first place found is chosen. If not, the best place is the smallest time and the task is configured there. If no place could be found to configure the task and meet its deadline, the task is switched with its faster version and the same process goes on. If no place could be found again, the availability of changing the size of partition is checked. If partition resizing was not possible, the task is rejected. If a task is significant, it goes through numerous steps which are more complicated than steps we pass for a non-significant task. Here is a list of works that are done on a significant task:

a) First of all, a task is searched through execution list (EL), reservation list (RL) and preserved list (PL), lists that contain tasks which are executing, are reserved and are waiting idle on chip consecutively. If we have the same task in these lists, we have to check if we re-use the task, deadlines are passed or not. If an instance is founded and deadlines are not passed, the task would be added to the reservation list and in this manner, the overhead is eliminated. If no instances of task were found, the algorithm goes to next step. If an instance of the task is found in these lists but could not be reused due to deadline restrictions, this means that if the algorithm places it anywhere else, the task which is in the list should no longer be preserved and it is deleted from the preserved list after a place for the new task is found. The same process goes for the faster version again.

b) If an instance of the task is not found or it could not be reused due to deadline restrictions, the algorithm has to search for free area on the significant partition. First places with minimum time of start are searched, considering EL, RL and PL. Among all places with that specific minimum number, the first one is chosen and if minimum start time is equal to time of arrival, configuration and execution of the task would start right away. If not, it will be added to the reserved list. The algorithm do the same exact procedure for the faster, but bigger version.

c) If no place could be found on the significant partition, the algorithm checks for partition resizing availability. Partition resizing availability has two conditions: First, the other side should be free enough. It means that more than 40 percent of its area should be free. Second, if the partition is resized, then border of two partitions would not crossed any task. If these two conditions were met, the algorithm move the border 1 unit and start from the beginning again.

d) If changing the partition's size is not possible, the algorithm checks for replacement availability. For checking replacement availability, the algorithm goes through all tasks in RL. It looks to find the smallest task that is bigger in width and height than the new task. If a task with these conditions were found, it will be replaced with the task and delete it from PL. If not, all tasks in PL are checked that if the algorithm removes them from the chip, will there be a free space big enough to accommodate the new task? It is done because sometimes a task in PL is neighbor to a free space in chip's surface and if the algorithm removes it from chip's surface, the free area created will become large enough to accommodate the new task. If it finds a task with these conditions in PL, it will be replaced with the new task.

e) If the algorithm could not find any task in PL to replace it with the new task, it goes through EL and RL to check if there are any instances of the new task in the non-significant partition that can be reused and prevent task rejection. If the algorithm finds an instance that if reused, deadlines are not passed, it will be reused in the non-significant partition, although the task is significant. The algorithm does the same exact search for the fast version again.

f) If no instance of the task could be found on the non-significant partition, the algorithm searches it for free area available and if it finds a place for it, the task starts to execute. The algorithm checks faster version afterwards and if no place could be found, the task is rejected.

Once the algorithm completes execution for a given task, it restarts for the subsequent task. Notably, the algorithm is designed to handle high-priority (significant) tasks with special consideration, ensuring that the rejection of such tasks is minimized this is a key focus of the proposed approach. As will be demonstrated in the following section, the proposed algorithm performs particularly effectively for workloads with a high repetition ratio. Since the method relies on task reuse, repeated tasks allow the system to significantly reduce reconfiguration overhead, thereby improving overall makespan.

5. RESULTS AND SIMULATION

In Fig. 1, the algorithm is run for a program with 500 tasks, repetition ratio of %50, rejection penalty of 3, group size of 200, 10 known tasks, initial partition border of 140 and height and width of the FPGA is assumed to be 200 and only 2 groups of this program are visible here. Width and height of the FPGA are across X and Y axis and Z is the time axis. Blue tasks are significant and white tasks are non-significant. As it can be seen, partition is resized in the first group and also it is resized after group 1 is completed. Some blue tasks are also visible in the non-significant partition which are significant tasks that are placed in the free area or reused from tasks in the non-significant partitions. Task reusing is clearly obvious in tasks which are all alike and on top of each other in the significant part, and also task replacement is seen in all tasks which are not alike but on top of each other in the significant area. The distance in time between two groups is caused by group's arrival time.

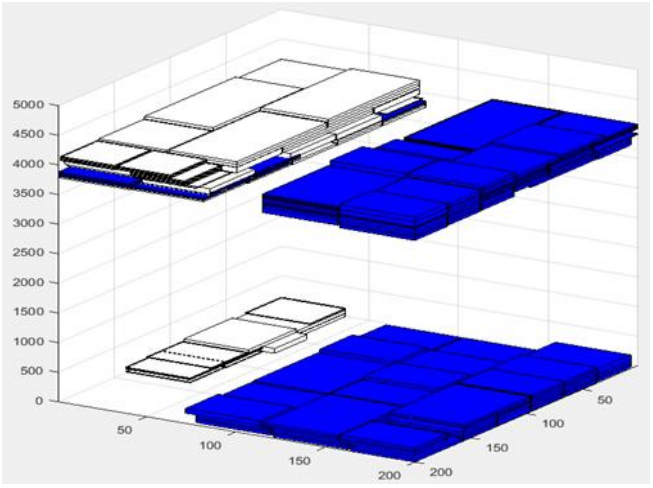

Fig. 1. Result of task placement with two significant and non-significant groups

Table 1. Simulation Conditions

Parameter	Value
FPGA size	200×200
No. of tasks in a program	1000
No. of known tasks	10
Group size	500
Rejection penalty	5
Width and Height of tasks	(27-45)×(27-45)
Task's lifetime	(5-100)
Ratio of reconfiguration overhead to lifetime	(0.3-0.5)
Time of next task's arrival	(0-3)
Deadline of start after arrival	(10-100)+queue time of CPU

To measure the performance of the algorithm, it can be compared to another state-of-the-art algorithm in task scheduling. To prove that our proposed algorithm can perform better in high repetition ratios, we compared it with [8], an algorithm which is contributed to finding the best place for a task to prevent fragmentation and performed very well during simulation, but it takes long time to calculate all of the parameters used by it. Sometimes it is crucial to decide the best place for a task quickly and this algorithm seems to face some problems in some applications. Another algorithm which we compare our work with it, is the first fit algorithm. It is known to decide very quick and waste little time deciding which place to put the task, bus it can be seen in this section that in high workloads, sometimes it has to search the entire FPGA surface to find a place for the task. Although it decides faster than the previous algorithm, it is far behind the proposed algorithm in terms of runtime.

The program is run for conditions noted in Table I. Repetition ratio is changed from %0 to %60 in steps of %10 to see whether our assumption about better performance of the proposed algorithm in higher repetition ratios is correct or not. Fields containing two numbers in parentheses divided with a hyphen means that they are randomly chosen between the two numbers.

The results derived from simulation are shown in fig 2-5. Fig. 2 shows average rejection ratio of the 3 algorithms in 50 Runs on a PC desktop with Intel® Core i3 550 CPU running at 3200 GHz, 6GB of RAM on MATLAB R2013a using windows 7. As it was expected, the Best Fit (BF) algorithm shows very little reaction to repetition ratio, since it can find places for tasks that are alike easier, due to lower fragmentation caused by repetitive tasks which are the same in size and shape. The First Fit algorithm (FF) also shows almost no reaction to repetition ratio, just like the best fit algorithm. But as it can be seen, the proposed algorithm outperforms two other algorithms as the repetition ratio goes high due to higher task reusing.

Fig 3. shows average utilization factor of the chip's surface. Again a slight change in BF, very small change in FF and huge improvement for the proposed algorithm since rejection ratio and utilization factor has reverse relationship with each other. Also Fig. 4 shows makespan of the 3 algorithms. Again for the same reason it can be seen that higher repetition ratios causes the algorithm to perform better and we can see %20.3 less makespan that means %20.3 speed improvement in %60 repetition ratio.

Fig 5. shows runtime of algorithms. As it can be seen, the BF and FF are far behind the proposed algorithm in all repetition ratios. As it was expected, the proposed algorithm's runtime is reduced when repetition ratios goes higher, since tasks are checked within lots of conditions before they are rejected and rejected tasks waste more time than tasks which are executed or reserved to be run in the future. Consequently, less rejection ratio means less runtime for the proposed algorithm. For the BF, as repetition ratio goes higher, fragmentation is reduced and consequently, calculations for each task is reduced, causing less runtime.

But since it first checks for rejection and rejected tasks waste very little time, runtime is increased with repetition ratio, causing middle range of repetition ratio to be the most challenging for this algorithm in terms of runtime. For the FF it is the opposite. Tasks that are rejected, has searched all the FPGA's surface and could not find a suitable place to be reconfigured.

So more rejection ratio means more runtime, but as tasks are rejected more and more, EL and RL are less occupied and going through them to check whether a place is suitable for task reconfiguration is much more simple, causing higher task rejection ratio to be the reason of less runtime from this point of view. Combining these two reasons, again middle range repetition ratios are the most challenging to the FF in terms of runtime.

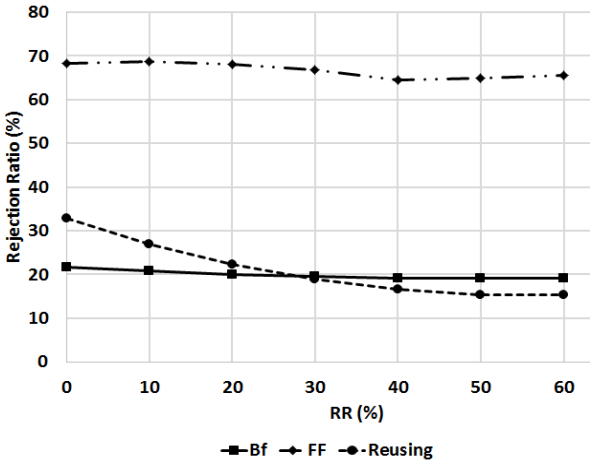

Fig. 2. Rejection Ratio against Repetition Ratio

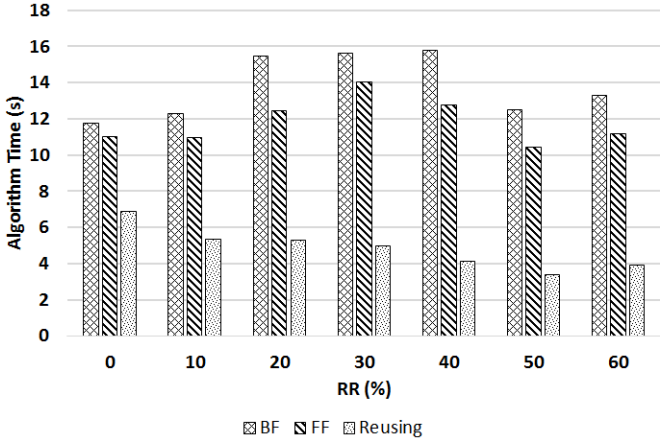

Fig. 3. Algorithm runtime against Repetition Ratio

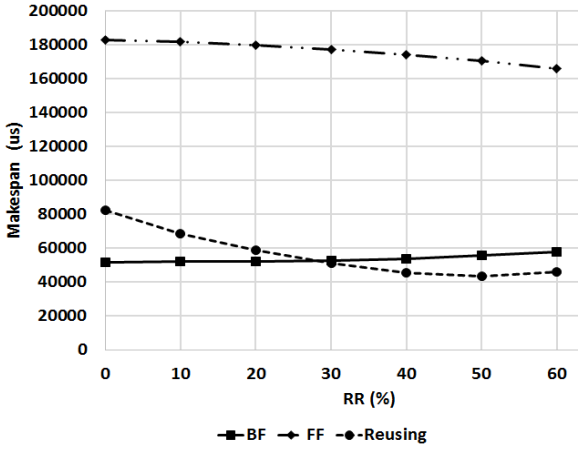

Fig. 4. Makespan against Repetition Ratio

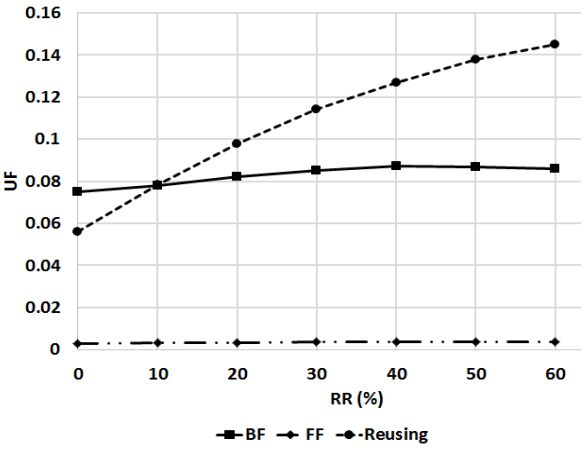

Fig. 5. Utilization Factor against Repetition Ratio

6. CONLUSION

In this paper, we introduced an algorithm that can place and organize tasks within a program to be configured on an FPGA's chip surface with concentration on task reusing. The algorithm was proven to perform better than other state-of-the-art algorithms in higher repetition ratios in terms of makespan and hugely better than very simple algorithms in terms of algorithm time. This makes it perfect for applications with high repetition ratios which task management unit is placed on the FPGA, but the FPGA is running on low frequencies or applications which task management unit is run on CPU, but the CPU is busy or it is running with low speed. The algorithm tries its best with use of some novel approaches to prevent tasks from being rejected like searching through the opposite partition for available task configuration.

This novel algorithm can be further developed by using intelligent algorithms such as intelligent and fuzzy methods to become more robust in different program conditions and applications. It can also be tested on a more realistic model with more parameters, like considering the interconnection resources, considering dependencies within a group or assuming that the main CPU has more than one core.

CONFLICTS OF INTEREST

The authors declare no conflict of interest.

REFERENCES

[1] Pöppelmann, T., Naehrig, M., Putnam, A., & Macias, A. (2015). Accelerating homomorphic evaluation on reconfigurable hardware. Lecture Notes in Computer Science (pp. 143–163). https://doi.org/10.1007/978-3-662-48324-4_8

[2] Liu, L., Wang, D., Chen, Y., Zhu, M., Yin, S., & Wei, S. (2016). An implementation of multiple-standard video decoder on a mixed-grained reconfigurable computing platform. IEICE Transactions on Information and Systems, E99-D(5), 1285–1295. https://doi.org/10.1587/transinf.2015EDP7369

[3] Fons, M., Fons, F., & Canto, E. (2010). Fingerprint image processing acceleration through run-time reconfigurable hardware. IEEE Transactions on Circuits and Systems II: Express Briefs, 57(12), 991–995. https://doi.org/10.1109/TCSII.2010.2087970

[4] Kao, C.-C. (2015). Performance-oriented partitioning for task scheduling of parallel reconfigurable architectures. IEEE Transactions on Parallel and Distributed Systems, 26(3), 858–867. https://doi.org/10.1109/TPDS.2014.2312924

[5] Saha, P., & El-Ghazawi, T. (2007). A methodology for automating co-scheduling for reconfigurable computing systems. Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE 2007), Nice, France. https://doi.org/10.1109/MEMCOD.2007.371229

[6] Banerjee, S., Bozorgzadeh, E., Dutt, N., & Noguera, J. (2007). Selective bandwidth and resource management

in scheduling for dynamically reconfigurable architectures. Proceedings of the 44th Annual Design Automation Conference (DAC '07), San Diego, CA. https://doi.org/10.1145/1278480.1278673

[7] Liang, H., Sinha, S., Warrier, R., & Zhang, W. (2015). Static hardware task placement on multi-context FPGA using hybrid genetic algorithm. 2015 25th International Conference on Field Programmable Logic and Applications (FPL), London, United Kingdom. https://doi.org/10.1109/FPL.2015.7293954

[8] Marconi, T. (2014). Online scheduling and placement of hardware tasks with multiple variants on dynamically reconfigurable field-programmable gate arrays. Computers & Electrical Engineering, 40(4), 1215–1237. https://doi.org/10.1016/j.compeleceng.2013.07.004

[9] Steiger, C., Walder, H., & Platzner, M. (2004). Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. IEEE Transactions on Computers, 53(11), 1393–1407. https://doi.org/10.1109/TC.2004.99

[10] Li, Z., Compton, K., & Hauck, S. (2000). Configuration caching management techniques for reconfigurable computing. Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2000), Napa Valley, CA. https://doi.org/10.1109/FPGA.2000.903390

[11] Li, Z., & Hauck, S. (2002). Configuration prefetching techniques for partially reconfigurable coprocessors with relocation and defragmentation. Proceedings of the 2002 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '02), Monterey, CA. https://doi.org/10.1145/503048.503076

[12] Khuat, Q.-H., Chillet, D., & Hubner, M. (2014). Considering reconfiguration overhead in scheduling of dependent tasks on 2D reconfigurable FPGA. 2014 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), United Kingdom. https://doi.org/10.1109/AHS.2014.6880151

[13] Clemente, J. A., Ramo, E. P., Resano, J., Mozos, D., & Catthoor, F. (2014). Configuration mapping algorithms to reduce energy and time reconfiguration overheads in reconfigurable systems. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 22(6), 1248–1261. https://doi.org/10.1109/TVLSI.2013.2271917

[14] Shahriar Shahhoseini, H., Mansub Bassiri, M., & Mehdi Mohtavipour, S. (2014). Performance Evaluation of Reusing Based Scheduling in On-line Reconfigurable Computing Systems. *The CSI Journal on Computer Science and Engineering*, *11*(2), e215844.