Natl. Acad. Sci. Lett. (January–February 2020) 43(1):27–31
https://doi.org/10.1007/s40009-019-00810-4

SHORT COMMUNICATION

Reducing Reconfiguration Overheads Using Configuration Prefetch, Optimal Reuse, and Optimal Memory Mapping

I. Hariharan[1] · **M. Kannan**[1]

Received: 18 March 2018 / Revised: 27 July 2018 / Accepted: 11 March 2019 / Published online: 6 April 2019

Abstract Modern embedded systems are packed with dedicated field-programmable gate arrays (FPGAs) to accelerate the overall system performance. But the main drawback in using FPGA as a reconfigurable system is that a lot of reconfiguration overheads are generated in the reconfiguration process. The reconfiguration overheads are mainly because of the configuration data being fetched from the off-chip memory and also due to the improper management of tasks during execution. This work focusses mainly on the prefetch heuristics, reuse technique, and the available memory hierarchy to provide an efficient management of tasks over the available resources. This short communication proposes a new optimal replacement policy which reduces the overall time and energy reconfiguration overheads for static systems in their subsequent iterations. It is evident from the results that most of the time and energy reconfiguration overheads are eliminated.

Keywords Reconfiguration overheads · Configuration mapping · Optimal replacement policy · Field-programmable gate array (FPGA) · Multimedia application · Scheduling

Abbreviations

HS	High-speed on-chip memory
LE	Low-energy on-chip memory
RU	Reconfigurable unit
CM	Configuration Mapper
L	Last
Info table	Information table

✉ I. Hariharan
hariharan166@gmail.com

M. Kannan
mkannan@annauniv.edu

[1] Department of Electronics Engineering, MIT Campus, Anna University, Chennai, Tamil Nadu, India

Field-programmable gate array (FPGA) has the flexibility to alter its characteristics in order to satisfy customer needs. This property influences many applications to opt for FPGA as an efficient alternative to other processor systems [1]. But the run-time partial reconfiguration feature of FPGA generates time (typically in the order of hundreds of milliseconds [2]) and energy reconfiguration overheads [3]. These overheads are because of the configuration data being fetched from the off-chip memory for loading the configurations onto the available FPGA resources before execution. Improper management of tasks and the available FPGA resources contribute further to reconfiguration overheads. These overheads can degrade the system's performance if it is not properly managed.

Several techniques have been proposed for reducing the reconfiguration overheads. In [4, 5], the authors give the importance of configuration prefetching technique to reduce the reconfiguration overheads. This technique involves in the pre-loading of future configuration when the current configuration is in the execution phase. Also, the technique of reusing the already stored configuration reduces the overall system's reconfiguration overheads. Better results are achieved using the reuse technique effectively [6]. Significant results are obtained [7, 8] by considering both the prefetch heuristics and reuse technique. Clemente et al. [9] introduced a memory hierarchy which consists of on-chip and off-chip memories. On-chip memory is divided into high speed (HS) and low energy

(LE). In addition, the authors used configuration mapping algorithms to reduce both the energy and time reconfiguration overheads. Authors [10] extended the traditional configuration caching approaches by dividing the configuration into blocks. Thus, the granularity of the configurations is reduced and managed efficiently to reduce the reconfiguration overheads.

The static system mainly consists of one task graph or a group of task graphs executed in similar fashion for a large number of iterations. The system which deviates from the static system scenario is called as dynamic systems. A novel run-time prediction-based algorithm for the dynamic system [11] is proposed where time reconfiguration overhead is significantly reduced. Many scheduling algorithms have been proposed for multi-core computing systems. But, most of the authors failed to focus on communication costs involved in the run-time reconfiguration process. In [12], efforts are made to reduce the communication overheads. Task scheduling algorithms like load balancing, simulated annealing, ant colony and CMWSL are proposed [13] to reduce the reconfiguration overheads. By properly utilizing the reconfigurable region, it is possible to reduce the reconfiguration overheads. Authors [14] worked in employing such methods.

The main aim of this short communication is to reduce the reconfiguration overheads generated in a reconfigurable system. To achieve this objective, an architecture is proposed as given in Fig. 1. This architecture can be realized in any of the last-generation FPGAs such as Xilinx Virtex-7 and Zynq-7000 EPP devices [15, 16], or its equivalent in Altera FPGAs [17, 18]. The communication infrastructure may use a network-on-a-chip (NOC) model, or it may be implemented using one or several buses. For instance, Xilinx provides communication infrastructures like Advanced Extensible Interface [19]. It can be used to link the computational cores and the local memory bus (LMB) [20] for connecting memories.

The architecture consists of a microprocessor which controls the general operations of an embedded system. Initially, all the possible configurations to be loaded at run-time are made available inside the off-chip memory. These configurations are available in the form of task graphs. A single task graph contains many tasks. Any task within the task graph is the basic scheduling unit loaded in individual reconfigurable units (RUs) for execution where RU is the smallest portion in an FPGA reserved for loading the configuration.

When the user interrupts for a specific application, then the microprocessor finds the exact task graph to be scheduled from the off-chip memory. The Configuration Mapper (CM) then analyzes the selected task graph and gives an optimal mapping (for every individual task present in the task graph) in the available on-chip memories. Apart from an optimal memory mapping, the CM gives the pattern of the schedule to be followed. The working principle of CM is explained in the following steps.

1. For any task from a task graph, the first step of the CM is to assign a specific RU from the available RUs.
2. After assigning the RU, the task is checked for the memory mapping condition. Based on the result, the same task is assigned to either HS or LE memory.

Fig. 1 Proposed architecture

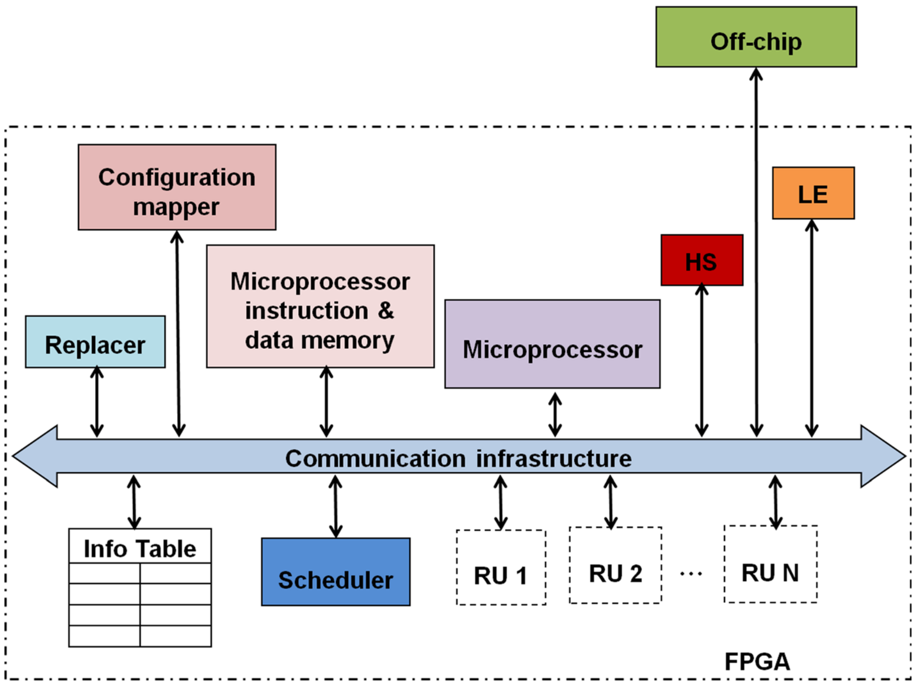


Table 1 Memory characteristics

Memory modules	The access time for each configuration fetch (ms)	Normalized energy consumption (units)
HS	4	1
LE	6	0.7
External memory	12	4

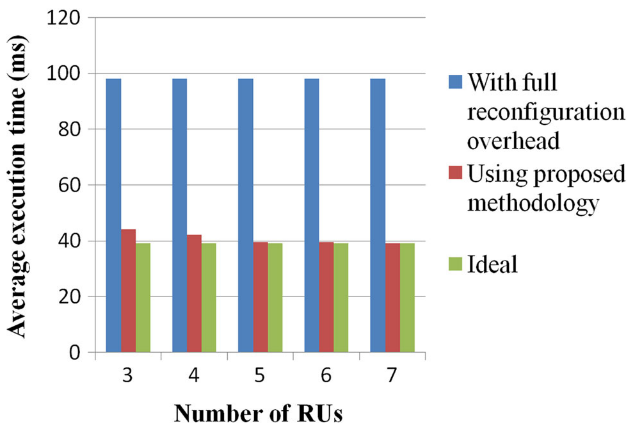


Fig. 2 Performance evaluation of the proposed architecture

The step 1 of CM concentrates on assigning RU (for every task). RU allocation is in a clockwise direction starting from the first RU. It is sometimes impossible to assign all the tasks in the available RUs without making any replacement. Because in most cases, the number of tasks present in any task graph is greater than the available number of RUs. Hence for larger task graphs, a replacement policy must be followed. The replacement policy used in this proposal aims to reuse the vital RUs. Normally for a static system, most of the RUs holding the initial tasks are considered as vital. This is because, in a static system, the same application is going to be executed for a large number of iterations. And by reusing the initial tasks, it is easier to completely eliminate the generation of reconfiguration overheads (for the initial tasks) in subsequent iterations of the same task graph execution. Meanwhile the execution of initial tasks, it is also feasible to prefetch the future tasks for execution. Hence, reusing initial tasks is of prime importance to reduce the reconfiguration overheads generated in a static system.

Only during a particular scenario, the above-proposed replacement policy fails in reducing reconfiguration overheads. Therefore, a slight modification is necessary to the above-proposed replacement policy. For example, consider a particular scenario, where only one task remains to be loaded in any of the available RUs and all RUs are already loaded with the configurations. In case of the above-proposed replacement policy applied to this particular scenario, it results in the generation of time reconfiguration overhead. This is because the proposed replacement policy chooses the last (L) RU for the particular scenario. Meanwhile, the last RU may be busy in executing the previous task. Therefore, choosing the last RU makes it difficult to perform the task prefetching technique for the current task. To avoid this, a small modification is imposed. In this modification, the proposed replacement policy chooses L-1th RU during the particular scenario. Hence, the current task can be prefetched without waiting for the completion of the previous task execution.

The step-2 of CM focuses on the memory mapping. As the time and energy reconfiguration overheads are to be reduced, it is necessary to map the tasks appropriately in either HS or LE memories. Before giving an appropriate mapping, the characteristics of the memories used in the architecture must be analyzed. In the proposed architecture, two on-chip memories and one off-chip memory are used. Out of the two on-chip memories, one is optimized for high speed and the other is optimized for low energy. HS memory offers quick access time with more energy consumption in comparison with the LE memory. But, the HS memory's energy consumption is comparatively lower than the off-chip memory. Similarly, LE memory consumes lesser energy but produces an additional delay in comparison with the HS memory. But, the access time offered by the LE memory is much faster than the off-chip memory. These are the characteristics of the on-chip memories.

To achieve the proposal's objective, the majority of tasks (Present in a task graph) ought to be mapped inside the on-chip memories before execution. Because fetching tasks from off-chip memory generates serious overheads. If time reconfiguration overhead is the only concern, then it is sufficient to map all the tasks inside the HS memory. Mapping all tasks in HS memory increases the overall energy consumption. Similarly, all tasks cannot be mapped inside the LE memory, since some tasks may produce additional delay. So, it is important to identify the tasks that can be mapped in LE without producing additional delay, as well as the tasks that must be mapped in HS to reduce the overall execution time.

A task is considered as vital when its reconfiguration time by applying the prefetch heuristic (when simulated from LE) causes additional delay. Only the vital tasks are mapped in HS memory, and the other non-vital task are mapped in LE memory. This is followed by the CM strictly.

The CM gives an ideal mapping without considering the sizes of on-chip memories. Therefore, the CM may sometimes allocate more tasks exceeding the size of the memory. In reality, the sizes of on-chip memories are limited. Hence, the replacer present in the proposed architecture considers the size of each memory and gives a

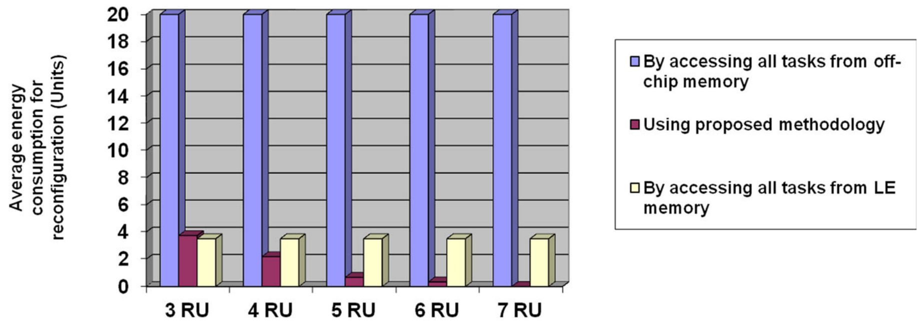


Fig. 3 Reduction in the energy reconfiguration overhead using our proposed architecture

suitable mapping. This mapping is also based on the individual task's vitality. The goal of the replacer is to replace the excess tasks present in one memory to other memory. By doing so, the delay experienced by the entire task graph execution process should be maintained to a minimum value. This can be achieved by particularly selecting the task that generates lesser delay from one on-chip memory (in comparison with the rest of the tasks occupying the same memory) and placing it on the other memory.

The CM and replacer mapping details are stored instantly in the information (info) table. These details can be used as the reference for the scheduler, and the applications can be executed successfully.

The characteristics of the HS and LE memories are obtained from CACTI tool [9] as shown in Table 1. A simulation environment is created to conduct the experiments. In our experiments, task graphs of multimedia application's benchmarks along with the thirty randomly generated task graphs are used.

The performance results are obtained for all the thirty-two task graphs individually executed for 1000 iterations. The results are given in Fig. 2. Figure 2 represents a graph between the average execution time and the number of RUs used. The ideal execution time and the execution time with full reconfiguration overheads are given in the graph for comparison. Ideal execution time is the execution time without any reconfiguration overhead. As seen in the graph, an average of 97% of time reconfiguration overheads are eliminated using the proposed architecture. The reduction in the time reconfiguration overheads is noticed because of the techniques used in combination with the proposed architecture. In the first iteration of application execution, memory mapping and task prefetching techniques are used. In the subsequent iterations, the proposed replacement policy is used in addition to the task prefetching and memory mapping. Thus, a significant amount of time reconfiguration overheads are eliminated.

A clear representation of reduction in the energy reconfiguration overhead is shown in Fig. 3 in the form of a graph. All the applications are executed for 1000 iterations individually, and their average value is given in the graph. It was observed that an average of 93.05% of energy reconfiguration overheads are totally eliminated. Without proper reusing of tasks, the proposed architecture can reduce the energy reconfiguration overheads by keeping all the tasks in LE memory. This is not possible in reality as the size of on-chip memories is restricted. By considering that all the tasks are kept in LE memory, an average reduction in the reconfiguration overhead obtained is 82.5%. By including proper reusing of tasks, the architecture reduces the energy reconfiguration overheads further by an average of 12.79% in comparison with the reduction achieved when all tasks are kept in LE memory.

References

1. Tessier R, Burleson W (2001) Reconfigurable computing for digital signal processing: a survey. J VLSI Signal Process Syst Signal Image Video Technol 28(1–2):7–27
2. Virtex-5 FPGA configuration user guide, Ug191(v3.10) (2011) Xilinx Inc., San Jose, CA, USA
3. Ramo EP, Resano J, Mozos D, Catthoor F (2007) Reducing the reconfiguration overhead: a survey of techniques. In: Proceedings of the international conference on ERSA, pp 191–194
4. Resano J, Mozos D, Catthoor F (2005) A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware. In: Proceedings of the conference on design, automation and test in Europe, vol 1. IEEE Computer Society, pp 106–111
5. Li Z, Hauck S (2002) Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. In: Proceedings of the 2002 ACM/SIGDA tenth international symposium on field-programmable gate arrays. ACM, pp 187–195
6. Li Z, Compton K, Hauck S (2000) Configuration caching management techniques for reconfigurable computing. In: 2000 IEEE symposium on field-programmable custom computing machines, pp 22–36
7. Clemente JA, Resano J, González C, Mozos D (2011) A hardware implementation of a run-time scheduler for reconfigurable systems. IEEE Trans Very Large Scale Integr Syst 19(7):1263–1276
8. Enemali G, Adetomi A, Arslan T (2017). FAReP: fragmentation-aware replacement policy for task reuse on reconfigurable FPGAs. In: 2017 IEEE international parallel and distributed processing symposium workshops (IPDPSW), May 2017, pp 202–206

9. Clemente JA, Ramo EP, Resano J, Mozos D, Catthoor F (2014) Configuration mapping algorithms to reduce energy and time reconfiguration overheads in reconfigurable systems. IEEE Trans Very Large Scale Integr Syst 22(6):1248–1261
10. Clemente JA, Gran R, Chocano A, del Prado C, Resano J (2016) Hardware architectural support for caching partitioned reconfigurations in reconfigurable systems. IEEE Trans Very Large Scale Integr Syst 24(2):530–543
11. Hariharan I, Kannan M (2018) Reducing reconfiguration overheads of a reconfigurable dynamic system using active run-time prediction. J Electr Eng 18(2):349–356
12. Yoosefi A, Naji HR (2017) A clustering algorithm for communication-aware scheduling of task graphs on multi-core reconfigurable systems. IEEE Trans Parallel Distrib Syst 10:2718–2732
13. Kanemitsu H, Hanada M, Nakazato H (2016) Clustering-based task scheduling in a large number of heterogeneous processors. IEEE Trans Parallel Distrib Syst 27(11):3144–3157
14. Morales-Villanueva A, Kumar R, Gordon-Ross A (2016). Configuration prefetching and reuse for preemptive hardware multitasking on partially reconfigurable FPGAs. In: Proceedings of the 2016 conference on design, automation and test in Europe. EDA consortium, March 2016, pp 1505–1508
15. 7 Series FPGAs Overview, DS180 (v1. 11) (2012) Xilinx, San Jose, CA, USA
16. ZC702 Evaluation Board for the Zynq-7000 XC7Z020 Extensible Processing Platform, User Guide, UG850 (v1. 0) (2012) Xilinx, San Jose, CA, USA
17. Altera (2011) Stratix V Device Datasheet, San Jose, CA, USA (online). http://www.altera.com/literature/hb/stratix-v/stx5_53001.pdf
18. Altera (2011) Quartus II Handbook Version 13.0, vol 1: design and synthesis, San Jose, CA, USA (online). http://www.altera.com/literature/hb/qts/quartusii_handbook.pdf
19. AXI Reference guide, UG761 (v13. 1) (2011) Xilinx, San Jose, CA, USA
20. Local Memory Bus (LMB) v1. 0 (v1. 00a) (2005) Xilinx, San Jose, CA, USA

Journal of Circuits, Systems, and Computers
Vol. 28, No. 14 (2019) 1950246 (34 pages)
© World Scientific Publishing Company
DOI: 10.1142/S0218126619502463

Efficient Use of On-Chip Memories and Scheduling Techniques to Eliminate the Reconfiguration Overheads in Reconfigurable Systems*

I. Hariharan† and M. Kannan‡

Department of Electronics Engineering,
MIT Campus, Anna University, Chennai 600044,
Tamil Nadu, India
†*hariharan166@gmail.com*
‡*mkannan@annauniv.edu*

Received 12 June 2018
Accepted 17 January 2019
Published 15 March 2019

Modern embedded systems are packed with dedicated Field Programmable Gate Arrays (FPGAs) to accelerate the overall system performance. However, the FPGAs are susceptible to reconfiguration overheads. The reconfiguration overheads are mainly because of the configuration data being fetched from the off-chip memory at run-time and also due to the improper management of tasks during execution. To reduce these overheads, our proposed methodology mainly focuses on the prefetch heuristic, reuse technique, and the available memory hierarchy to provide an efficient mapping of tasks over the available memories. Our paper includes a new replacement policy which reduces the overall time and energy reconfiguration overheads for static systems in their subsequent iterations. It is evident from the result that most of the reconfiguration overheads are eliminated when the applications are managed and executed based on our methodology.

Keywords: Configuration mapping; field-programmable gate array; multimedia application; optimal-replacement policy; reconfiguration overheads; scheduling.

1. Introduction

New generation embedded devices are very complex. They include functionalities that are initially developed for general purpose platforms such as multimedia support which includes sound processing, texture rendering, image and video displaying, etc. In fact, the modern portable devices have inherited the area and energy constraints[1] of embedded systems, and at the same time, they must achieve the performance required by multimedia applications. The best way to meet all these constraints is by

*This paper was recommended by Regional Editor Tongquan Wei.
† Corresponding author.

providing hardware support which can simultaneously speed up the execution and reduce the energy consumption. Usually, application specific integrated circuits (ASICs) are used for these purposes. However, three main drawbacks prevent their use as a general solution. First, the area available in an embedded/portable device is limited. Hence, only important functionalities can be migrated to the hardware. Second, their time-to-market is very high. And the third is their fixed functionality which cannot be updated in the future (even if the system has sufficient scope to improve).

One attractive option to overcome these limitations is to include reconfigurable hardware resources like field-programmable gate array (FPGA).[2–4] In FPGA, run-time partial reconfiguration[5–8] reuses the same hardware for executing different functionalities. This feature fulfills our area constraint. Secondly, their time-to-market is comparatively very low than ASIC. And finally, the flexibility of the FPGA is superior without compromising much on their performance side.

As the FPGA resources are comparatively less to more extensive applications, it is not always possible to accommodate full configurations (Required for the specific application) in the FPGA. Sometimes this happens even for a static system. A static system is the one in which either one application or a group of applications (executed in a similar fashion) is active at any time. To overcome the limitation of lesser availability of FPGA resources, the entire configuration is divided into many sub-parts. The divided configurations are managed and executed using the run-time partial reconfiguration process. This feature helps FPGA to run the entire application with limited resources. In the run-time partial reconfiguration process,[9] a portion of the FPGA is loaded with the configurations (Needed for future execution) during which the remaining part of the FPGA is still executing its already loaded application. This is also called a dynamic partial reconfiguration process.

One major disadvantage of FPGA which takes all its merits is the generation of reconfiguration overheads during the dynamic partial reconfiguration process. This generation of reconfiguration overheads during the run-time partial reconfiguration process is mainly because of the improper management of configurations and the available FPGA resources. Also, the configurations are fetched from off-chip memory during run-time partial reconfiguration. Hence, the time (typically in the order of hundreds of milliseconds)[10] and energy[11] required for fetching the configuration from off-chip memory are also the sources of undesired reconfiguration overheads generated. Fetching the configuration from off-chip memory is becoming a serious concern only during the last decade.[12] This is because, over the past few years, the gap between the off-chip memory bandwidth and the system computing capabilities has been increasing significantly. At the same time, on an average, on-chip memory access with current technology costs 250 times lesser energy consumption per bit than an off-chip one.

The reconfiguration overheads have to be eliminated from the reconfigurable system to improve its performance[13] and it is not noticed in ASICs.[14,15] Thus the paper under discussion aims at making FPGA as an efficient alternative to ASIC.

To reduce the reconfiguration overheads, some techniques are already proposed and many of them are discussed. Authors of papers[16,17] use multi-context FPGAs. In which, several configurations are available in parallel as context and depending upon the user interrupt the switching between these configurations takes place. This switching takes only a few nanoseconds. However, this type of implementation of the application requires very high area. Walder and Platzner present[18] an environment for executing hardware tasks. In this work, the primary objective is to partition the FPGA with the optimal size of Reconfigurable Units (RUs) for every specific task's occurrence. Once a particular portion of the FPGA is allocated for a specific task, it must be provided with suitable communication infrastructure. This process of assigning communication infrastructure is complex and also time-consuming. Hence, in our methodology, we use RUs of same sizes having fixed communication infrastructure. In Ref. 19, many reconfiguration controllers are used to perform reconfigurations in parallel. However, most of the FPGAs are having single reconfiguration circuitry and therefore parallel reconfigurations are not possible without introducing multiple reconfiguration controllers which occupies a definite amount of space in the FPGA. Papers[20,21] efficiently utilize the data parallelism by considering different possibilities for the same task graph at design-time. Also, based on the current scenario at run-time, the scheduler is free to select a particular version of the task graph which is most suited for the current scenario. Our methodology also includes this technique to schedule the configurations efficiently. Authors of papers[22–24] discuss some of the compression algorithms to accelerate the reconfiguration process. These algorithms use the bit-stream compression technique to compress the configuration data. This technique aims at reducing the number of configuration fetches required. However, one disadvantage is that the compressed configuration data has to be decompressed before reconfiguring them into the target FPGA. The decompression process consumes a certain amount of energy as well as it generates some delay which again results in the generation of overheads. Another important technique in reducing reconfiguration overheads is the configuration prefetch technique. This technique involves in the pre-loading of future configuration when the current configuration is in the execution phase. Using this technique alone, a significant amount of reconfiguration overheads is reduced and it can be noticed in Refs. 25 and 26. In most cases, it is not possible to eliminate the reconfiguration overheads using the prefetching technique alone. This is because some of the tasks still generate overheads even if it is prefetched.

Even after prefetching, those tasks that generate overheads are considered vital. They need to be reused to eliminate their reconfiguration overheads (In future iterations of the same task graph execution). Hence, reusing the already loaded configurations[26–28] is one of the best ways for reducing the reconfiguration overheads. Authors of Refs. 29–31 got significant results by considering both the prefetch heuristic and reuse technique. Where Refs. 29–31 used different optimal replacement policies to improve the system performance. Even though better results are achieved by combining both the prefetch and reuse techniques, the performance achieved by the application is not optimal. Therefore, the authors of the paper[32] provided a memory hierarchy which consists of two levels of memories. These two levels consist of on-chip and off-chip. On-chip memory is divided into high speed (HS) and low energy (LE). Additionally, the authors proposed two configuration mapping algorithms specifically for static and dynamic systems, respectively, which end up in reducing both the energy and time reconfiguration overheads. In paper,[33] the authors extended the traditional configuration caching approaches by dividing the configuration into blocks. Thus, the granularity of the configurations is reduced and managed efficiently to reduce the reconfiguration overheads. Many of the works discussed above are most suited for systems whose functioning (At run-time) can be fully or partially predicted during the design-time phase. However, authors of paper[34] worked on a dynamic system whose nature of their application execution is entirely unpredictable at design-time. To overcome the issue, they developed algorithms where the prediction is also dynamic. Based on the prediction, future tasks are systematically prefetched and loaded inside the on-chip memories. The main advantage of this approach is that the time reconfiguration overheads are reduced significantly. However, concerning energy consumption, it costs much. This is because; the tasks are continuously predicted and scheduled at every instant in their run-time phase. Authors[35] proposed a power efficient configuration cache structure to reduce the overheads generated due to the configuration memory explicitly. To achieve the objective, two design schemes were implemented. Also, they are reusable context pipelining architecture and dynamic context management strategy. This proposal is exciting as it does not degrade the performance and the flexibility of coarse-grained reconfigurable architectures despite achieving minimum power. Authors[36] analyzed the importance of thermal characteristics and how it affects in designing 3D processors. They exhaustively reviewed works on optimization techniques like memory management and task scheduling to efficiently construct 3D processors. This is an interesting paper, where the authors suggest that the 3D processors are the future when their thermal impacts are efficiently managed.

Our paper tries to reduce the reconfiguration overheads generated in a static system. A new replacement policy combined with the memory mapping and prefetch heuristic makes the objective achievable. The proposed replacement policy does not aim at reusing a maximum number of tasks. However, when there is a possibility for task reuse, it tries to reuse the vital RUs. Most of the RUs holding the initial tasks are considered vital. Hence, they must be reused by our proposed methodology. A slight modification to the proposed replacement policy is needed in certain cases. For example, consider a particular scenario, where only one task is remaining to get loaded in any of the available RU. However, all the RUs are already loaded or being loaded with the configurations. In this case, when the proposed replacement policy is implemented, then it generates additional time reconfiguration overhead. Because reusing the last (L) RU makes it harder to perform the prefetching technique. To avoid this, a small modification is imposed on the proposed replacement policy which reuses the $L-1$th RU only at this particular scenario.

For task prefetching, as soon as possible (ASAP) is used. In the case of memory mapping, first, the vitality of the individual tasks is found. For calculating the vitality of a task, hiding value concept is introduced in this paper. Hiding value gives the exact amount by which the current task can be hidden under the shadow of the previous task's ideal execution time. Based on the hiding value, the task is either kept in HS or LE memory. The task is kept inside the HS memory if its corresponding hiding value calculation gives a negative value. Similarly, a positive hiding value indicates that the corresponding task can be kept in LE memory. Combinations of the techniques mentioned above presented uniquely to form the proposed methodology. Also, this novel methodology offers very high performance with reduced energy consumption.

Major contributions of this paper

(1) This paper includes three essential techniques to reduce reconfiguration overheads generated in a reconfigurable system. They are efficient memory mapping, task prefetching, and efficient task reusing techniques.

(2) Using efficient memory mapping, the vital tasks are placed inside the HS memory and the other nonvital tasks are placed inside the LE memory. The individual task's vitality is entirely based on the hiding value calculation proposed in the methodology.

(3) For task prefetching, ASAP approach is used. It means when the current task starts its execution then at the same time the next task is fetched and reconfigured in the available RU.

(4) Some of the vital RUs are found and the tasks allocated in those RUs are retained. This, in turn, reduces the reconfiguration overheads considerably. In this proposal, the proposed replacement policy is novel and it tries to achieve the objective of this paper when there is a possibility for task reuse.

The rest of this paper is organized as follows: Section 2 briefly describes the target architecture. Section 3 describes the proposed methodology. Section 4 briefly discusses the results and discussions.

2. Target Architecture

The target architecture shown in Fig. 1 is considered as the reference architecture. This architecture can be realized in any of the last generation FPGAs, such as Xilinx

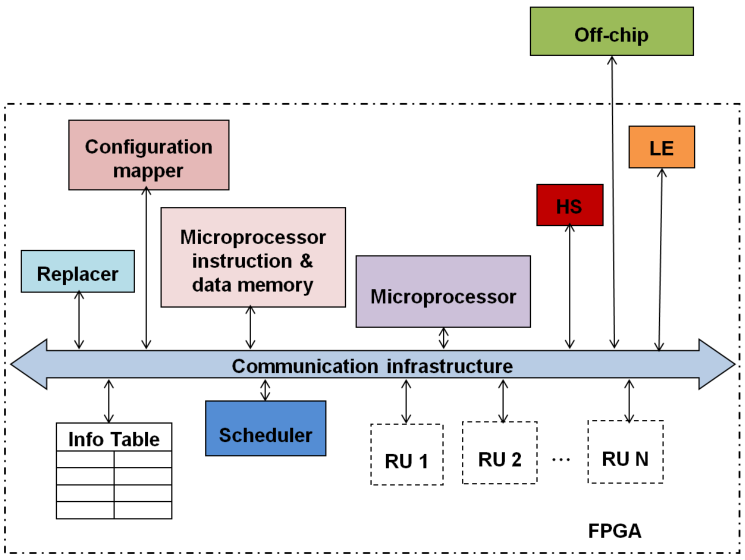


Fig. 1. Target architecture.

Virtex – 7 and Zynq – 7000 EPP devices,[37,38] or its equivalent in Altera FPGAs.[39,40] The communication infrastructure may use a network-on-a-chip (NOC) model, or it may be implemented using one or several buses. Xilinx provides some of the communication infrastructures like Advanced Extensible Interface communication infrastructure[41] which is used to connect the computational cores and the local memory bus (LMB)[42] for connecting the memories.

The proposed architecture includes two on-chip and one off-chip memory. Out of the two on-chip memories, one is optimized for HS and the other is optimized for LE. HS memory has quick access time but involves additional energy consumption in comparison with the LE memory. However, the HS memory's energy consumption is very low than off-chip memory. Similarly, LE memory has lesser energy consumption but produces an additional delay in comparison with the HS memory. The access time offered by the LE memory is much faster than the off-chip memory. These are the characteristics of the on-chip memories developed in today's market.

The on-chip memories are similar to the configuration cache. They are not real cache, but SRAMs controlled by software (i.e., a scratchpad). If these memories are used properly, they improve the system performance without affecting the overall energy consumption of the system, since they prevent the use of high-capacitance off-chip bus.[43] Only the vital task (additional delay in their execution is not acceptable) is placed inside the HS memory. Other, nonvital tasks (additional delay in their execution is completely acceptable) are kept inside the LE memory. By taking advantage of this flexibility, the energy consumption of the system can be controlled despite achieving the maximum performance.

Task	Hiding value	Memory to map	RU to load
1	***	HS	RU 1
2	16	LE	RU 2
3	27	LE	RU 3
4	15	LE	RU 2

Fig. 2. Info table details for JPEG task graph.

Initially, the applications are modeled as task graphs (TGs) and stored in the off-chip memory. Based on the task graph and the available number of RUs, the configuration mapper (CM) decides the configurations to be placed inside the HS and LE memories. In addition to mapping, it also gives the pattern of the schedule to be followed in the first iteration. Hence, the vital configurations are reused in the future iterations of the same task graph execution.

The critical drawback of using on-chip memories is their limited size. For example, the Virtex-5 FPGA (XUPV5-LX110T) from Xilinx development board contains only 6448 kB of on-chip memory[44] and 256 MB of the external Double Data Rate (DDR2) memory that can be attached to it. Similarly, high-end FPGAs has less than 50 MB of on-chip memory (not sufficient for larger reconfigurable regions). Therefore, the replacer takes the amount of on-chip memory space into account and decides the final mapping of configurations in respective memories of HS, LE, and off-chip. All these mapping results are stored in the info table, a kind of memory as shown in Fig. 2 so that it can be easily accessed and scheduled. The scheduler prefetches the configuration or task from the available memory and maps it into the RU by partial reconfiguration. RU is the smallest portion in an FPGA reserved for loading the configuration. After loading the configuration, the execution of that task is performed. This architecture is efficiently utilized by the proposed methodology to reduce the reconfiguration overheads.

3. Proposed Methodology

The main aim of the proposed methodology is to reduce the reconfiguration overheads generated in a reconfigurable system. Hence to achieve this motive, techniques like prefetching, memory mapping, and reusing of tasks are included in the proposed methodology. The methodology is divided into two phases namely the design-time phase and a run-time phase. Almost all the calculations and analysis are performed during the design-time phase. Based on the decisions taken during the design-time phase, the execution of the application is carried in the run-time phase.

Initially, all the applications are modeled using a two-level hierarchy[45] as shown in Fig. 3. At the top level, the applications are represented as a set of task graphs. Each task graph in their lower level consists of several tasks that show limited dynamic

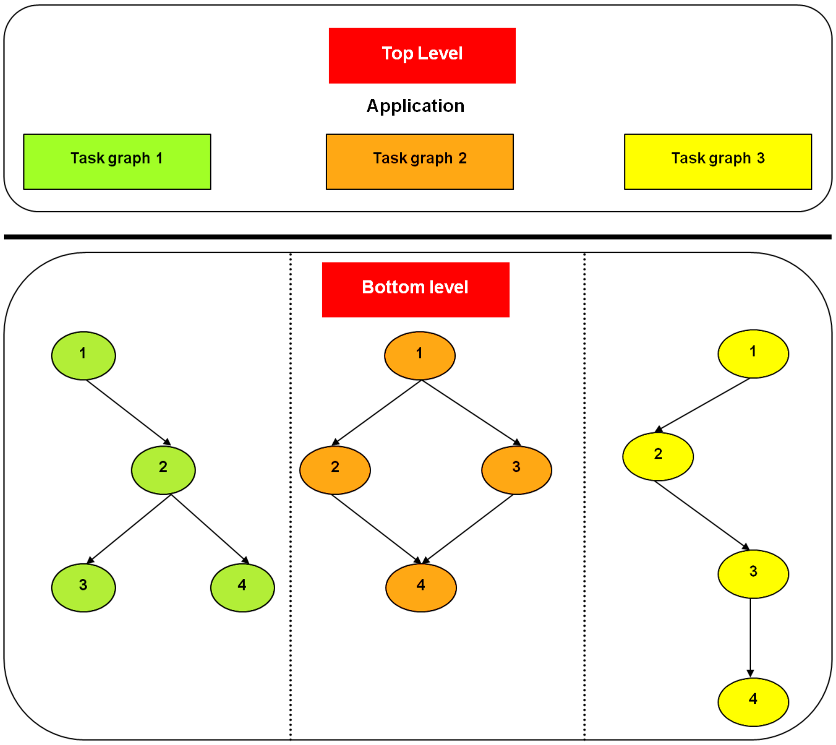


Fig. 3. Combination of task graphs forming an application.

behavior between them. Any task within the task graph is the basic scheduling unit loaded in individual RUs.

It is not always possible to accurately predict the execution time of the task graph at design-time phase. Because the execution time of a task graph depends on the input data also. To overcome this limitation, different possibilities of combinations of task graph execution scenarios are considered. They are converted into appropriate parallel task graphs. These task graphs form the inputs for the proposed methodology and they will be analyzed and processed to schedule efficiently.

3.1. *Design-time phase*

Useful information from the task graph is extracted during the design-time phase. Even complex task graphs can be scheduled using our methodology. However, the first step is to assign priority to individual tasks occupying the same level in a task graph. This is because most of the FPGAs are having single reconfiguration circuitry. Reconfiguration circuitry is the dedicated hardware unit present inside an FPGA responsible for the dynamic reconfiguration process. Hence in the case of several tasks occupying the same level, there comes competition for reconfiguration.

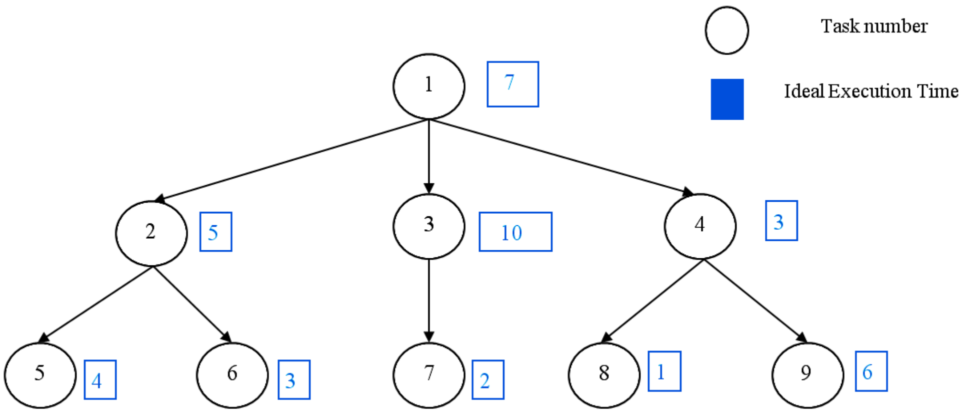


Fig. 4. Example of a parallel task graph.

The task must be selected with utmost caution so that the reconfiguration of the selected task does not generate additional time overhead. For this reason, a weight based ranking system is followed.

Consider a task graph shown in Fig. 4. This task graph consists of three levels. Out of the three, two levels consists of multiple tasks. At each level of the task graph, a separate ranking is given for individual tasks. This ranking is based on the weights of individual tasks. The weight assignment followed in this methodology mainly focuses on the critical path, so that there will not be any additional overhead generated in the task graph execution process.

The weight assignment for any task is performed as follows: For leaf task, the weight is only their ideal execution time. Ideal execution time is the execution time of a task without any reconfiguration overhead. Also, for other tasks having branches, the weight of any task is the sum of their ideal execution time and the maximum weight of their immediate successive tasks. If two tasks consist of same weight occupying the same level, then their order of execution is decided based on the task number, i.e., the task that comes first in the task graph is given higher priority. Based on the methodology, Fig. 4 is converted to Fig. 5 for efficient processing. The task with the top rank is first reconfigured using the proposed methodology.

The converted task graph is placed inside the off-chip memory. This converted task graph constraint holds only for the reconfiguration process and this will not affect the execution pattern of the task graph.

The proposed architecture consists of HS and LE on-chip memories. A critical function of the proposed methodology is to fully utilize these available memories, thereby reducing most of the reconfiguration overheads. If time reconfiguration overhead is the only concern, then it is sufficient to keep all the tasks inside the HS memory. Keeping all the tasks in HS memory increases the overall energy consumption. It is also not possible to keep all the tasks inside the LE memory, because

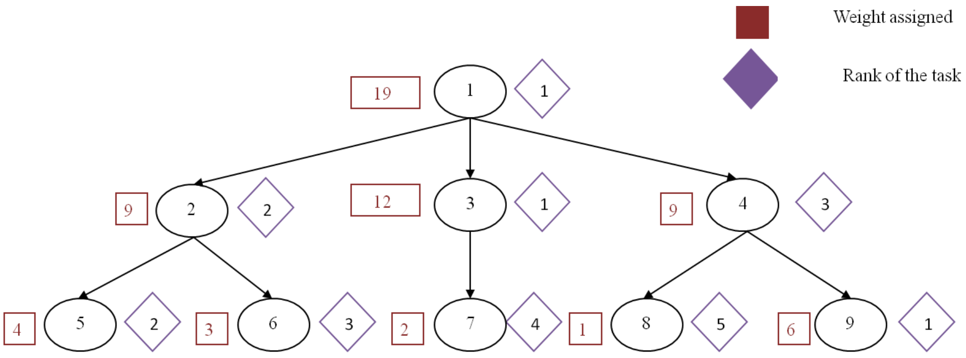


Fig. 5. Equivalent task graph for efficient processing.

some tasks may produce additional delay. So, the proposed methodology identifies the tasks that can be kept in LE without producing any additional delay, as well as the tasks that must be kept in HS to reduce the overall execution time. To further reduce the reconfiguration overheads prefetching and reusing of tasks is also included in the proposed methodology.

3.1.1. *Configuration mapper*

Configuration Mapper (CM) analyzes every task graph and gives an appropriate mapping of configurations onto the HS and LE memories. CM provides an ideal mapping (irrespective of the available memory size). It also tries to give the pattern of the schedule to be followed for any task graph. So the same task graph in their future iterations can reduce some of the reconfiguration overheads by configuration reuse.

The input for the CM is the task graph with their task execution times and the available number of RUs. For any task from a task graph, the first step is to assign a specific RU from the available number of RUs. After assigning the RU, the same task is checked for the memory mapping. The task when fetched from LE can be made hidden under the shadow of previous task's execution time by applying prefetch heuristic; then the same task can be kept in LE memory to minimize the energy consumption. On the contrary, the same task must be kept in the HS memory to avoid any delay. These principles are followed in assigning tasks to HS and LE memories in step-1, step-2, and, step-3 of our methodology.

3.1.1.1. Step-1

Assume the first task need to be configured in the first available RU. For the first task, there is no previous task, so it cannot be made hidden. Hence, assign the first task ($n = 1$) to HS memory. For future tasks prediction, current task assignment is updated using the following equations:

$$\text{Simulated Reconfiguration Time for the first task } (\mathrm{RT}_1) = \mathrm{rt}_1\,. \tag{1}$$

$$\text{Simulated Execution Time for the first task } (\mathrm{ET}_1) = \mathrm{RT}_1 + \mathrm{et}_1\,, \tag{2}$$

where rt_1 = First task's reconfiguration time being fetched from HS memory,
et_1 = First task's ideal execution time,
n = Current task.

3.1.1.2. Step-2

For deciding the remaining tasks to be kept either in HS or LE memory, a separate procedure is followed and is shown in Fig. 6. From the second task to ith task, the hiding value (V) is calculated using Eq. (3). Where 'i' is the total number of RUs available.

$$V_n = \mathrm{ET}_{n-1} - (\mathrm{RT}_{n-1} + \mathrm{rt}_n(\mathrm{LE}))\,, \tag{3}$$

where V_n is the hiding value of the nth task,
ET_{n-1} is the simulated execution time of the $(n-1)$th task,
RT_{n-1} is the simulated reconfiguration time of the $(n-1)$th task,
$\mathrm{rt}_n(\mathrm{LE})$ is the reconfiguration time of the nth task when it is fetched from the LE memory.

Depending upon the hiding value, the task is assigned in respective memories. If V is positive, then assign to LE and when it is negative, assign to HS. After assigning each task into their respective memories, their simulated reconfiguration and simulated execution time are updated using the following equations:

$$\mathrm{RT}_n = \begin{cases} \mathrm{RT}_{n-1} + \mathrm{rt}_n(\mathrm{HS}), & \text{if } V_n \text{ is negative} \\ \mathrm{RT}_{n-1} + \mathrm{rt}_n(\mathrm{LE}), & \text{if } V_n \text{ is positive} \end{cases}\,, \tag{4}$$

$$\mathrm{ET}_n = \mathrm{Max}\{\mathrm{ET}_{n-1}(\text{or})\ \mathrm{RT}_n\} + \mathrm{et}_n\,, \tag{5}$$

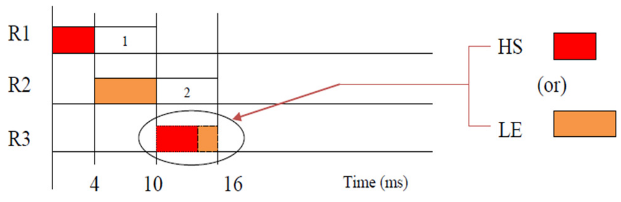
```
# From 2nd to ith task
for 2 to i
      n = n+1;
      calculate $V_n$ using equation (3);
      if $V_n$ is positive
           assign nth task in LE;
       else
           assign nth task in HS;
       end if;
       update using equations (4) and (5);
end for;
```

Fig. 6. Memory mapping from second to ith task.

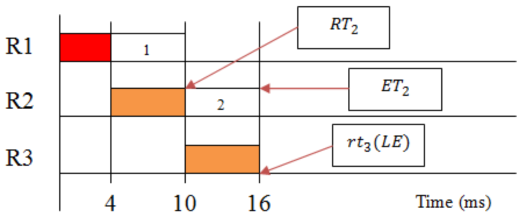
where RT_n is the simulated reconfiguration time of the nth task,
$\mathrm{rt}_n(\mathrm{HS})$ is the reconfiguration time of the nth task when it is being fetched from the HS memory.
ET_n is the simulated execution time of the nth task,
et_n is the ideal execution time of the nth task.

Hiding value gives the exact worth of a task to be kept in either of the on-chip memories. For calculating the hiding value (V), one should know the specific RU allocated to execute the current task and also the previous task's simulated or updated reconfiguration time and execution time. The concept of calculating hiding value is explained in Fig. 7. In Fig. 7(a), the third task has to be kept in either HS or LE. So, in Fig. 7(b) the third task is assumed to be kept in LE and reconfigured (using prefetch technique). If the reconfiguration did not introduce any delay, then the same task can be kept in LE. Otherwise, it should be placed in HS memory. Our Eqs. (3) and (9) does the same work. The main aim of the hiding value is to check whether the current task's reconfiguration time (Fetched from LE using prefetch) is made hidden under the shadow of the previous task's ideal execution time. In Fig. 7(b), the third task when loaded from LE is made hidden under the shadow of the second task's ideal execution time. When hiding value (V) is calculated for the third task using Eq. (3), it is zero. Hence, the third task is kept inside the LE memory by our configuration mapper.

A task graph contains a large number of tasks. It is sometimes impossible to assign all the tasks in the available RUs without doing any replacement. Because in most cases, the available number of RUs has always been lesser than the number of tasks



(a)



(b)

Fig. 7. Hiding value calculation.

in any task graphs. When the available number of RUs is already assigned and if there is still some new tasks available to get executed, then in order to assign the new task in any of the RU, a replacement policy is followed. This replacement policy in the proposed methodology (discussed in step-3) was intended to optimally reuse the already loaded tasks to reduce the reconfiguration overheads further.

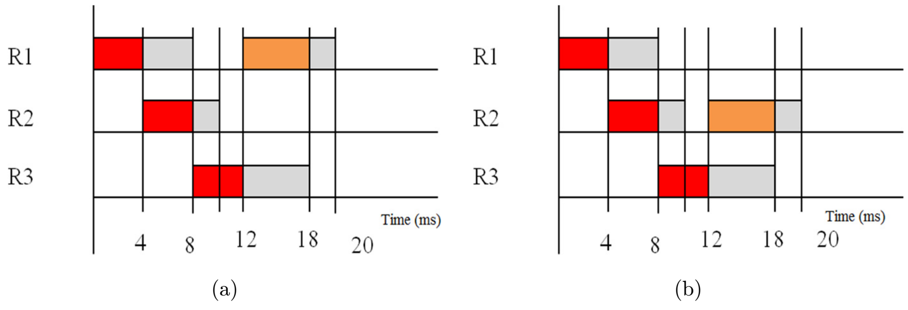
3.1.1.3. Step-3

A replacement scenario is the one in which all the available RUs are already assigned with the configurations and still there are one or more tasks to get loaded. In this scenario, the proposed methodology follows either least recently used (LRU) policy or newly proposed replacement algorithm in selecting the RU for task loading. This is decided based on the remaining number of tasks available and the total number of RUs present. The algorithm for this is shown in Fig. 10.

In the proposed methodology, the maximum reusing of tasks is not the ultimate aim. However, optimal reusing of RUs is preferred to reduce the reconfiguration overheads. For this purpose, the proposed methodology aims at reusing most of the initial tasks of the application. This is because first few tasks account for the significant reconfiguration overheads if it is not being reused. For example, even though performing task prefetch, the first task generates overheads if it is not being reused. Therefore it is reused compulsorily. Another advantage of using this technique is that future tasks can be prefetched at the time of the execution of initial tasks. This happens in the subsequent iterations of the same application execution. This, in turn, reduces the reconfiguration overheads.

Sometimes implementing the above-proposed replacement technique (reusing most of the initial tasks) results in additional reconfiguration overheads. For example, consider a situation, where only one task is remaining in the replacement scenario. At this point assume there is 'i' number of RUs available to replace. If the above-proposed replacement technique is followed in this situation, then it involves in the generation of additional reconfiguration overheads. It can be noticed in Fig. 8(c). Figure 8(c) exactly matches with the scenario under discussion but to be more specific it follows most recently used (MRU) policy after the occurrence of the replacement scenario. To avoid this kind of overhead generation, a slight modification to the proposed replacement technique is imposed. Hence, the proposed methodology in this situation chooses the $(i-1)$th RU. Because ith RU is executing the previous task. When the same RU is chosen for the current task, then it will create additional overhead and the scenario is clearly explained in Figs. 8(c) and 9(c). This overhead is eliminated with our slight modification and can be noticed in Figs. 8(b) and 9(b).

A simple comparison of the proposed replacement policy with other existing policies is shown in Figs. 8 and 9. As seen from the figure, both LRU and optimal replacement policy outperform the MRU policy in their first iteration. However, in the second iteration when the tasks are reused LRU suffers from overheads because of



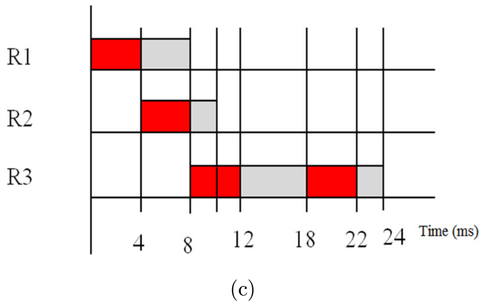
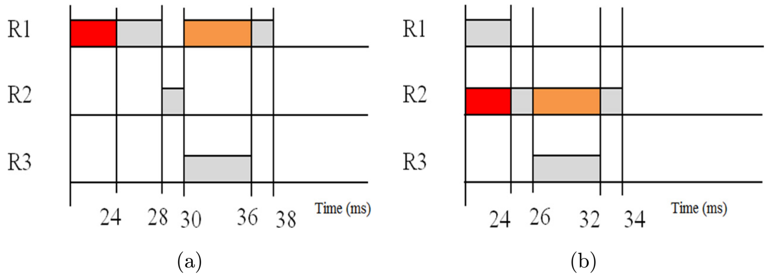


Fig. 8. Replacement scenario with one task remaining to get executed. (a) Applying LRU, (b) applying optimal replacement policy and (c) applying MRU.



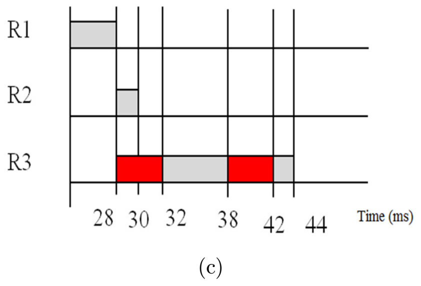


Fig. 9. The second iteration of the scheduling. (a) Applying LRU, (b) applying optimal replacement policy and (c) applying MRU.

```
#Proposed replacement algorithm

1  : for every task replacement scenario do
2  :         if (Number of remaining tasks to get executed ≥ Total number of RUs)
3  :             follow LRU;
4  :         else
5  :             follow optimal replacement policy;
6  :         end if;
7  : end for;
```

Fig. 10. Algorithm for replacement scenario.

not reusing the initial task. Optimal replacement policy in their second iteration shows lesser overheads than LRU which is shown in Fig. 9.

The proposed methodology makes sure that for every replacement scenario all the existing RUs are replaced with the new configuration before it goes to the next replacement scenario. This unique approach makes the proposed methodology not greedy enough for reusing more tasks. However, optimally reuses the vital RUs (if reuse is possible).

The algorithm used at the time of replacement scenario shown in Fig. 10 is explained. During the task replacement scenario, the condition from line 2 of Fig. 10 is checked. If the condition is found to be true, then the LRU algorithm is followed. This is because none of the RUs can be skipped from loading new tasks since the number of new tasks available is equal to or greater than that of the available RUs. The LRU algorithm is given in Fig. 11. Similarly, if line 2 of Fig. 10

```
# LRU

for 2 to s
      for 1 to i
            n = n+1;
            $m = m + 1$;
            calculate $V_n$ using equation (9);
            if $V_n$ is positive
                assign $n^{th}$ task in LE;
            else
                assign $n^{th}$ task in HS;
            end if:
            update using equations (10) and (5);
      end for;
end for;
```

Fig. 11. Algorithm for LRU.

```
# Finding the RU, 'a'

1: Compute $a$;
2: if $((a+1)^{th}\ RU\ \ is\ \ not\ \ MRU)$,
3:       $a = a+1$;
4: else
5:       $a = a$;
```

Fig. 12. Algorithm for finding the value 'a'.

is found false, then the optimal replacement algorithm is followed, which is given in Fig. 13.

Before performing LRU, it is necessary to calculate the value 'S'. 'S' is the total number of replacement scenarios to be executed using LRU for any task graph. It is calculated using Eqs. (6)–(8). In Eq. (6), the total number of tasks is divided by the number of RUs available and the remainder is stored. After calculating 'S', the value of 'm' is initialized to zero. This initialization is necessary because the variable 'm' used in our algorithm holds the specific RU allocated for a particular task. Because before calculating the hiding value for any task, the first step of our proposed

```
# Optimal Replacement
1   :  n = n+1;
2   :  calculate $V_n$ using equation (9);
3   :  if $V_n$ is positive
4   :      assign nth task in LE;
5   :  else
6   :      assign nth task in HS;
7   :  end if;
8   :  update using equations (10) and (5);
9   :  calculate b using equation (15);
10 :  for 1 to b
11 :        n = n+1;
12 :        m = m+1;
13 :        calculate $V_n$ using equation (9);
14 :        if $V_n$ is positive
15 :            assign nth task in LE;
16 :       else
17 :            assign nth task in HS;
18 :      end if;
19 :      update using equations (10) and (5);
20 : end for;
```

Fig. 13. Proposed replacement algorithm.

algorithm is to allocate a specific RU for the particular task.

$$O = \text{Remainder}(K, i)\,, \tag{6}$$

where K is the Total number of tasks in a task graph,
i is the number of RUs, available

$$W = K - O\,, \tag{7}$$

$$S = W/i\,. \tag{8}$$

Figure 11 gives the LRU algorithm followed during the replacement scenario. In which, a 'for-loop' is created from the second to the Sth replacement scenario. In every replacement scenario, there will be 'i' number of tasks. The hiding value for each task is calculated using Eq. (9). Since it is a replacement scenario Eqs. (3) and (4) are modified to (9) and (10), respectively. This modification is necessary to accurately predict the hiding value of a task after the replacement scenario. Because after the replacement scenario, the current task's hiding value is affected by both the previous task's reconfiguration time and the execution time of the prior task (before the occurrence of the current replacement scenario) which is already executed on the allocated RU. Every task will have its own 'V', based on which the task is either assigned to HS or LE memories.

$$V_n = \text{ET}_{n-1} - (\text{Max}\{\text{ET}_m \text{ (or) } \text{RT}_{n-1}\} + \text{rt}_n(\text{LE}))\,, \tag{9}$$

$$\text{RT}_n = \left\{ \frac{\text{Max}\{\text{ET}_m \text{ (or) } \text{RT}_{n-1}\} + \text{rt}_n(\text{HS}),\ \text{When } V_n \text{ is negative}}{\text{Max}\{\text{ET}_m \text{ (or) } \text{RT}_{n-1}\} + \text{rt}_n(\text{LE}),\ \text{When } V_n \text{ is positive}} \right. . \tag{10}$$

If line-2 of Fig. 10 is found false, then the proposed replacement algorithm is followed and it is given in Fig. 10. Before performing the proposed replacement, it is necessary to calculate the value of 'm'. To calculate 'm', the value 'a' is calculated first and its algorithm is given in Fig. 12. The value 'a' is the difference between the total number of RUs available and the remaining number of tasks to get executed. After finding 'a', line 2 of Fig. 12 checks for the condition and if it is found to be true, then $(a + 1)$ is stored in 'a'. Similarly, if it is found to be false, then 'a' is kept as such. If $(a + 1)$th RU is the MRU RU, then by replacing this RU it is tough to hide the reconfiguration latency. It is clearly explained in Figs. 8(c) and 9(c). After calculating the value 'a', the exact value of 'm' is found using Eqs. (11) to (14).

$$r = K/i\,, \tag{11}$$

$$t = \text{fix}(r)\,, \tag{12}$$

$$h = (t * i) - i\,, \tag{13}$$

$$m = h + a\,. \tag{14}$$

After finding 'm', the proposed replacement algorithm can be performed as shown in Fig. 13. For the first task under the proposed replacement algorithm, the hiding value is calculated using Eq. (9) and the corresponding task is assigned to HS or LE based on the hiding value's sign. After assignment, it is updated using Eqs. (10) and (5). The value of 'b' is calculated using Eq. (15) and it gives the remaining number of tasks to get executed. For the remaining tasks, the allocation of RU is in the clockwise direction. After the RU allocation, the specific task is checked for hiding value. Based on the hiding value, the task is assigned to specific on-chip memory. This procedure is followed in Fig. 13 of lines 9–20.

$$b = K - n\,. \tag{15}$$

Information regarding the mappings provided by the CM is stored instantly (for every task present in a task graph) in the off-chip memory. This information is unique for each task graph and therefore it occupies a specific place in the off-chip memory. All this information can help the scheduler to schedule the configurations at run-time effectively.

3.1.2. *Replacer*

CM allocates the tasks present inside the task graph in either HS or LE memory based on the hiding value. The CM assumes that the on-chip memories have enough capacity. This is not true in the real world. Hence, the replacer considers the size of the HS and LE memories and gives a suitable mapping. The input for the replacer is the size of the on-chip memories and the CM output and is shown in Fig. 14.

The algorithm followed for replacer is given in Fig. 15. The goal of this algorithm is to replace the excess tasks present in one memory to other memory with minimum delay experienced in the entire task graph execution process. This is achieved by particularly selecting the task that generates lesser delay from one on-chip memory (in comparison with the rest of the tasks occupying the same memory) and placing it on the other memory.

Replacer algorithm mainly works on the hiding value. When the size of tasks present inside the HS memory exceeds the total size of HS, then the task (from HS memory) which is having the maximum hiding value (V) is chosen. The chosen task is assigned to LE memory and gets updated in the info table. This process is repeated until no more additional tasks are available in HS memory and it is given in Fig. 15.

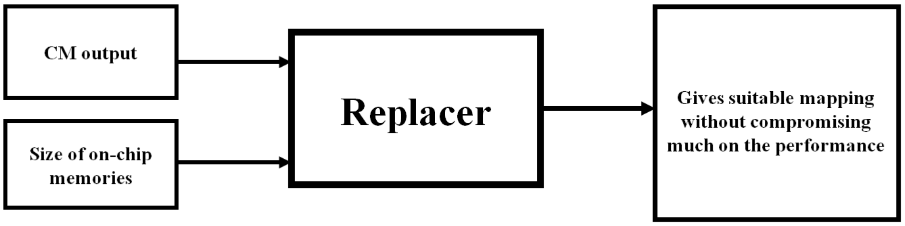

CM output
Size of on-chip memories
Replacer
Gives suitable mapping without compromising much on the performance

Fig. 14. Replacer block diagram.

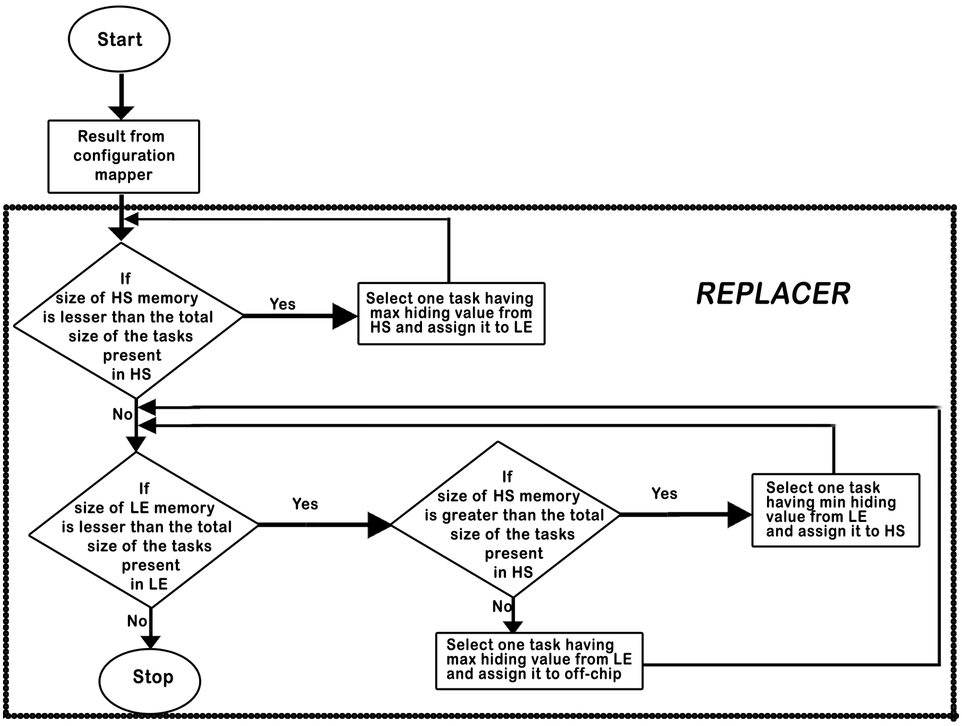
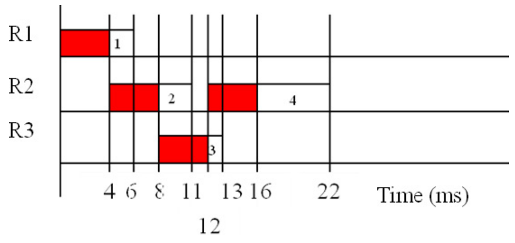

Start
Result from configuration mapper
REPLACER
If size of HS memory is lesser than the total size of the tasks present in HS
Yes
Select one task having max hiding value from HS and assign it to LE
No
If size of LE memory is lesser than the total size of the tasks present in LE
Yes
If size of HS memory is greater than the total size of the tasks present in HS
Yes
Select one task having min hiding value from LE and assign it to HS
No
Select one task having max hiding value from LE and assign it to off-chip
No
Stop

Fig. 15. Replacer algorithm.

In this case, the task with maximum hiding value is chosen for replacement because replacing other tasks (from the same memory) generates the additional delay. Selection of task with minimum hiding value or maximum hiding value for replacement depends entirely on the memories where the task is initially assigned and to where it gets replaced.

A similar procedure is followed to replace the excess task from LE memory to HS or off-chip memory. It is also given in Fig. 15. When the size of available tasks inside the LE exceeds the size of LE, as well as the size of HS tasks, is lesser than the total size of HS, then a task is selected from LE which is having minimum hiding value (V) and its task is assigned to HS. This procedure is repeated until no more tasks could occupy HS memory (or) the size of tasks available in LE memory equals the size of LE. The excess task present inside the LE memory is allocated to external memory based on the hiding value. Usually, the task with maximum hiding value is chosen for this purpose.

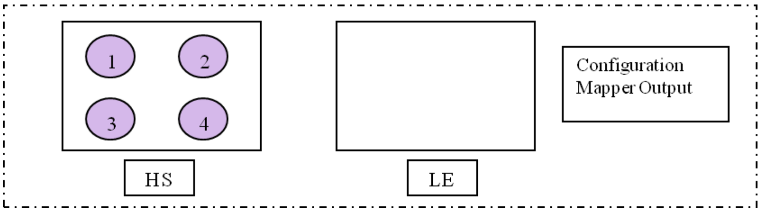
All the details are updated and are stored in the info table. The replacer's mapping is the final updated mapping for any task graph. The final updated details of info table are stored in the off-chip memory. Consider an example shown in Fig. 16. The CM output and the info table details for a specific task graph are given



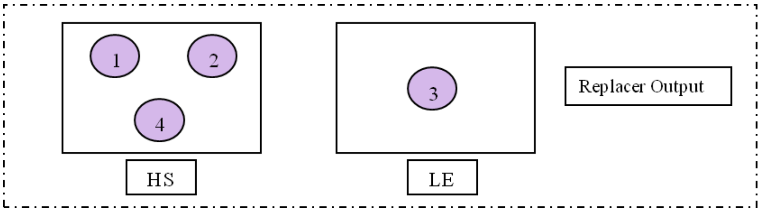
(a)

Task graph info table	
Task	Hiding Value (V_n)
1	Maximum negative value
2	-4
3	-3
4	-5

(b)



(c)



(d)

Fig. 16. Working example of replacer.

Figs. 16(a)–16(c). For this scenario, the main constraint is, HS and LE can occupy only three configurations each. However, based on the hiding value of individual tasks, the CM allocates all the four tasks to HS memory. Therefore, one of the tasks must be selected and should be kept in LE memory by the replacer. Hence, the replacer selects the task which is having the maximum hiding value (from HS memory) and place it inside the LE memory. In the example given in Fig. 16, task 3 is selected and placed in the LE memory. It is shown in Fig. 16(d). If some other tasks are selected for replacement, then it will affect the performance of the task graph execution.

3.2. *Run-time phase*

All the possible configurations to be loaded at run-time are considered to be present in the off-chip memory along with its info table details. The configurations are available in the form of task graphs (With all possible combinations).

When the user interrupts for a specific application and also the microprocessor finds the exact task graph to be scheduled, then the same task graph's mappings are loaded in the info table. Also, as per the info table, the corresponding task graph's individual tasks are placed inside the on-chip memories. Then the scheduler strictly performs the execution of the application based on the information provided by the info table.

For every new application arrival, the existing on-chip memories and the information present in the info table are deleted. This is because when a new task graph arrives in a static system, it is going to be active for a finite number of clock cycles. Since this paper focuses mainly on static systems, the time and energy required for mapping the configurations in on-chip memories can be considered as negligible in comparing it with the entire task graph execution process.

4. Results and Discussions

A simulation environment is created to analyze the proposed methodology. Task graphs (TGs) of multimedia applications benchmark, along with thirty randomly generated TGs are used for the analysis. These randomly generated TGs are carefully selected with different possibilities to validate the methodology accurately. They are grouped into three categories. Each group consists of ten individual TGs. The multimedia benchmarks include JPEG decoder and a sequential version of the MPEG-1 encoder.

Memories available in different versions of FPGA are not uniform. Hence to realize the characteristics of HS and LE memories, a tool named CACTI is used.[46,47] Table 1 provides the required memory features obtained from CACTI tool. These values are taken from the paper.[32] The authors did not use absolute values. Because the actual features of a memory vary considerably depending on the technology used. So, they considered only the relationship in the time and energy consumed per memory access for different representative memories.

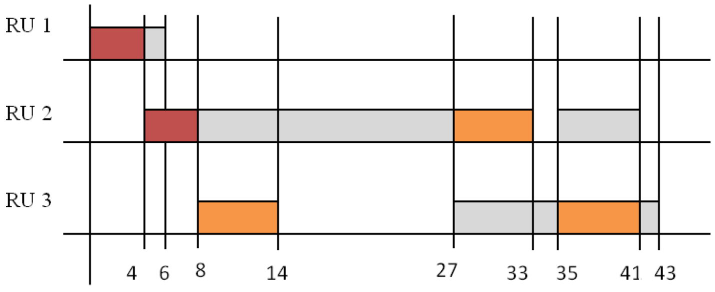
The total execution time of a particular task is the sum of its reconfiguration time and ideal execution time. So, in order to calculate the total execution time of a task, it is necessary to find its reconfiguration time and ideal execution time. For example, consider a task (Configuration) is already loaded inside the FPGA. Moreover, when it is executed (Without reconfiguration), the time taken for its execution is termed as ideal execution time. This ideal execution time of a task can be calculated at the time of design-time phase itself.

Table 1. Memory characteristics.

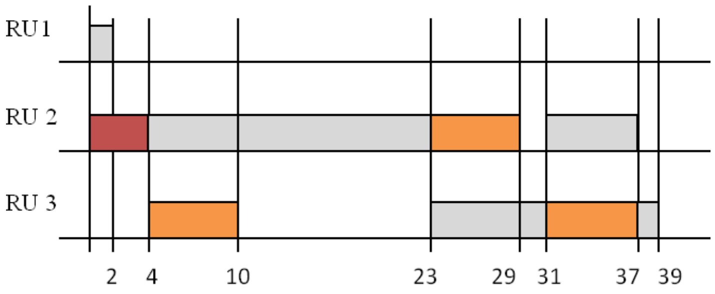
Memory module	The access time for each configuration fetch	Normalized energy consumption
HS	4 ms	1 unit
LE	6 ms	0.7 units
External memory	12 ms	4 units

The reconfiguration time of a task discussed in this proposal is entirely dependent on the memories from where the corresponding task is being fetched. For example, in this proposal, three memories are used in the architecture. Each memory (HS, LE, and off-chip memories) has its own access time (Time to fetch a task from memory and configure it inside the FPGA) and energy consumption values. So, depending upon the memory from where the task is being fetched, the reconfiguration time of a task varies. Also, it is clearly given in Table 1. For instance, consider a task is fetched from HS memory. As per the Table 1, the reconfiguration time spent on loading that task is 4 ms. Similarly, if the same task is considered to be available inside the LE or off-chip memory, then the time spent for reconfiguration of that particular task will be 6 ms or 12 ms respectively.

Using the proposed methodology, the simulation of a MPEG-1 TGs with only 3 RUs is shown in Fig. 17. For this experiment, the configuration size of HS and LE is chosen with 3 each. This simulation output is displayed for the first and the rest of the iterations. In the first iteration, some of the time reconfiguration overheads are eliminated by using memory mapping and task prefetching of configurations. In the second iteration and so on, the application execution time is reduced by 4 ms compared with the first iteration. This is because of the proposed replacement policy used



(a)



(b)

Fig. 17. Simulation of MPEG-1 using our proposed methodology. (a) For first iteration and (b) for the second iteration and so on.

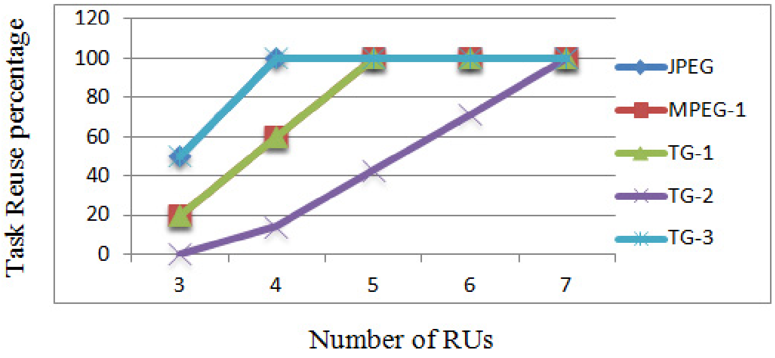


Fig. 18. Simulation output for task reuse percentage.

in addition to the task prefetching and memory mapping of tasks onto the available memories. The subsequent iteration's execution time is just 2 ms away from the ideal execution time.

Figure 18 gives the detail about the task reuse percentage for various numbers of RUs. Using our methodology, when the TGs are executed with more number of RUs, it increases the task reuse percentage. When the task reuse percentage is increased, it automatically reduces the time and energy reconfiguration overheads.

4.1. *Performance obtained using the proposed methodology*

Table 2 provides the simulation output for the proposed methodology obtained for various RUs by keeping the configuration size of HS and LE memories to 3. Since the proposed architecture's on-chip resources are meant to be very less, it is a valid assumption of keeping on-chip memory resources capacity to three configurations each. The percentages obtained in Table 2 are in relation to their ideal execution time. During the first iteration, the proposed methodology uses memory mapping and prefetching techniques. Similarly, in the subsequent iterations along with the memory mapping and prefetching, reuse technique is also used.

To reduce the complexity of Tables 2 and 3, not all the thirty randomly generated TGs are shown. However, representatively one task graph is shown for each group with their average results mentioned. Consider the output result obtained for only 3 RUs. Here, using the proposed methodology, the time reconfiguration overhead (for the first iteration) is reduced by 86% (compared with the performance obtained with full reconfiguration overhead). In the subsequent iterations, the average reduction in time reconfiguration overhead achieved by the proposed methodology is 91%. Except for TG-2, remaining TGs show improved results in their subsequent iteration in comparison with their first iteration. This is because of our proposed replacement policy which reuses the vital RUs. For TG-2, as the average number of tasks is seven and available RUs are only three, the proposed methodology cannot reuse any of the

Table 2. Execution time for the 1st and subsequent iterations of the application for various numbers of RUs.

				Number of RUs									
				3		4		5		6		7	
Application		Ideal execution time (ms)	Execution time with full reconfiguration overhead	1st iteration (ms)	Subsequent iteration (ms)	1st iteration (ms)	Subsequent iteration (ms)	1st iteration (ms)	Subsequent iteration (ms)	1st iteration (ms)	Subsequent iteration (ms)	1st iteration (ms)	Subsequent iteration (ms)
JPEG		79	+53%	+5%	+0%	+5%	+0%	+5%	+0%	+5%	+0%	+5%	+0%
MPEG-1		37	+162%	+16%	+5%	+16%	+0%	+16%	+0%	+16%	+0%	+16%	+0%
Randomly generated task graphs	TG-1	28	+214%	+29%	+14%	+29%	+4%	+29%	+0%	+29%	+0%	+29%	+0%
	TG-2	28	+300%	+68%	+68%	+68%	+54%	+68%	+7%	+68%	+7%	+68%	+0%
	TG-3	24	+200%	+17%	+0%	+17%	+0%	+17%	+0%	+17%	+0%	+17%	+0%
Average			+150%	+21%	+13%	+21%	+8%	+21%	+1%	+21%	+1%	+21%	+0%

Table 3. Energy consumption due to reconfiguration for various numbers of RUs.

				Number of RUs									
				3		4		5		6		7	
Application		Accessing all tasks from off-chip (Units)	Initial mapping (Units)	1st iteration (Units)	Subsequent iteration (Units)	1st iteration (Units)	Subsequent iteration (Units)	1st iteration (Units)	Subsequent iteration (Units)	1st iteration (Units)	Subsequent iteration (Units)	1st Iteration (Units)	Subsequent iteration (Units)
JPEG		16	1	0.19	0.09	0.19	0	0.19	0	0.19	0	0.19	0
MPEG-1		20	1	0.21	0.16	0.21	0.07	0.21	0	0.21	0	0.21	0
Randomly generated task graphs	TG-1	20	1	0.22	0.17	0.22	0.07	0.22	0	0.22	0	0.22	0
	TG-2	28	0.86	0.33	0.33	0.33	0.28	0.33	0.12	0.33	0.06	0.33	0
	TG-3	16	1	0.21	0.11	0.21	0	0.21	0	0.21	0	0.21	0
	Average		0.972	0.23	0.17	0.23	0.08	0.23	0.02	0.23	0.01	0.23	0

existing configurations in their second iteration. Hence, the performance is the same for both the iterations of TG-2.

By taking account of the result obtained for various RUs it can be seen that in the first iteration, almost 86% of the time reconfiguration overheads are eliminated. In the subsequent iterations of the same task graph execution, as the number of RUs gets increased then task reuse percentage also gets increased. Therefore, an average reduction in the time reconfiguration overhead of a further 13% is obtained in comparison with the first iteration. It is because of the proposed replacement policy that provides a scheduling pattern in their first iteration, which enables to reuse some of the vital configurations during their subsequent iterations. Thus, the time reconfiguration overheads are reduced significantly and the performance of the application execution is improved. This can be clearly noticed in Fig. 19 which shows a graph of the average execution time and the number of RUs used. The results are obtained for 1000 iterations and their average value is compared. As it can be seen from the graph that an average of 97% of time reconfiguration overheads is eliminated using the proposed methodology.

4.2. *Energy reduction obtained using the proposed methodology*

Table 3 provides the energy consumption results for various TGs by keeping the individual configuration size of on-chip memories at 3 each. The values from column 3 to column 13 provided in Table 3 are normalized to the energy consumption obtained when all the tasks are fetched from off-chip memory. The energy consumption for mapping the configuration in their respective memories is also shown in column 3 of Table 3. As the number of iterations becomes a larger number, then the energy consumption for mapping the configurations becomes negligible. In the first iteration, 76% of the energy reconfiguration overheads are eliminated using the proposed methodology. In the subsequent iterations, an average reduction of 93.05%

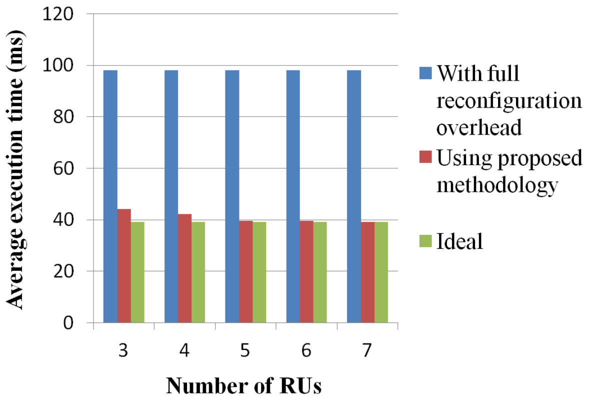


Fig. 19. Performance evaluation of the proposed methodology.

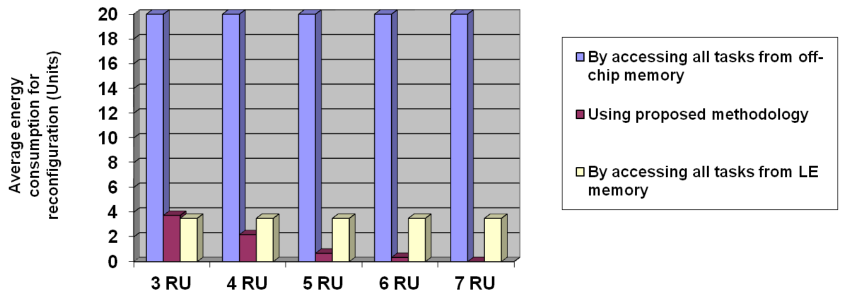


Fig. 20. Reduction in the energy reconfiguration overhead using our proposed methodology.

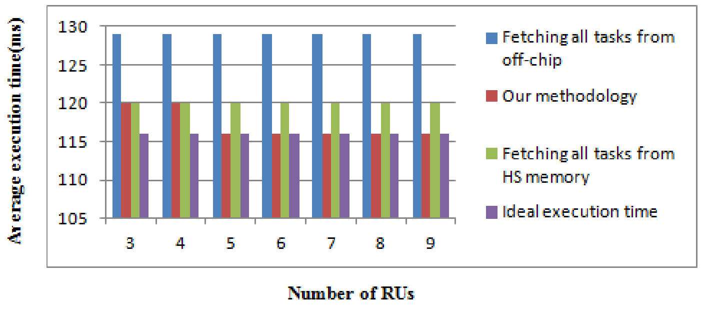
of energy reconfiguration overheads is obtained. On an average, there is an excess reduction of 22.43% of energy reconfiguration overhead in the subsequent iteration in comparison with their first iteration.

A clear representation of reduction in the energy reconfiguration overhead is shown in Fig. 20 in the form of a graph. The applications are executed for 1000 iterations. It is observed that an average of 93.05% of energy reconfiguration overheads is totally eliminated using the proposed methodology. Without proper reusing of tasks, the proposed architecture can reduce the energy reconfiguration overheads by keeping all the tasks in LE memory. This is not possible in the real world as the size of on-chip memories is restricted. By considering that all the tasks are kept in LE memory, an average reduction in the reconfiguration overhead achieved is 82.5%. The proposed methodology reduces the energy reconfiguration overheads further with an average of 12.79% in comparison with the reduction achieved when all tasks are kept in LE memory.

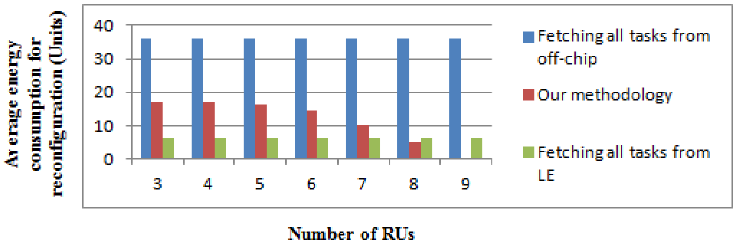
4.3. *Experimental results obtained for larger TGs*

In this experiment, the TGs are grouped and their performance and energy consumption values are analyzed. These experiments are conducted because in a static system there is always a possibility of execution of a group of TGs in a similar fashion. Moreover, this experiment also accounts for the scenario where there is a considerable difference in the number of tasks available to the number of RUs in a system.

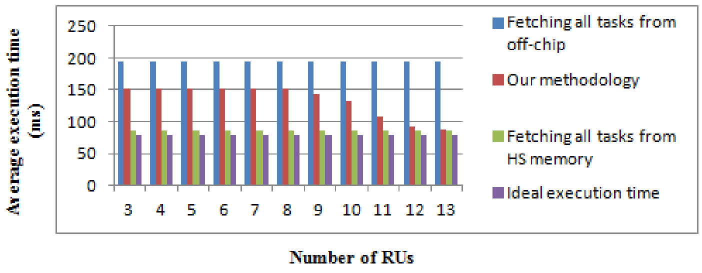
For this purpose, two different groups are considered. Group-1 consists of JPEG and MPEG-1 task graphs and similarly, group-2 consists of TG-1, TG-2, and TG-3 task graphs. Each group of task graph is separately executed with all the possible combinations being considered. These groups of TGs with certain combinations are executed in a similar fashion for 100 iterations and the average results are shown in Fig. 21. In order to execute group-2, a separate procedure is followed. This is because each of the randomly generated task graph groups consists of ten individual task graphs. Out of the ten, one is chosen for every group and the execution of group-2 is



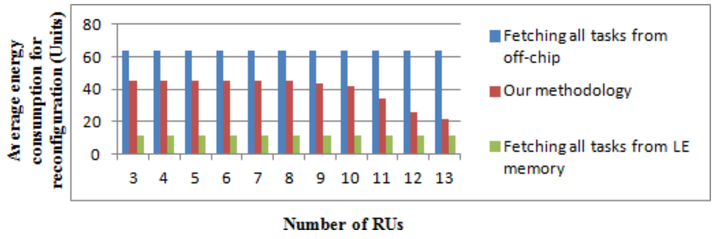
(a)



(b)



(c)



(d)

Fig. 21. (a) Performance evaluation, (b) energy consumption for group-1 TGs, (c) Performance evaluation and (d) energy consumption for group-2 TGs by keeping the on-chip memory capacities to three configurations each for both HS and LE memory.

performed. This is repeated until most of the combinations are executed. Only the average results are shown in Figs. 21(c) and 21(d).

Figure 21(a) gives the execution time of group-1 TGs using our methodology for various RUs. Here, the size of HS and LE memories is kept at 3. The execution time of the TGs when all the Tasks are fetched from off-chip, HS memory, and also their ideal execution times are provided for comparisons.

In group-1, for 3 and 4 RUs, the execution time obtained using our methodology exactly coincides with the execution time obtained when all its tasks are fetched from only HS memory. This is achieved because of the memory mapping combined with prefetch approach. As the number of RUs reaches 5, the execution time obtained using our methodology matches with the ideal execution time of group-1 TGs. It is because of our replacement policy combined with mapping and prefetch approach. Even though we have less on-chip memory space for a larger task graph, our methodology reduces most of the reconfiguration overheads by adequately managing the available resources and configurations.

In Fig. 21(b), energy consumption values of group-1 are plotted for various RUs by keeping the on-chip memories constant at three configurations each (for HS and LE memories). For comparison, energy consumption obtained when all its tasks are fetched from off-chip memory and also from LE memory are used. This is established by considering there is enough space left to store all the tasks inside LE and off-chip memories. Our methodology reduces some of the energy reconfiguration overheads using only memory mapping and prefetch heuristics for RUs 3 and 4, respectively. However, when the number of RUs is increased then the energy reconfiguration overhead is still reduced because of our reuse of tasks combined with memory mapping and prefetch approach.

Similar results are obtained for group-2 TGs having 3 capacities each for both the HS and LE memory and are shown in Figs. 21(c) and 21(d). Execution time obtained using our methodology for various RUs is shown in Fig. 21(c). Up to 8 RUs, the reconfiguration overheads are reduced by memory mapping and prefetch approach. By increasing the RUs number after 8, our methodology efficiently reduces further reconfiguration overheads by using reuse of tasks. The energy consumption values for various RUs of group-2 TGs is shown in Fig. 21(d). The energy reconfiguration overheads are reduced by memory mapping of tasks onto the on-chip memories and prefetch heuristics for RUs 1 to 8. It linearly decreases as the number of RUs gets increased after 8. It is because of our replacement policy. Here, both the group-1 and group-2 TGs are having many tasks and the available on-chip memory capacities are limited (three configurations each for HS and LE memories). By proper management of resources available and the efficient usage of on-chip memories combined with the prefetch approach, reduces most of the reconfiguration overheads.

4.4. *Comparison of the proposed methodology*

This experiment is intended to compare the proposed methodology with the reference works. In this experiment, TGs of multimedia application's benchmarks and randomly generated TGs are used. These TGs are executed individually as well as in combinations. During combinational execution, the pattern in which they get executed is maintained throughout (Since the system under consideration is a static system). Comprehensive combinations of TGs are considered and they are given as the input for different techniques being compared. Each Task graph and its variety of combinations are executed for 100 iterations and their average results are given in Figs. 22 and 23.

Authors of papers[29–31] used task prefetching and reuse techniques together to reduce the reconfiguration overheads. As for reusing the tasks, authors[30] used random replacement policy. Similarly, in papers,[29,31] the authors used their own task reusing techniques like cooperative-2 and Look Forward + Critical (LF + C), respectively. Another interesting approach in the aim of reducing reconfiguration overheads, authors[32] introduced a memory hierarchy which includes two on-chip memories. In which, one is optimized for HS and offers faster memory access. Moreover, the other on-chip memory is optimized for LE and it consumes very less energy for accessing memory. To efficiently utilize these on-chip memories, authors[32] proposed algorithms that use a combination of techniques like optimal memory mapping and task prefetching. All the papers[29–31] mentioned above use any of the two techniques (Either optimal memory mapping and task prefetching or task

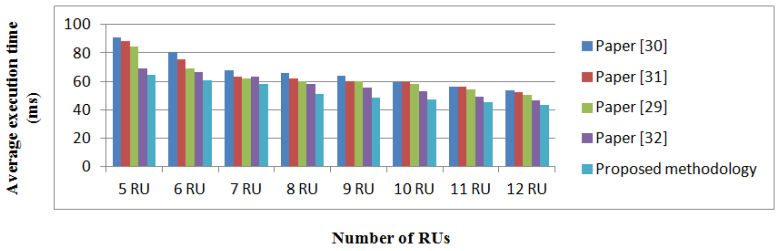


Fig. 22. Performance comparison.

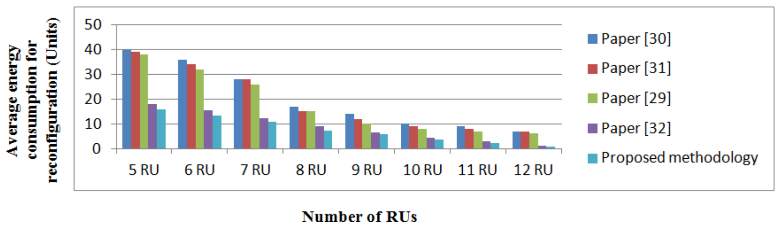


Fig. 23. Comparison of the energy consumption required for reconfiguration.

prefetching and task reusing) in combination for reducing the reconfiguration overheads. However, the proposed methodology used in this paper includes all the three important techniques in combination to reduce the reconfiguration overheads effectively. Because of the combination of three important techniques, the proposed methodology always produces superior results in comparison with the other benchmarking methods.

To make an effective comparison with the proposed methodology, the static configuration mapping algorithm proposed in paper[32] is integrated with an intelligent scheduler that focuses on effective task reuse. Figure 22 shows the performance comparison obtained for various techniques. As can be seen from Fig. 22, the proposed methodology shows the improved result regarding performance compared with the reference techniques. Moreover, it can also be noted; when the number of RUs is lesser (Say, when the number of RUs available is 5 and 6), there are minimal possibilities for task reuse. Hence, the proposed methodology and paper[32] shows improved performance. This is because both the approaches use optimal configuration mapping technique and it is entirely absent in other papers. When the number of RUs used in the architecture is increased (Say, when the number of RUs available reaches beyond 6), then the possibility of task reusing also gets increased. Because of this, papers[29–31] competes with the proposed methodology (Concerning performance) as they include efficient task reusing policies. Even after using some of the efficient task reusing policies in combination with the task prefetching technique, the performance obtained by our proposed methodology is always superior in comparison with other works.[29–31] It is because the proposed methodology uses optimal task mapping along with the task prefetching and efficient task reusing techniques. From Fig. 22, it is also clear that there is a slight increase in the speed of execution is always noticed in the proposed methodology in comparison with the paper.[32] It is due to the proposed novel replacement algorithm which reuses the vital RUs in their subsequent iterations of the same application execution.

Figure 23 shows the energy consumption (For reconfiguration) comparison obtained for the proposed methodology with the reference works. As the obtained results are carefully analyzed from Fig. 23, it can be clearly seen that the proposed methodology's result is far better than other benchmarking methods in terms of energy consumption (For reconfiguration). This is mainly because the proposed methodology uses an optimal configuration mapping technique and this technique is absent in references such as.[29–31] When the number of RUs is lesser (Say from 5 to 7) because of the presence of optimal configuration mapping technique, the proposed methodology's result is very significant when it is compared with references such as.[29–31] As the number of RUs is increased beyond 7 the proposed methodology still looks better in reducing the energy consumption (For reconfiguration), but the results are not very significant as in the case when the number of RUs is lesser. It is because; Refs. 29–31 efficiently utilize the available RUs and effectively reuse the vital RUs to reduce energy reconfiguration overheads. Even after reusing most of the tasks (When the available RUs is larger), they still consume a little higher amount of energy when compared with the proposed methodology. This is mainly because, Refs. 29–31 uses only task prefetching and efficient task reusing techniques. Also, the optimal memory mapping is absent and which makes the proposed methodology superior.

It can be seen from Fig. 23, that the proposed methodology outperforms the paper[32] in reducing the energy consumption (For reconfiguration). As the proposed methodology includes optimal task reusing technique, there is always a significant improvement (in comparison with the paper[32]) in the reduction of energy consumption required for reconfiguration. Moreover, this is true for all numbers of RUs.

5. Conclusion

Reconfiguration overheads in FPGA are generated because of fetching the configurations from off-chip memory and also due to the improper management of configurations with the available FPGA resources. To reduce these reconfiguration overheads, the proposed methodology uses configuration mapping on the available on-chip memories, prefetch heuristics, and configuration reuse technique. All these techniques are implemented for a static system. Most of the reconfiguration overheads are eliminated in the first iteration of the task graph execution using only the memory mapping and configuration prefetch technique. In the subsequent iterations of the same task graph execution, some of the remaining reconfiguration overheads are significantly reduced by using the proposed replacement policy. As the proposed replacement policy reuses some of the vital tasks, the application approaches towards its ideal execution time in their subsequent iterations of the same task graph execution. Similarly, the energy reconfiguration overhead approaches to zero in their subsequent iterations of the same task graph execution. Therefore, for a static system, our approach eliminates most of the reconfiguration overheads by adequately managing the available resources and configurations.

References

1. X. X. Xia and T. T. Tay, Intra-application energy reduction for microprocessor low-power design, *J. Circuits, Syst. Comput.* **18** (2009) 181–198.
2. A. Marshall *et al.*, A reconfigurable arithmetic array for multimedia applications, *Proc. the 1999 ACM/SIGDA Seventh Int. Symp. on Field Programmable Gate Arrays.* ACM, 1999.
3. R. Tessier and W. Burleson, Reconfigurable computing for digital signal processing: A survey, *J. VLSI Signal Process.* **28** (2001) 7–27.
4. L. Jozwiak and N. Nedjah, Modern architectures for embedded reconfigurable systems — a survey, *J. Circuits Syst. Comput.* **18** (2009) 209–254.
5. E. J. McDonald, Runtime FPGA partial reconfiguration, *Aerospace Conf. 2008 IEEE*, IEEE, 2008.

6. M. Liu *et al.*, Run-time partial reconfiguration speed investigation and architectural design space exploration, *Int. Conf. on Field Programmable Logic and Applications, 2009. FPL 2009*, IEEE, 2009.
7. S. Liu, R. N. Pittman and A. Forin, Energy reduction with run-time partial reconfiguration, *FPGA* (2010).
8. B. Ouni and A. Mtibaa, Online scheduling and placement of hardware modules on partially dynamic architectures, *J. Circuits Syst. Comput.* **22** (2013) 1350005.
9. I. Belaid *et al.*, Complete and approximate methods for off-line placement of hardware tasks on reconfigurable devices, *J. Circuits, Syst. Comput.* **22** (2013) 1250080.
10. Virtex-5 FPGA Configuration User Guide, Ug191(v3.10), Xilinx Inc., San Jose, CA, USA (2011).
11. S. Liu, R. N. Pittman, A. Form and J.-L. Gaudiot, On energy efficiency of reconfigurable systems with run-time partial reconfiguration, *Proc. IEEE Int. Conf. ASAP*, Jul. 2010, pp. 265–272.
12. S. Keckler, W. Dally, B. Khailany, M. Garland and D. Glasco, GPUs and the future of parallel computing, *IEEE Micro* **31** (2011) 7–17.
13. E. P. Ramo, J. Resano, D. Mozos and F. Catthoor, Reducing the reconfiguration overhead: A survey of techniques, *Proc. Int. Conf. ERSA*, 2007, pp. 191–194.
14. T. Becker, W. Luk and P. Y. K. Cheung, Energy-aware optimization for run-time reconfiguration, *Proc. IEEE Annu. Int. Symp. FCCM*, May 2010, pp. 55–62.
15. I. Kuon and J. Rose, Measuring the gap between FPGAs and ASICs, *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **26** (2007) 203–215.
16. D. I. Lehn, K. Puttegowda, J. H. Park, P. Athanas and M. Jones, Evaluation of rapid context switching on a CSRC device, *Proc. Int. Conf. ERSA*, 2002, pp. 209–215.
17. B. Yitzhak and E. Fiksman, Dynamic reconfiguration architectures for multi-context FPGAs, *Comput. Electr. Eng.* **35** (2009) 878–903.
18. H. Walder and M. Platzner, Online scheduling for block-partitioned reconfigurable devices, *Proc. Des., Autom. Test Eur. (DATE)*, Munich, Germany, 2003, pp. 290–295.
19. Y. Qu, J.-P. Soininen and J. Nurmi, A parallel configuration model for reducing the run-time reconfiguration overhead, *Proc. Des., Autom. Test Eur. (DATE)*, Munich, Germany, 2006, pp. 1–6.
20. K. N. Vikram and V. Vasudevan, Mapping data-parallel tasks onto partially reconfigurable hybrid processor architectures, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **14** (2006) 1010–1023.
21. S. Banerjee, E. Bozorgzadeh and N. Dutt, Exploiting application data-parallelism on dynamically reconfigurable architectures: Placement and architectural considerations, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **17** (2009) 234–247.
22. S. Radu and S. D. Cotofana, Bitstream compression techniques for Virtex 4 FPGAs, *2008 Int. Conf. Field Programmable Logic and Applications*, IEEE, 2008
23. Z. Li and S. Hauck, Configuration compression for virtex FPGAs, *Proc. 9th Annu. IEEE Symp. FCCM*, April 2001, pp. 147–159.
24. A. Dandalis and V. K. Prasanna, Configuration compression for FPGA based embedded systems, *Proc. ACM/SIGDA Int. Symp. FPGA*, December 2001, pp. 173–182.
25. J. Resano, D. Mozos and F. Catthoor, A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable HW, *DATE'05*, 2005, pp. 106–111,
26. Z. Li and S. Hauck, Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation, *FPGA'02*, 2002, pp. 187–195.

27. Z. Li *et al.*, Configuration cache management techniques for FPGAs, *IEEE FCCM'00*, 2000, pp. 22–36.
28. G. Enemali, A. Adetomi and T. Arslan, FAReP: Fragmentation-aware replacement policy for task reuse on reconfigurable FPGAs, *2017 IEEE Int. Parallel and Distributed Processing Symp. Workshops (IPDPSW)*, IEEE, 2017.
29. J. Clemente, J. Resano, C. Gonzalez and D. Mozos, A hardware implementation of a run-time scheduler for reconfigurable systems *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* 19 (2011) 1263–1276.
30. J. Resano, D. Verkest, D. Mozos, S. Vernalde and F. Catthoor, Run-time scheduling for multimedia applications on dynamically reconfigurable systems, *ESTImedia* (2003), pp. 156–162.
31. J. Resano, D. Mozos, D. Verkest and F. Catthoor, A reconfiguration manager for dynamically reconfigurable hardware, *IEEE Des. Test Comput.* **22** (2005) 452–460.
32. J. A. Clemente, E. P. Ramo, J. Resano, D. Mozos and F. Catthoor, Configuration mapping algorithms to reduce energy and time reconfiguration overheads in reconfigurable systems, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **22** (2014) 1248–1261.
33. J. A. Clemente *et al.*, Hardware architectural support for caching partitioned reconfigurations in reconfigurable systems, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **24** (2016) 530–543.
34. I. Hariharan and M. Kannan, Reducing reconfiguration overheads of a reconfigurable dynamic system using active run-time prediction, *J. Electr. Eng.* **18** (2018) 349–356.
35. Y. Kim, Power-efficient configuration cache structure for coarse-grained reconfigurable architecture, *J. Circuits Syst. Comput.* **22** (2013) 1350001.
36. K. Cao, J. Zhou, T. Wei, M. Chen, S. Hu and K. Li, A survey of optimization techniques for thermal-aware 3D processors, *J. Syst. Archit.* (2019), doi: https://doi.org/10.1016/j.sysarc.2019.01.003.
37. 7 Series FPGAs Overview, DS180 (v1. 11), Xilinx, San Jose, CA, USA (2012).
38. ZC702 Evaluation Board for the Zynq – 7000 XC7Z020 Extensible Processing Platform, User Guide, UG850 (v1. 0), Xilinx, San Jose, CA, USA, (2012).
39. Altera (2011), Stratix V Device Datasheet, San Jose, CA, USA [Online], Available: http://www.altera.com/literature/hb/stratix-v/stx5_53001.pdf.
40. Altera (2011), Quartus II Handbook Version 13.0, Volume 1: Design and Synthesis, San Jose, CA, USA [Online], Available: http://www.altera.com/literature/hb/qts/quartusii_handbook.pdf.
41. AXI Reference guide, UG761 (v13. 1), Xilinx, San Jose, CA, USA (2011).
42. Local Memory Bus (LMB) v1. 0 (v1. 00a), Xilinx, San Jose, CA, USA (2005).
43. R. Fromm *et al.*, The energy efficiency of IRAM architectures, *Proc. 24th Annu. Int. Symp. Comput. Archit. (ISCA)* (1997), pp. 327–337.
44. Virtex-5 Family Overview, DS100(V5.0), Xilinx, San Jose, CA, USA, (2009).
45. C. Wong, P. Marchal and P. Yang, Task concurrency management methodology to schedule the MPEG4 IM1 player on a highly parallel processor platform, *Proc. Ninth Int. Symp. Hardware/Software Codesign, CODES* 2001.
46. http://www.hpl.hp.com/research/cacti/.
47. http://www.ece.ubc.ca/∼ stevew/cacti/run_frame.html.

Journal of Circuits, Systems, and Computers
Vol. 27, No. 3 (2018) 1830001 (21 pages)
© World Scientific Publishing Company
DOI: 10.1142/S0218126618300015

A Survey on FFT/IFFT Processors for Next Generation Telecommunication Systems*

Elango Konguvel† and Muniandi Kannan‡

Department of Electronics Engineering,
Madras Institute of Technology Campus,
Anna University, Chennai, Tamilnadu, India
†*konguart08@gmail.com*
‡*mkannan@annauniv.edu*

Received 21 September 2016
Accepted 29 May 2017
Published 7 July 2017

The Fast Fourier Transform and Inverse Fast Fourier Transform (FFT/IFFT) are the most significant digital signal processing (DSP) techniques used in Orthogonal Frequency Division Multiplexing (OFDM)-based applications which include day-to-day wired/wireless communications, broadband access, and information sharing. The advancements in telecommunication technologies require an efficient FFT/IFFT processing device to meet the necessary specifications which depend on the particular application. A real-time implementation of high-speed FFT/IFFT processor with less area that operates in minimal power consumption is essential in designing an OFDM integrated chip. A comparative study of efficient algorithms and architectures for FFT chip design is presented in this paper. It is also recommended that mixed-radix/higher-radix algorithm combined with Single-path Delay Commutator (SDC) architecture is appropriate for massive MIMO in 5G, optical OFDM, cooperative MIMO and multi-user MIMO-based applications.

Keywords: FFT; IFFT; pipelined FFT; sequential FFT; OFDM; 5G.

1. Introduction

Orthogonal Frequency Division Multiplexing (OFDM) is quite a new spectrally efficient digital modulation scheme which employs multiple carriers that are mutually orthogonal to one another over a given time interval. Multiple Input Multiple Output-Orthogonal Frequency Division Multiplexing (MIMO-OFDM) has extensive applications in the field of wireless communications such as IEEE 802.11,[1] IEEE 802.15, IEEE 802.16,[2] IEEE 802.20, WiMax and 3GPP Long Term Evolution (LTE).[3]

*This paper was recommended by Regional Editor Emre Salman.
† Corresponding author.

Table 1. Various FFT sizes and their applications.

Applications	Standard	Supporting FFT/IFFT size
Wireless networks	IEEE 802.11a/g	64
	IEEE 802.11ac	64/128/256/512
	IEEE 802.11n	64/128
	IEEE 802.16e	128/256/512/1024/2048
	IEEE 802.15	512
Wired networks	ADSL	512
	VDSL	256/512/1024/2048/4096/8192
Terrestrial broadcasting	DAB	256/512/1024/2048
	DVB-T	2048/8192
	DVB-H	2048/4096/8192
	DMB-T/H	4096
	DVB-C2	4096
	DVB-T2	1024/2048/4096/8192

The crucial blocks in a MIMO-OFDM transceiver are Inverse Fast Fourier Transform (IFFT) and Fast Fourier Transform (FFT) units, which implement efficiently the highly complex computation for Inverse Discrete Fourier Transform (IDFT) and Discrete Fourier Transform (DFT) operations, respectively.[4,5] Major digital signal processing (DSP) applications depend on the size (N) of the FFT/IFFT (64–8192 points) computation. Table 1 provides various FFT sizes with their corresponding applications.[6–8]

The recent advancements in telecommunication technologies such as massive MIMO in 5G, optical OFDM, cooperative MIMO and multi-user MIMO require an efficient FFT/IFFT processing module to meet the necessary specifications of that particular application. A real-time implementation of high-speed FFT/IFFT module with less area that operates in minimal power consumption is very essential in designing an MIMO-OFDM integrated chip. The objective of this analysis is to select an area, power and delay efficient FFT/IFFT processor for MIMO-OFDM-based applications. A comparative study of efficient algorithms and architectures for FFT chip design is presented in this paper. The efficiency of FFT/IFFT processors can be evaluated using the measures like area utilization, power consumption and delay. Therefore, the technology process of integrated chip, type of FFT/IFFT architecture, the size N of FFT/IFFT, area utilization and power consumption are the various parameters considered for the analysis and discussion.

The organization of the paper is as follows. A brief introduction to FFT algorithms is given in Sec. 2. Memory references, as well as sequential FFT algorithms, are discussed in Sec. 3. Section 4 lists the different pipelined FFT architectures. Comparative analysis and discussions of various FFT/IFFT processors are given in Sec. 5 and concluding remarks are stated in Sec. 6.

2. Preliminaries

The DFT and IDFT of N-point data sequence can be given as

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi kn/N}, \quad 0 \leq k \leq N-1, \tag{1}$$

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{j2\pi kn/N}, \quad 0 \leq n \leq N-1. \tag{2}$$

The DFT and IDFT of any given sequence of length N are $x(n)$ and $X(k)$, respectively. From Eqs. (1) and (2), it is clear that n and k are nth and kth samples of N data points, where N can be 64/128/256/512/1024/2048/4096/8192 points which depends on the specific application. The exponential term given in Eqs. (1) and (2) represents the twiddle factor needed for FFT/IFFT computation. The direct implementation of DFT requires N^2 complex multiplications and $N(N-1)$ complex additions, where N is the number of data points. Divide and conquer technique, FFT, proposed by Cooley and Tukey, considering the periodicity and symmetric properties of DFT algorithm significantly reduced the number of complex multiplications to $(N/2)\log_2 N$ and a number of complex additions to $N\log_2 N$.[9] Table 2 provides the comparison of computational complexity for direct computation algorithm versus FFT algorithm.

By using divide and conquer approach, the number of data points N is partitioned into $r_1, r_2, r_3, \ldots$ and so on, where r is called the radix of FFT. The most common and familiar FFTs are with $r = 2$, FFT algorithm is referred to as a radix-2 FFT algorithm. However, further radices of range 2–10 are also used based on its application. In single-radix FFTs, the size of data points N must be the power of radix value. For example, with radix-4, the number of data points N in the FFT should be a power of 4.

Table 2. Computational complexity of direct computation versus FFT algorithm.

	Direct computation		FFT algorithm	
Number of data points N	Complex multiplications N^2	Complex additions $N(N-1)$	Complex multiplications $(N/2)\log_2 N$	Complex additions $N\log_2 N$
16	256	240	32	64
64	4096	4032	192	384
128	16384	16256	448	896
256	65536	65280	1024	2048
512	262144	261632	2304	4608
1024	1048576	1047552	5120	10240
2048	4194304	4192256	11264	22528
4096	16777216	16773120	24576	49152
8192	67108864	67100672	53248	106496

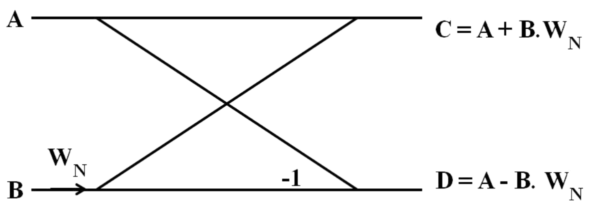


Fig. 1. Radix 2 DIT butterfly.

But, mixed-radix FFT can be done on composite sizes for decomposing a non-prime into prime factors. For example, an FFT of size 500 can be decomposed in five stages using radices of 2 and 5 since $500 = 2 \times 2 \times 5 \times 5 \times 5$ or 3 stages using radices of 5 and 10 since $500 = 5 \times 10 \times 10$. This basic r-point computation is called as butterfly unit.

The decomposition can be categorized as Decimation In Frequency (DIF) and Decimation In Time (DIT), depending upon the partition that takes place from input and output data points respectively. Figures 1 and 2 show the basic radix-2 butterfly diagrams for DIT-FFT and DIF-FFT computations, respectively. In Figs. 1 and 2, A and B denote the complex input from previous stage whereas C and D denote the complex output of the current stage (or complex input to the next stage). The twiddle factors W_N are defined as the co-efficients which are used to compute results from the previous stage and to form inputs to the next stages of FFT algorithm. Mathematically,

$$W_N = e^{-j2\pi/N}. \tag{3}$$

The implementation of FFT/IFFT processors widely uses in-place-memory updating and pipeline architectures. An in-place memory updating architecture, a single radix-r butterfly will be sufficiently re-used for N-point FFT/IFFT computation. Radix-r butterfly will be idle in the case of writing and reading input and output values. If the re-composition has different radix points, it is called mixed-radix FFT algorithm. When the radix is higher, the number of complex multiplications and additions are lesser, which in turn reduces the chip area and cost. Continuous flow mixed radix-based FFT architecture performs the computation with two N-sample memories for the continuous data stream.

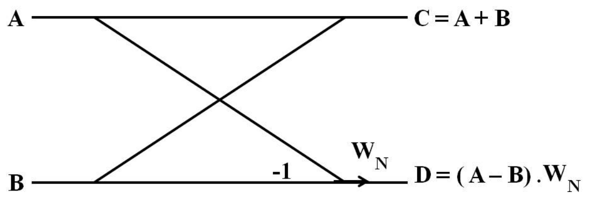


Fig. 2. Radix 2 DIF butterfly.

3. Sequential FFT Architectures

The direct implementation of the conventional Cooley–Tukey algorithm is sequential since the same radix butterfly unit is used in every stage. This radix-2 architecture requires $(N/2)\log_2 N$ complex multipliers and $N\log_2 N$ complex adders for N input data points as well as the twiddle factors for the computations are stored and accessed from a look-up table. Since FFT/IFFT operations are complex, the real and imaginary operations should be placed apart from each other. The hardware requirements for this architecture should be made up of adders, multipliers and individual memory for input/output bit reversals. Even though hardware requirements are low, latency is higher for the stream of input data points.[10,11] The sequential architecture for $N = 16$ is shown in Fig. 3. Since radix-2 butterfly computations are used, FFT for 16 data points are decomposed using four stages from stage 1 to stage 4. The inputs $x(0)$–$x(15)$ are fed into the algorithm in bit reversal (since bit-reversed indices are used to combine various stages in FFT) order whereas outputs $X(0)$–$X(15)$ are taken in natural order.

3.1. *Modified sequential architecture*

In this method, the identical twiddle factors required for the computations are grouped, so that the grouped twiddle factors are fed only once for the entire FFT computation. This reduces the latency as well as the memory occupied by the twiddle

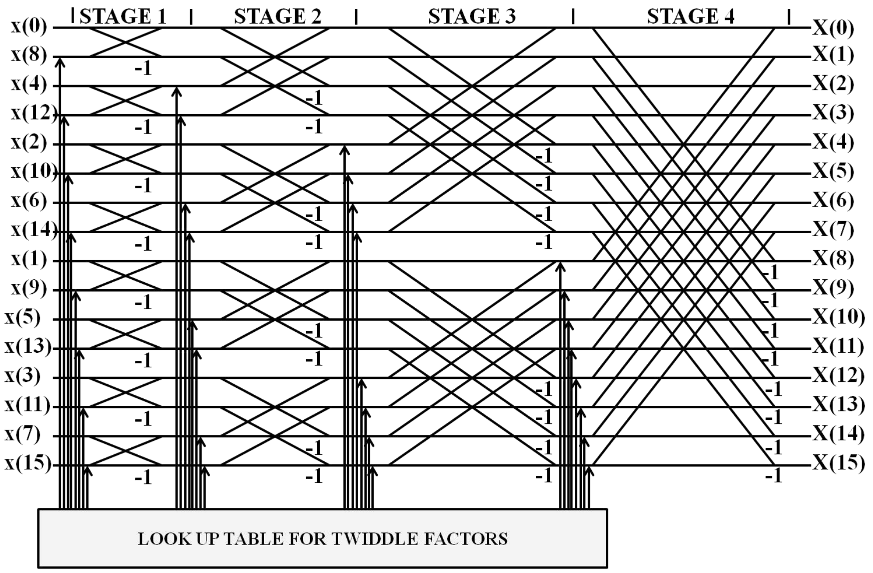


Fig. 3. Conventional DIT-FFT for $N = 16$.

factors. The identities accomplish the grouping of twiddle factors,

$$\mathrm{W}_N^m = \mathrm{W}_N^{N/4}.\mathrm{W}_N^{m-N/4} = -j\mathrm{W}_N^{m-N/4}\,, \quad N/4 \le m \le N/2 \tag{4}$$

$$= \mathrm{W}_N^{N/2}.\mathrm{W}_N^{m-N/2} = -\mathrm{W}_N^{m-N/2}\,, \quad N/2 \le m \le 3N/4 \tag{5}$$

$$= \mathrm{W}_N^{3N/4}.\mathrm{W}_N^{m-3N/4} = j\mathrm{W}_N^{m-3N/4}\,, \quad 3N/4 \le m \le N \tag{6}$$

$$= \mathrm{W}_N^m, \quad 0 \le m \le N/4\,. \tag{7}$$

The twiddle factors are fed into the algorithm in such a way that the other computations are unaffected as well as the redundant memory modules are eliminated which is shown in Fig. 4.[12] Stage 1 is further decomposed into four substages to compute the results faster since identical twiddle factor computations are grouped. A 16-bit 64-point modified sequential algorithm along with cache memory based 1-dimensional (1D) FFT architecture[13] results in an area efficient, high-speed processor optimal for Wireless Local Area Network (WLAN)-based applications. Floating point butterfly unit using Fused-Dot-Product-Add (FDPA) based on Binary-Signed-Digit (BSD) representation is proposed to maximize the speed of the FFT computation.[14,15] Energy efficient computation units are also achieved through a stage skipping and merging to implement FFT/IFFT in Single Instruction Multiple Data (SIMD) and non-SIMD architectures.[16] The radix-2 butterfly is optimized to reduce one addition and one subtraction operation; optimized radix-2 butterfly is used in the radix-4 butterfly to reduce the number of real multipliers. Since radix-4 butterfly is also optimized, power utilization is reduced when compared with the conventional radix-4 butterflies.[17]

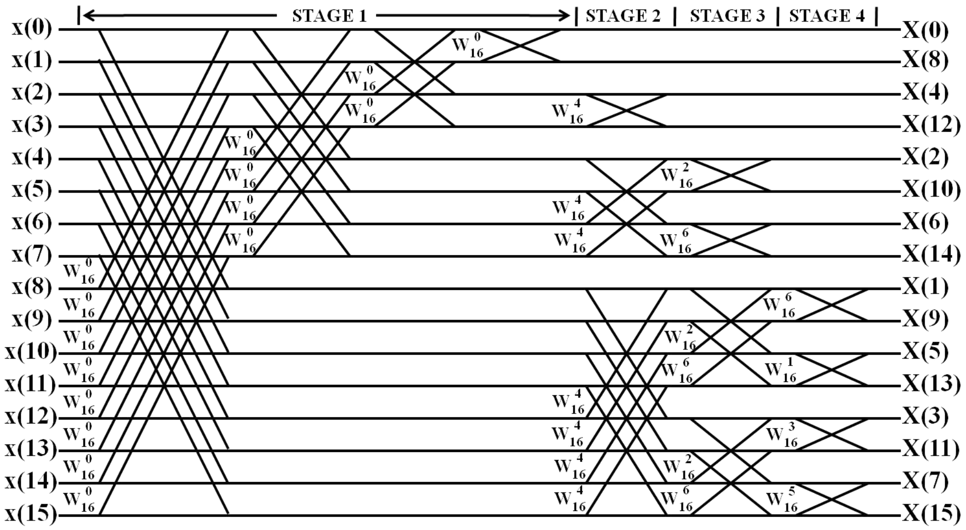


Fig. 4. Modified DIF FFT for $N = 16$.

3.2. *Real-valued FFT (RFFT) architecture*

The real-time bio-medical, sensor and radar-based communication processing generates redundant values in the output since the real-valued data signals exhibit conjugate symmetry. Better hardware utilization can be achieved by eliminating the redundancy using several techniques such as radix-2^5,[18] interleaved real and hybrid datapaths,[19] modified radix-2,[20] register based storage design[21] and radix-2^2 canonic based Real-valued FFT (RFFT).[22]

3.3. *Mixed radix FFT architecture*

Mixed Generalized High-Radix (GHR) FFT algorithm[23,24] uses 2D and 1D FFT factorization methods to reduce the hardware usage much less than the conventional FFT/IFFT methods. Processing speed and hardware efficiency are improved on the GHR method since it endures eight radices and 34 different FFT lengths for all LTE applications. Multiplier-less split-radix-based FFT architecture using Distributed Arithmetic (DA) for complex multiplications has been proposed.[25] Even though the number of overall arithmetic operations is reduced in this architecture, the complexity and number of operations in a butterfly are increased. Distributed Floating Point Arithmetic (DPFA) based variable length (32–2048) FFT architecture eliminates the power consuming complex twiddle factor multipliers by incorporating folded DA butterflies which reduce area usage as well as power consumption.[26] The FFT core is divided into three parts, namely dependant, in-dependant and reordering to minimize processing time in Multicore DSP implementation and is achieved with a reduction factor of 1/3.[27] A 60% resource utilization and 14% performance improvement are achieved by using the complex math processor in a low-cost radix-4/radix-2^2 butterfly-based FFT processor.[28]

A novel radix-2 split-radix FFT is proposed in which memory-sharing and clock gating are considered to avoid the inevitable switching action of multipliers which makes the circuit to operate faster. Look-up tables are used to generate the addresses for twiddle factors which eliminate the time taken for address generation.[29] An eight parallel data path for 512-point FFT computation with modified radix-25 butterfly is proposed to reduce the number of complex multipliers for twiddle factor multiplication for OFDM Wireless Personal Area Networks (WPAN) applications with a high throughput of 2.5 GS/s at 310 MHz.[30] Radix-2/4 butterfly based 64 to 128 point mixed FFT processor with Coarse Grain Reconfigurable Array (CGRA) embodied Reduced Instruction Set Computing (RISC) processor is proposed to eliminate partial execution timing constraints for IEEE 802.11 based FFT applications.[31] A parallel pipelined Real Fast Fourier Transform (RFFT) architecture in which four inputs are processed using modified processing elements that have two radix-2 butterfly units that remove redundant operations in the flow graph with conflict-free address generation scheme is proposed for real-valued signals.[32]

A scalable radix-2 based N-point FFT processor in which two radix-2 butterfly units are used for computation of input data at single clock pulse is proposed for multi-carrier systems like OFDM transceivers.[33] A Transport Triggered Architecture (TTA) and programmable Instruction Level Parallelism (ILP) based mixed radix-4/2/3 FFT processor that supports all the FFT sizes for LTE (128~2048/1536) applications is exploited in Refs. 34 and 35. A mixed-radix comprises of radix-2, radix-2^2 and radix-2/4/8 that computes at 512–8192 points FFT operation, optimized by sub-structure sharing mechanism, results in an area efficient, low-complex processor for applications such as Digital Audio Broadcasting (DAB), Digital Video Broadcasting-Terrestrial (DVB-T) and Digital Video Broadcasting-Handheld (DVB-H).[36] By using hardware sharing schemes in radix-2^2 algorithms, hybrid architecture is developed that retains the systematic and variable characteristics of recursive DFT algorithms.[37] A 256 point mixed-radix-$2^3/2^5$ algorithm is proposed to reduce the complex multipliers by adopting 32 parallel data paths and eight sequential groups. The performance of this mixed-radix architecture is suitable for optical OFDM systems.[38] A CORDIC module is used to reduce the complexity of trigonometric computations (for twiddle factor multiplication) in the butterfly operations involved in mixed-radix FFT architecture.[39]

4. Pipelined FFT Architectures

A separate arithmetic unit can be used in each stage of the FFT unlike in the sequential architectures. This type of parallelism among each stage improves the processor performance in a significant way. A factor of $\log_2 N$ enhances the throughput of pipelined FFT architecture. Since each stage of this FFT algorithm has its basic computational units, the pipelined FFT can be called as cascaded FFT algorithm. Each stage does its operation as soon as the input data to be processed are available. These types of pipelined FFT processors are more suitable for real-time MIMO-OFDM as well as in-place signal processing applications. In pipeline architectures, Single-path Delay Feedback (SDF) and Multipath Delay Commutator (MDC) are the two popular architectures. SDF along with input scheduling enables multiple input streams processing with a single FFT/IFFT processor. MDC uses more switch boxes to resolve feedback path into feedforward paths. The MDC-based FFT architecture saves more area than SDF architecture-based FFT processor with multiple streams. Single radix-N butterflies are used at each folding stage of the MDC architecture-based FFT processor with N data streams. The pipelined FFT architectures are categorized into the following groups.

4.1. *Radix 2 multipath delay commutator (R2MDC)*

The most straightforward approach to implement radix-2 butterfly operation is the radix-2 Multipath delay commutator (R2MDC). The input data sequence is split

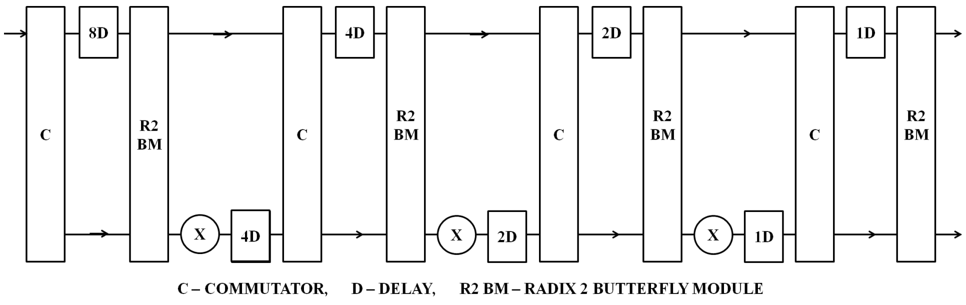


Fig. 5. Radix-2 multipath delay commutator for $N = 16$.

into two parallel data streams (for radix-2 butterfly) using two path commutator (C) and is fed into the data processing elements which are radix-2 Butterfly Modules (R2BM) at an equal distance by incorporating appropriate delay units (8D for 8 units of delay, 4D, 2D and 1D). The processing elements as well as the multiplier unit are used only half of the time and hence utilization rate of this pipelined FFT is 50%. A generalized R2MDC FFT architecture is shown in Fig. 5.

Mixed/multi-radix butterfly stages are also used with MDC based FFT/IFFT processors to reduce the memory usage or to share the memory between the butterfly operations. A modified radix-4 butterfly algorithm has been proposed that can perform a single radix-4 operation or two radix-2 operations. Eight parallel data paths are used before the modified radix-4 butterfly for higher throughput rate and low hardware complexity.[38] A simple counter is used to generate the mixed radix butterfly sequences and fixed/mixed radix memory addressing schemes are introduced in MDC-FFT architectures. In fixed radix memory addressing scheme, two memory words for real and imaginary elements, two counters for address and column counter are used. The number of counters can be reduced in fixed radix memory addressing scheme without sacrificing hardware utilization for Mixed Radix (MR) addressing operation.[40] Flexible Radix Configuration (FRC) based MDC-FFT architecture is proposed[42] to improve the throughput and power efficiency. A radix-2 DIF butterfly and pipeline data scheduling based 1024 point fixed (32-bit) FFT structure with ping-pong memory access scheme is proposed for high speed and real-time signal processing applications.[43]

4.2. *Radix 4 multipath delay commutator* (*R4MDC*)

The radix-4 MDC architecture is similar to radix-2 MDC in which Radix-4 Butterfly Module (R4BM) replaces radix-2 butterfly unit, and its corresponding 4-path commutator (C) is used. Since the input data sequence is separated into four parallel data streams, only 1/4 of input data is fed into the multipliers and computing elements; the utilization rate is reduced to 25%. This architecture is not

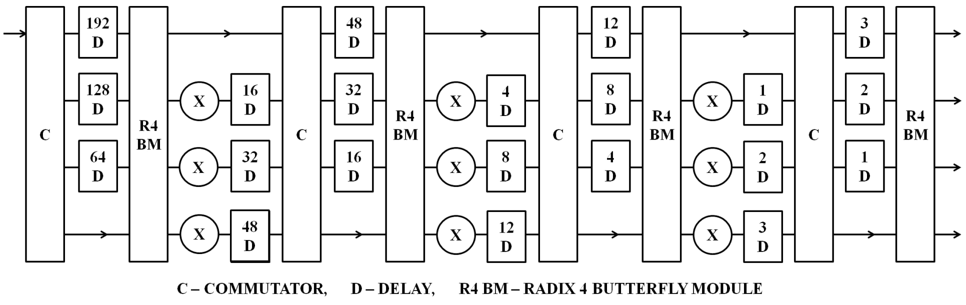


Fig. 6. Radix-4 multipath delay commutator for $N = 256$.

recommended when hardware requirement and utilization rate are taken into consideration. Radix-4 MDC architecture for 256 input data points is shown in Fig. 6.

Since larger data point is considered for computation delay, units are also larger that ranges upto 192 units of delay (192D for 192 units of delay). An input memory scheduling scheme is introduced with MDC to increase the utilization rate and obtain the variable length (128/512/1024/2048) MDC-FFT processor. Here, Radix N butterfly is used for N input data streams. One group of the memory bank is used for storing data from one input stream. $3N/4$ samples from one memory group are processed at a time in radix-4 butterfly unit. At the same instance, the elements of the other memory group are updated from the other input data streams. A configurable radix-8/radix-4 butterfly is used in the last stage with a constant multiplier for twiddle factor multiplications. A control/select input line is used for selecting the intermediate stages to the number of data points/streams.[44–46] Radix-2^k MDC architecture, for both DIF and DIT decompositions, can be used for any number of parallel input data streams. This architecture achieves high throughput with fewer hardware resources needed for recent applications.[46] The number of non-trivial multipliers is reduced by using a mixed-radix butterfly operation along with multipath delay feedback architecture for 1–4 channel 1024/2048/4096 FFT computation with less hardware complexity.[47] To lessen the complexity in twiddle factor multiplication, a radix-2^4–2^2–2^3 based MDF-FFT architecture is proposed which is optimal for IEEE 802.11ad (wireless personal area) applications[48] and higher radix butterfly units.[76] The conventional Multipath Delay Feedback (MDF) architecture inefficiently utilizes the adders and multipliers. A mixed-decimation MDF (M^{2}DF) architecture based on radix-2^k algorithm[49] and radix-4 algorithm[50] are proposed for better hardware utilization. A variable length radix-2/3/2^2/2^3 algorithm-based MDC architecture is proposed to reduce the number of operating cycles for FFT computation.[51] Many applications require simultaneous calculation of independent FFT samples; MDC architecture is configured for two data streams for achieving 100% hardware utilization.[52]

4.3. *Radix 2 single-path delay feedback* (*R2SDF*)

As the number of delay elements in MDC architecture is a major disadvantage, a feedback mechanism is provided in such a way that one-half of output data from each computing stage is fed back into the input of the same computing stage, as the data is directly given to the butterfly unit for computation. This technique was introduced in Ref. 53, and 100% memory utilization is achieved in Ref. 54. When the butterfly computation is radix-2, it is called Radix-2 Single-path delay feedback (R2SDF). Although the number of butterfly units and multipliers required is the same as that of Radix-2 MDC architecture, the memory (delay elements) and commutator requirement is very much reduced. A simple R2SDF architecture for $N = 16$ is shown in Fig. 7. An efficient FPGA implementation of generalized radix-2 SDF based FFT architecture by using coarse-grained hardware design is presented in Ref. 55. For OFDM baseband processing and better resource utilization, FFT/IFFT reconfiguration is achieved by FPGA-based Dynamic Partial Reconfiguration (DPR).[56] Radix-2[57] and radix-2^2[58] SDF-based pipelined FFT architectures are proposed for 16/64/256/1024 point with word-length scaling approach. A low power radix-2 64–1024 point SDF-based FFT architecture that shares hardware between 2 and 4 data streams is proposed.[59] The efficiency of the processor is doubled compared to the conventional SDF architectures.

4.4. *Radix 4 single-path delay feedback* (*R4SDF*)

Radix-4 single-path delay feedback (R2SDF) is similar to R2SDF in which radix-2 butterfly is replaced by radix-4 basic butterfly computation units. The number of multiplier units in radix-4 SDF architecture, as well as the utilization rate, is reduced to 25%, but hardware complexity is increased when compared with radix-2 SDF architecture. The generalized block diagram of radix-4 SDF architecture for 16 input data points is shown in Fig. 8. 1536 point FFT computation is achieved in variable length FFT processors[60,62] that suffers from very high latency and increased hardware costs. Three-stage SDF pipeline architecture and a hardware sharing mechanism are employed for FFT/IFFT processor to reduce the chip area.[61] Radix-2 SDF butterfly asymmetric with last three stages of radix-3 FFT butterfly is used for the

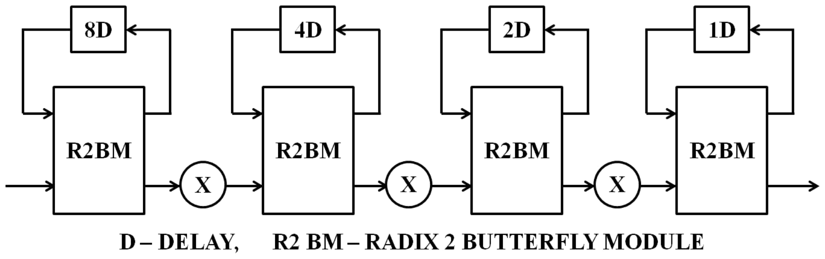


Fig. 7. Radix-2 single-path delay feedback for $N = 16$.

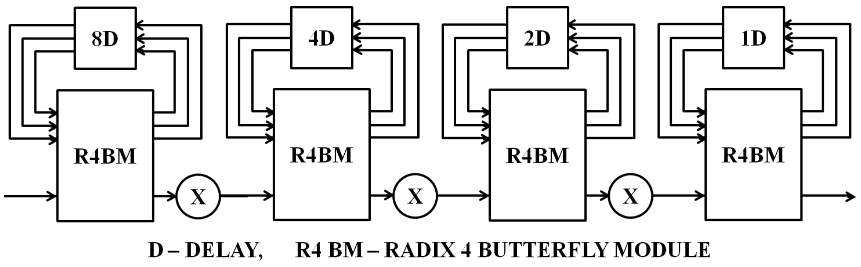


Fig. 8. Radix-4 single-path delay feedback for $N = 16$.

computation of radix-2 FFT. By using a simple multiplexer switch, the same architecture can be used to compute 128–2048/1536 FFT operations. Delay line buffers are further used to access the memory from idle processing elements to provide necessary buffering capacity for 1536 point FFT computation since it requires a larger buffer size.[62]

4.5. *Radix-4 single-path delay commutator* (*R4SDC*)

A simplified radix-4 butterfly is proposed in such a way that only one output is computed by using four input data from the input buffer at a time. This process is repeated in the same butterfly module (R4BM) until all the four same outputs are computed to achieve 100% utilization. Radix-4 single-path delay commutator (R4SDC) architecture needs additional computational units to provide the same input data four times. So a four path commutator (C) is used. The number of multipliers needed is much less than radix-4 MDC FFT architecture. A simplified R4SDC commutator architecture is shown in Fig. 9.

Multipath delay commutator and single path delay feedback are made parallel to achieve low latency, high throughput, and memory efficient 128–2048 point FFT/IFFT processor. An input selector or multiplexer switch is used to feed the input data to the appropriate stage.[63] A low-power 64-point pipeline FFT processor based on radix-4^3 butterflies, which computes four inputs in parallel, achieving 25% minimization in clock rate when compared with traditional MDC and SDF architectures, targeting IEEE 802.11a/g applications, is proposed.[64] R4SDC is

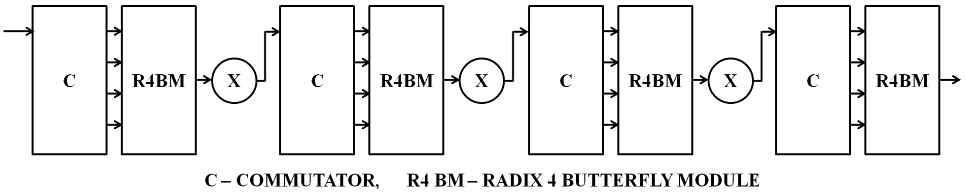


Fig. 9. Radix-4 single-path delay commutator for $N = 256$.

appropriate for MIMO-OFDM based applications for its low complexity and high hardware utilization.[65]

4.6. *Radix 2^2 single-path delay feedback* ($R2^2SDF$)

Radix-2^2 single-path delay feedback is an efficient evolving pipeline architecture for FFT computation. This architecture utilizes two radix-2 butterfly units with coefficient multipliers which are suitable for typical pipeline FFT implementation. The number of multipliers and memory requirement is very much reduced when compared with the previous designs. Radix-2^2 single-path delay feedback structure for $N = 256$ is shown in Fig. 10.

A mixed DIT/DIF FFT algorithm is used along with a modified SDF FFT architecture to obtain high utilization with same throughput and latency. In FFT computation, the final stage is computed by DIT, whereas, other stages are computed by DIF algorithm so as to maintain input and output in normal order that eliminates the usage of an additional clock. The modified SDF architecture computes radix-4 as radix-2^2, radix-8 as radix-2^3 and so on to reduce the number of complex adders and complex multipliers.[66] A mixed-radix variable length FFT processor is proposed based on Radix-2/2^2 butterfly structure by incorporating pipeline SDF method to minimize area with increased efficiency for floating point day-to-day FFT/IFFT applications.[67] A radix-2 algorithm based combined SDC-SDF architecture is presented with $\log_2(N-1)$ SDC stages, one SDF stage, and one-bit reverser to reduce up to 50% of complex multipliers.[68] The power consumption of

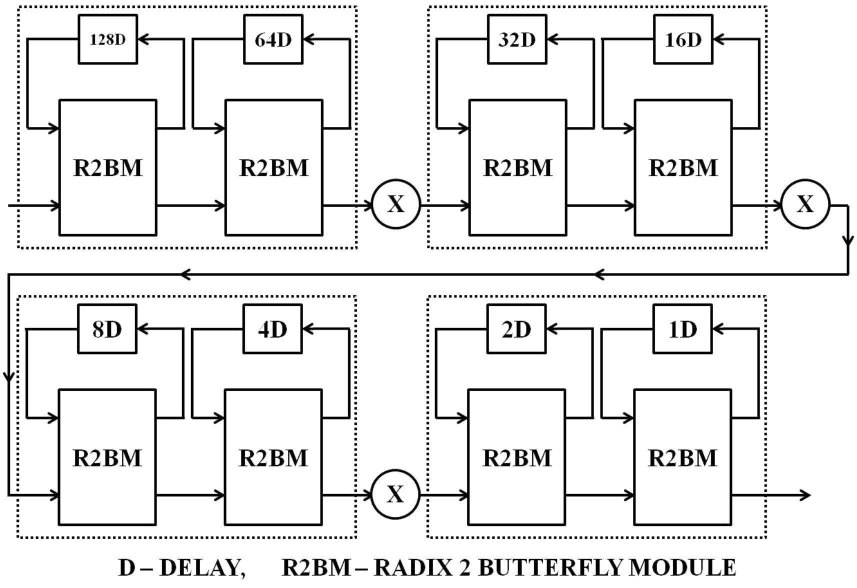


Fig. 10. Radix-2^2 single-path delay commutator for $N = 256$.

Table 3. Hardware requirements and throughput rate for various pipeline architectures.

Pipeline architecture	No. of complex adders	No. of complex multipliers	Memory requirement	Throughput rate
R2MDC	$2\log_2 N$	$\log_2(N-2)$	$(N-2)/5$	2R
R4MDC	$8\log_4 N$	$3\log_4(N-1)$	$5(N-4)/2$	4R
R2SDF	$2\log_2 N$	$\log_2(N-2)$	$N-1$	R
R4SDF	$8\log_4 N$	$\log_4(N-1)$	$N-1$	R
R4SDC	$3\log_4 N$	$\log_4(N-1)$	$N-1$	R
$R2^2$SDF	$4\log_4 N$	$\log_4(N-1)$	$N-1$	R

radix-4 SDF architecture is 11% lesser when compared to the radix-2 SDF architecture.[69]

The hardware comparison of pipeline architectures discussed above is shown in Table 3.[70]

5. Analysis and Discussions

The objective of this analysis is to select an area, power and delay efficient FFT/IFFT processor for MIMO-OFDM-based applications. Normalized area, energy, and throughput are the significant measures to evaluate the performances of various FFT/IFFT processors.[71] Several methods have been proposed to assess the normalized area and energy for FFT/IFFT processors,[71–74] but these evaluations were carried out for the same number of FFT/IFFT data points (N).

Since different FFT/IFFT sizes (N) were considered for our comparison, the evaluation proposed in Ref. 75 is optimal that can be given as,

$$A_{\text{Normalized}} = \frac{\text{Area}}{N \times \left(\frac{\text{Tech}}{0.18}\right)^2} \tag{8}$$

$$E_{\text{Normalized}} = \frac{\text{Power} \times \text{Exec.Time}}{N \times \left(\frac{V_{DD}}{1.8}\right)^2} \tag{9}$$

Table 4 compares the frequency, normalized area and normalized energy of the different FFT/IFFT architectures discussed in the literature. As previously stated, the processors have different fabrication CMOS library and various synthesis constraints, hence the FFT/IFFT processors with same values of N are considered for the discussion.

For FFT/IFFT processors with $N = 2048$, the MDF architecture in Ref. 75 has only 42.53% and 33.42% of the normalized area to that of SDF-based architectures in Refs. 76 and 77. The normalized area of MDF scheme in Ref. 73 is only 15.53% to that of MDC-based FFT/IFFT architecture in Ref. 44. It can be noted that the MDF architecture proposed in Ref. 75 saves more area, but execution time is almost

Table 4. Comparision among various FFT processors.

FFT processor	Architecture	Clock frequency (MHz)	Process (nm)	FFT size	Normalized energy (nJ)	Normalized area (mm^2)
Raja[13]	Cached FFT	251	180	64	53.85	9.98
Chen[23]	GHR	122.88	180	128–2048	4.12	2.44
Cho[30]	MR-Pipelined	310	90	512	1.31	6.093
Lai[37]	Recursive Radix-2^2	25	180	512	6.875	8.50
Kim[38]	MR-MDC	430	90	128–256	—	—
Lee[41]	MR-MDC	20	180	8192	5.4	0.33
Tang[42]	FRC-MDC	300	180	512	3.26	6.25
				256	6.00	
				128	8.85	
Yang[44]	MDC	40	90	2048	5.16	6.05
				1024	5.09	
				512	4.65	
				128	4.16	
Wang[48]	Radix-2^4–2^2–2^3 MDF	220	130	512	—	3.00
Yu[62]	SDF	40	90	2048	0.313	2.813
				1536	0.328	
				1024	0.535	
				512	0.602	
				256	1.03	
				128	1.83	
Kala[64]	R4-SDC	5	130	64	2.11	24.8
Sophy[67]	MR-SDF	50/100	90	2048	3.05	39.84
				512	11.3	
				128	592	
Chen[74]	Cached FFT	51	180	128–1024	2.06	2.05
Peng[75]	MDF	35	180	128–2048/1536	1.28	0.94
Patil[76]	SDF	40	180	128–2048	1.39	2.21
Huang[77]	Memory base	324	90	512	14.41	19.21
Tang[78]	Multi-data scaling	300	90	2048	0.83	2.265
		52		128	0.568	
Huang[79]	Cascaded	324	90	512	0.820	7.265
Ahamed[80]	Feed-forward	330	65	512	1.84	21.41

4.5 times higher than other MDC[44] and SDF[76]-based FFT/IFFT schemes. Since delay elements are more in 2048 MDC/MDF-based FFT/IFFT processors, the normalized energy dissipation is less in SDF scheme which is of 4–16 times lesser when compared to the other MDC,[44] MDF[75] and SDF[76] schemes. This variation is mainly due to the usage of Radix-3 FFT in SDF pipeline stages which reduces the number of computations. On the other hand, power consumption is greatly reduced. This trend in the area, power and delay can be carried on to the FFT/IFFT processors with $N = 1024, 512, 256$ and 128.

When considering the FFT/IFFT processors with size $N = 1024$, the normalized energy consumption by the SDF[62] processor is approximately 4 times higher than that of the Cached FFT processor,[73] and is 10.5%–41.7% to that of MDC,[44] GHR[62] and MDF[75] schemes. For the processors with FFT/IFFT size $N = 512$, the memory-based FFT architecture[77] consumes more power and occupies larger area as the number of memory elements is higher when compared with the pipelined-FFT[42,44,62,74,75] as well as mixed-radix pipelined FFT[30,67] As the clock frequency is higher, the normalized area and normalized energy also increase in feed-forward FFT architecture[81] when compared with other pipelined FFT/IFFT schemes. The normalized energy increases linearly with the normalized area as the number of logic and delay elements increase in line with the size of FFT/IFFT computation. The normalized area of mixed radix pipelined FFT[30] scheme is 15.30% of mixed radix SDF FFT scheme,[67] 28.46% of feed-forward architecture[80] and 31.71% of memory-based schemes.[77]

To reduce the delay, area and power effectively in an FFT/IFFT processor, the foremost solution is to reduce the memory usage. This can be achieved not only by selecting the proper FFT/IFFT architecture but also by adopting efficient input–output memory scheduling schemes and bit-reversal circuits. These techniques not only reduce the core size of the processor, they also reduce the power consumption and execution time of an FFT/IFFT scheme by reducing the number of accesses to the processor and main memory. Further, the reliability of core FFT processor can be improved by adopting error correction codes and Parseval checks.[82] Several strategies in implementing a reliable sequential/pipelined FFT core are discussed in Refs. 83–85.

6. Conclusion

The significance of various algorithms, architectures, and implementations of FFT/IFFT processor design for MIMO-OFDM based communication systems has been focused on in this review article. The aim of this article is to select an area efficient, low power, high-speed FFT/IFFT processor by adopting an architecture that has a desired twiddle factor multiplier circuit, an effective design of butterfly processing modules with appropriate feedback/feedforward delay units and input–output data scheduling schemes that have lesser access to memory. Higher-radix, as well as mixed-radix algorithms, based SDC pipelined FFT/IFFT architectures have less area and consume low-power at a higher throughput. It can be concluded that mixed-radix/higher-radix algorithm combined with Single-path Delay Commutator (SDC) architecture is appropriate for massive MIMO in 5G, optical OFDM, cooperative MIMO and multi-user MIMO based applications.

References

1. IEEE, IEEE Standard 802.11: The Wireless LAN Media Access Control (MAC) and Physical Layer (PHY) Specifications, IEEE Computer Society, Washington DC (1999).
2. IEEE, IEEE Standard 802.16: IEEE Standard for Local and Metropolitan Area Networks. Part16: Air interface for fixed broadband wireless access systems, IEEE Computer Society, Washington DC (2004).

3. IEEE, IEEE Standard 802.16: IEEE Standard for Local and Metropolitan Area Networks. Part16: Air interface for fixed broadband wireless access systems, IEEE Computer Society, Washington DC (2006).
4. Y. G. Li, J. H. Winters and N. R. Sollenberger, MIMO-OFDM for wireless communications: Signal detection with enhanced channel estimation, *IEEE Trans. Commun.* **50** (2002) 1471–1477.
5. A. Goldsmith, *Wireless Communications* (Cambridge University Press, 2005).
6. Asymmetric Digital Subscriber Line Transceivers 2 (ADSL2), *ITU-T Standard G.992.3*, 2005.
7. Very-High-Bit-Rate Digital Subscriber Line Transceiver 2(VDSL2), *ITUT Standard G.993.2*, Feb. 2006.
8. Q. Lu, X. Wang and J. C. Niu, A low-power variable-length FFT processor base on radix-24 algorithm, *Proc. IEEE Conf.* 2009, pp. 129–132.
9. J. G. Proakis and D. G. Manolakis, *Digital Signal Processing Principles, Algorithms and Applications*, 4th edn. (Pearson, 2007).
10. S. Josue Saenz, J. J. Raygoza, P. E. C. Becerra, A. S. O. Cisneros and J. R. Dominguez, FPGA design and implementation of radix-2 Fast Fourier Transform algorithm with 16 and 32 points, *Proc. IEEE International Autumn Meeting on Power, Electronics and Computing (ROPEC)* (Ixtapa, 2015), pp. 1–6.
11. A. Tiwari, S. Phapal, D. Kadam, S. Sivanantham and K. Sivasankaran, FPGA implementation of FFT blocks for OFDM, *Proc. Online International Conf. Green Engineering and Technologies (IC-GET)* (Coimbatore, 2015), pp. 1–4.
12. J. Raja and M. Kannan, VLSI implementation of high throughput MIMO OFDM transceiver for 4th generation systems, *Indian J. Eng. Materials Sci.* **19** (2012) 307–319.
13. J. Raja, P. Mangaiyarkarasi and K. Moorthi, Area efficient low power high performance cached FFT processor for MIMO OFDM application, *Int. J. Appl. Eng. Res.* **10** (2015) 11853–11868.
14. A. Kaivani and S. Ko, Floating-point butterfly architecture based on binary signed-digit representation, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **24** (2016) 1208–1211.
15. A. Kaivani and S. B. Ko, Area efficient floating-point FFT butterfly architectures based on multi-operand adders, *Electron. Lett.* **51** (2015) 895–897.
16. N. Sharma, P. R. Panda and F. Catthoor, Energy efficient FFT implementation through stage skipping and merging, *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis (CODES + ISSS)* (Amsterdam, 2015) pp. 153–162.
17. R. Neuenfeld, M. Fonseca and E. Costa, Design of optimized radix-2 and radix-4 butterflies from FFT with decimation in time, *IEEE 7th Latin American Symp. Circuits & Systems (LASCAS)* (Florianopolis, 2016), pp. 171–174.
18. A. A. Naoghare and A. V. Sakhare, Review on FFT architecture for real valued signals using Radix 2^5 algorithm, *Proc. Int. Conf. Pervasive Computing (ICPC)* (Pune, 2015), pp. 1–3.
19. A. Chinnapalanichamy and K. K. Parhi, Serial and interleaved architectures for computing real FFT, *Proc. IEEE Int. Conf. Acoustics, Speech and Signal Process. (ICASSP)*, South Brisbane, QLD, 2015, pp. 1066–1070.
20. A. R. Kuralkar, Design of fast fourier transform using processing element for real valued signal, *Proc. Int. Conf. Commun. Signal Process. (ICCSP)* (Melmaruvathur, 2015), pp. 1446–1448.
21. P. K. Meher, B. K. Mohanty, S. K. Patel, S. Ganguly and T. Srikanthan, Efficient VLSI architecture for decimation-in-time fast fourier transform of real-valued data, *IEEE Trans. Circuits Syst. I Regular Papers* **62** (2015) 2836–2845.

22. Y. Lao and K. K. Parhi, Canonic real-valued radix-2n FFT computations, *Proc. 49th Asilomar. Conf. Signals, Systems and Computers* (Pacific Grove, CA, 2015), pp. 441–446.
23. J. Chen, J. Hu, S. Lee and G. E. Sobelman, Hardware efficient mixed radix 25/16/9 FFT for LTE systems, *IEEE Trans. VLSI Syst.* **23** (2015) 221–229.
24. C.-F. Hsiao, Y. Chen and C.-Y. Lee, A generalized mixed-radix algorithm for memory-based FFT processors, *IEEE Trans. Circuits Syst. II Exp. Briefs* **57** (2010) 26–30.
25. N. Laguri and K. Anusudha, VLSI implementation of efficient split radix FFT based on distributed arithmetic, *Proc. Int. Conf. Green Comput. Electrical Eng.* (2014), pp. 1–5.
26. A. S. B. Paul, S. Raju and R. Janakiraman, Low power reconfigurable FP-FFT core with an array of folded DA butterflies, *Eurasip J. Adv. Signal Process.* **144** (2014) 1–17.
27. A. Kharin, S. Vityazev, V. Vityazev and N. Dahnoun, Parallel FFT implementation on Tms320c66x multicore DSP, *2014 6th European Embedded Design in Education and Research Conference (EDERC)*, Milano 1–5 (2014), pp. 46–49, doi: 10.1109/ICDC-Syst.2014.6926131.
28. S. Ranganathan, R. Krishnan and H. S. Sriharsha, Efficient hardware implementation of scalable FFT using configurable radix-4/2, *Proc. 2nd Int. Conf. Devices, Circuits and Systems* (2014), pp. 1–5, doi: 10.1109/ICDCSyst.2014.6926131.
29. Z. Qian, N. Nasiri, O. Segal and M. Margala, FPGA implementation of low-power split-radix FFT processors, *Proc. 24th Int. Conf. Field Programmable Logic and Applications* (2014), pp. 1–2.
30. T. Cho and H. Lee, A high-speed low-complexity modified radix-2^5 FFT processor for high rate WPAN applications, *IEEE Trans. Very Large Scale Integr.* **21** (2013) 187–191.
31. W. Hussain, X. Chen, G. Ascheid and J. Nurmi, A reconfigurable application-specific instruction-set processor for fast fourier transform processing, *Proc. IEEE 24th Int. Conf. Application–Specific Systems, Architecture and Processors* (2013), pp. 339–345.
32. M. Ayinala, Y. Lao and K. K. Parhi, An in-place FFT architecture for real-valued signals, *IEEE Trans. Circuits Syst. II* **60** (2013) 652–656.
33. D. Revanna, O. Anjum, M. Cucchi, R. Airoldi and J. Nurmi, A scalable FFT processor architecture for OFDM based communication systems, *Int. Conf. Embedded Computer Systems: Architectures, Modeling, and Simulation* (Agios Konstantinos, 2013), pp. 19–27, doi: 10.1109/SAMOS.2013.6621101
34. T. Patyk, D. Guevorkian, T. Pitkanen, P. Jaaskelainen and J. Takala, Low-power application specific FFT processor for LTE applications, *Proc. Int. Conf. Embedded Computing Systems: Architecture, Modelling and Simulation* (Agios Konstantinos, 2013), pp. 28–32, doi: 10.1109/SAMOS.2013.6621102.
35. H. Corporal, *Microprocessor Architectures: From VLIW to TTA* (John Wiley & Sons, Chichester, UK, 1997).
36. S. S. Wang and C. S. Li, An area-efficient design of variable-length fast fourier transform processor, *J. Signal Process. Syst.* **51** (2007) 245–256.
37. S. C. Lai *et al.*, Hybrid architecture design for calculating variable-length fourier transform, *IEEE Trans. Circuits Syst. II: Exp. Briefs* **63** 279–283.
38. H. S. Kang, S. H. Chang, I. K. Hwang and J. K. Lee, A design and implementation of 32-paths parallel 256-point FFT/IFFT for optical OFDM systems, *Proc. 18th Int. Conf. Adv. Commun. Technol. (ICACT)* (Pyeongchang, 2016), pp. 417–421.
39. N. Sarode, R. Atluri and P. K. Dakhole, Mixed-radix and CORDIC algorithm for implementation of FFT, *Proc. Int. Conf. Commun. Signal Process. (ICCSP)* (Melmaruvathur, 2015), pp. 1628–1634.
40. E. J. Kim and M. H. Sunwoo, High speed eight-parallel mixed-radix FFT processor for OFDM systems, *Proc. IEEE Int. Symp. Circuits and Systems* (2011), pp. 1684–1687.

41. S. Y. Lee, C. C. Chen, C. C. Lee and C. J. Cheng, A low-power VLSI architecture for a shared-memory FFT processor with a mixed-radix algorithm and a simple memory control scheme, *Proc. IEEE Int. Symp. Circuits and Systems* (Island of Kos, 2006), pp. 157–160.
42. S. N. Tang, C. H. Liao and T. Y. Chang, An area- and energy-efficient multimode FFT processor for WPAN/WLAN/WMAN systems, *IEEE J. Solid State Circuits* **47** (2012) 1419–1435.
43. S. Zhou, X. Wang, J. Ji and Y. Wang, Design and implementation of a 1024-point high-speed FFT processor based on the FPGA, *Proc. 6th Int. Cong. Image and Signal Processing* (2013), pp. 1112–1116.
44. K. J. Yang, S. H. Tsai and G. C. H. Chuang, MDC FFT/IFFT processor with variable length for MIMO-OFDM systems, *IEEE Trans. VLSI Syst.* **21** (2013) 720–731.
45. A. Amjadha, E. Konguvel and J. Raja, Design of multipath delay commutator architecture based FFT processor for 4th generation systems, *Int. J. Comput. Appl.* **89** (2014) 23–28.
46. M. Garrido, J. Grajal, M. A. Sanchez and O. Gustafsson, Pipelined radix-2^k feedforward FFT architectures, *IEEE Trans. VLSI Syst.* **21** (2013) 23–32.
47. Y. W. Lin and C. Y. Lee, Design of an FFT/IFFT processor for MIMO-OFDM systems, *IEEE Trans. Circuits and Syst. I* **54** (2007) 807–815.
48. C. Wang, Y. Yan and X. Fu, A high-throughput low-complexity radix-2^4-2^2-2^3 FFT/IFFT processor with parallel and normal input/output order for IEEE 802.11ad systems, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **23** (2015) 2728–2732.
49. J. Wang, C. Xiong, K. Zhang and J. Wei, A mixed-decimation MDF architecture for radix-2^k parallel FFT, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **24** (2016) 67–78.
50. L. P. Thakare and A. Y. Deshmukh, Area efficient FFT/IFFT processor design for MIMO OFDM system in wireless communication, *Proc. 7th Int. Conf. Emerging Trends in Engineering & Technology (ICETET)* (Kobe, 2015), pp. 10–13.
51. H. F. Luo, M. D. Shieh and K. H. Lee, A radix-$2/3/2^2/2^3$ MDC architecture for variable-length FFT processors, *Proc. Int. Conf. Consumer Electronics - Taiwan (ICCE-TW)* (Taipei, 2015), pp. 180–181.
52. A. X. Glittas, M. Sellathurai and G. Lakshminarayanan, A normal I/O order radix-2 FFT architecture to process twin data streams for MIMO, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **24** (2016) 2402–2406.
53. S. He and M. Torkelson, A new approach to pipeline FFT processor, *Proc. IEEE Parallel Processing Symp.* (1996), pp. 766–770.
54. S. Sukhsawas and K. Benkrid, A high-level implementation of a high performance pipeline FFT on Virtex-E FPGAs, *Proc. IEEE Computer Society Annual Symp. VLSI (ISVLSI '04)* (2004), pp. 229–232.
55. I. Ali Qureshi, F. Qureshi and G. M. Shaikh, Efficient FPGA-mapping of 1024 point FFT pipeline SDF processor, *Proc. Sixth Int. Symp. Parallel Architectures, Algorithms and Programming* (2014), pp. 29–34.
56. M. L. Ferreira, A. Barahimi and J. C. Ferreira, Dynamically reconfigurable FFT processor for flexible OFDM baseband processing, *Proc. Int. Conf. Design and Technology of Integrated Systems in Nanoscale Era (DTIS)* (Istanbul, Turkey, 2016), pp. 1–6.
57. Y. Xie, Y. K. Feng, C. Yang, Y. Z. Xie and H. Chen, Design of a large point FFT processor with configurable transform length, *Proc. IET Int. Radar Conf.* (Hangzhou, 2015), pp. 1–5.

58. C. Yang, Y. Z. Xie, L. Chen, H. Chen and Y. Deng, Design of a configurable fixed-point FFT processor, *Proc. IET Int. Radar Conf.* (Hangzhou, 2015), pp. 1–4.
59. M. Dali, R. M. Gibson, A. Amira, A. Guessoum and N. Ramzan, An efficient MIMO-OFDM radix-2 single-path delay feedback FFT implementation on FPGA, *Proc. Conf. Adaptive Hardware and Systems (AHS)* (Montreal, QC, 2015), pp. 1–7.
60. S. Y. Peng, K. T. Shr, C. M. Chen and Y. H. Huang, Energy-efficient 128/2048/1536-point FFT processor with resource block mapping for 3 GPP-LTE system, *Proc. Int. Conf. Green Circuits Syst.* (2010), pp. 14–17.
61. C. H. Yang, T. H. Yu, and D. Markovic, Power and area minimization of reconfigurable FFT processors: A 3 GPP-LTE example, *IEEE J. Solid-State Circuits* **47** (2012) 757–768.
62. C. Yu and M. H. Yen, Area-efficient 128- to 2048/1536-point pipeline FFT processor for LTE and mobile WiMAX systems, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **23** (2015) 1793–1800.
63. T. Adiono and R. Mareta, Low latency parallel-pipelined configurable FFT-IFFT 128/256/512/1024/2048 for LTE, *Proc. IEEE 4th Int. Conf. Intelligent and Advanced Systems* (Kuala Lumpur, 2012), pp. 768–773.
64. S. Kala, S. Nalesh, S. K. Nandy and R. Narayan, Design of a low power 64 point FFT architecture for WLAN applications, *Proc. 25th Int. Conf. Microelectronics* (2013), pp. 1–4.
65. S. Singh and J. Kedia, Pipelined FFT architectures: A review, *Proc. Int. Conf. Electrical, Electronics, Signals, Communication and Optimization (EESCO)* (Visakhapatnam, 2015), pp. 1–5.
66. S. Lee and S. C. Park, Modified SDF architecture for mixed DIF/DIT FFT, *Proc. IEEE Int. Symp. Circuits and Syst.* (New Orleans LA, 2007), pp. 2590–2593.
67. A. Sophy, R. Srinivasan, J. Raja and S. Anand Ganesh, Variable length floating point FFT processor using radix-2^2 butterfly elements, *Int. J. Eng. Technol.* **6** (2014) 764–771.
68. Z. Wang, X. Liu, B. He and F. Yu, A combined SDC-SDF architecture for normal I/O pipelined radix-2 FFT, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **23** (2015) 973–977.
69. P. A. Sophy, R. Srinivasan, J. Raja and M. Avinash, Analysis and design of low power radix-4 FFT processor using pipelined architecture, *Proc. Int. Conf. Comput. Commun. Technol. (ICCCT)* (Chennai, 2015), pp. 227–232.
70. B. Fu and P. Ampadu, An area efficient FFT/IFFT processor for MIMO-OFDM WLAN 802.11n, *J. Signal Process. Syst.* **56** (2008) 59–68.
71. B. M. Baas, A low-power, high-performance, 1024-point FFT processor, *IEEE J. Solid-State Circuits* **34** (1999) 380–387.
72. Y. T. Lin, P. Y. Tsai and T. D. Chiueh, Low-power variable-length fast fourier transform processor, *IEEE Proc. Comput. Digit. Tech.* **152** (2005) 499–506.
73. T. D. Chiueh and P. Y. Tsai, *OFDM Baseband Receiver Design for Wireless Communications* (Wiley, New York, 2007).
74. C. M. Chen, C. C. Hung and Y. H. Huang, An energy-efficient partial FFT processor for the OFDMA communication system, *IEEE Trans. Circuits Syst. II Exp. Briefs* **57** (2010) 136–140.
75. S. Y. Peng, K. T. Shr, C. M. Chen and Y. H. Huang, Energy-efficient 128/2048/1536-point FFT processor with resource block mapping for 3GPP-LTE system, *Proc. Int. Conf. Green Circuits Syst.* (2010), pp. 14–17.
76. M. S. Patil, T. D. Chhatbar and A. D. Darji, An area efficient and low power implementation of 2048 point FFT/IFFT processor for mobile WiMAX, *Proc. Int. Conf. Signal Process. Commun.* (2010), pp. 1–4.

77. S. J. Huang and S. G. Chen, A green FFT processor with 2.5-GS/s for IEEE 802.15.3c (WPANs), *Proc. Int. Conf. Green Circuits Syst.* (2010), pp. 9–13.
78. S. N. Tang, J. W. Tsai and T. Y. Chang, A 2.4-GS/s FFT processor for OFDM-based WPAN application, *IEEE Trans. Circuits Syst.-II: Exp. Briefs* **57** (2010) 451–455.
79. S. J. Huang and S. G. Chen, A high-throughput radix-16 FFT processor with parallel and normal input/output ordering for IEEE 802.15. 3c systems, *IEEE Trans. Circuits Syst. I: Regular Papers* **59** (2012) 1752–1765.
80. T. Ahmed, M. Garrido and O. Gustafsson, A 512-point 8 parallel pipelined feedforward FFT for WPAN, *Proc. 45th Conf. Signals, Systems and Computers (ASILOMAR)* (2011), pp. 981–984.
81. Z. Gao *et al.*, Fault tolerant parallel FFTs using error correction codes and parseval checks, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **24** (2016) 769–773.
82. P. Y. Tsai and C. Y. Lin, A generalized conflict-free memory addressing scheme for continuous-flow parallel-processing FFT processors with rescheduling, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **19** (2011) 2290–2302.
83. M. Kannan, E. Konguvel, J. Madhumitha, M. Mohamed Kamil Ashiq and K. Abhishek, FPGA implementation of FFT architecture using modified Radix-4 algorithm, *Asian J. Res. Soc. Sci. Humanit.* **7** (2017) 47–57.
84. M. Kannan and S. K. Srivatsa, Low power hardware implementation of high speed FFT core, *J. Comput. Sci.* **3** (2007) 376–382.
85. M. Kannan and S. Srivatsa, Hardware implementation low power high speed FFT core, *Int. Arab J. Inf. Technol.* **6** (2009) 1–6.

A COMPARATIVE ANALYSIS OF DES, AES AND RSA CRYPT ALGORITHMS FOR NETWORK SECURITY IN CLOUD COMPUTING

M.Kannan[1], Dr.C.Priya[2], S.VaishnaviSree[3]
[1]Research Scholar,[2]Associate Professor,[3]Research Scholar
[1]Department of Computer Science,
[1]School of Computing Sciences, Vels Institute of Science, Technology and Advanced Studies (VISTAS), Pallavaram, Chennai, Tamil Nadu, India.

DOI: http://doi.one/10.1729/Journal.19997

Abstract: In the modern existence Data communication is a copious key to share the data from one faction to another faction through the nexus or transmission medium using few social media like Whatsapp, Messanger, facebookin between this,third factionmight obtain or retrieve the entropy or data. Cryptography is mainly used for authentication scope. For this equip Cryptography concept and algorithms are applied in an every social media (ex: twitter, Facebook, Instagram, LinkeIn), agencies, institutions, industries, etc. The view of DES, AES and RSA is one the major concern of this paper to explore, how these crypt theoriesare secure the data by the use of encryption and decryption.This paper takes a simple literature survey and also comparing the relationship between DES, AES and RSA crypt notion. In which they are one of the significant modules to secure and/or the messages using crypt file keys. In these, DES, AES and RSA concepts are used to protect the file with the service of encryption and decryption affair. The main intent of the DES, AES and RSA in cryptography is to secure the files.According to this all the key size is static.In this paper describes how cryptosystems are protecting the plaintext from the unsupported client.

***IndexTerms* - DES, AES, RSA, Encryption, Decryption, Secret Key and Cryptography.**

I. Introduction

Cryptography is the proficiency used for secure communication without interfering third parties called opponent.Cryptographic algorithms are devising around **computational inclemency (hardness) notion.** Initially cryptography has supplementary paramount goals such as authentication, confidentiality, non-repudiation, integrity, availability. Advanced Encryption Standard(AES), Data Encryption Standard (DES), and Rivest, Shamir Adleman (RSA) is the main solicitude of this paper. And also in this paper have roughlycompare the concepts among AES, DES, and RSA. In which, Advanced Encryption Standard (AES) was nominated by Rijndael, is an enumeration for the encryption of electronic data.Which is a block cipher technique,the size of the key is 128-bits and also is a symmetric key. The key is a salient featureto excrement the file from one subscriber to another subscriber become confident. This is the major part of the cryptography globe.Whereas Data Encryption Standard (DES) was lodged by NBS/NIS (National Bureau of Standards/ National Institute of Standards) in1977. Which is also be a block cipher technique with 64-bit key. DES has a six rounds, using these roundsthe plaintext (original message) is metamorphosed into cipher text (other text), finally it will crown their tour to constitute the 64-bit plaintext. And Ron Rivest,AdiShamir Adlemanand Leonard Adleman(RSA) is the first public key cryptosystem in 1977. In the RSA, has separate encryption and decryption key which has disguised from one another. While comparing with other cryptographic algorithm, it is a pretty slow one.There are two componentsessential to encrypt the data by the use of cryptographic algorithm and akey. The Cryptographic algorithm kept a key as a stealthy so any other information should not be published.The key is a very large number that should be impossible to presume.

II. Related Works

To give more prospective about the comparability of those concepts, this section of the paper mostly discusses the literature survey and comparative analysis of cryptography. In this survey, we multitude figures about those cryptographic concepts for authentication and also protected from the third parties. According to these reason AES and DES are performed in many sites. Because which makes more confidence. Which means it will share the message only with a confident person.

III. Cryptography Basic Model

Cryptography involves provoke codes that allow information to be kept secret from the unauthorized user. All the information is kept secure and protected from the third party. For example, if Alice posts some paramount message to Bob, the third person might be read that information without the license of Alice and Bob. So the information might be unsecure. So that cryptography takes the charge to secure the secret. For this cryptography concept is used in many places.

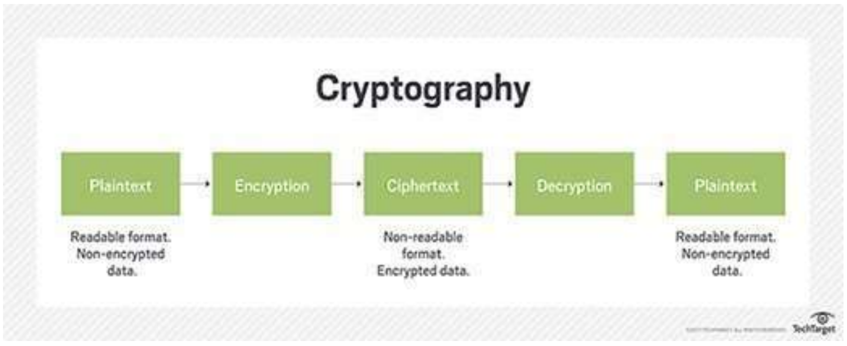

Cryptography
Plaintext
Encryption
Ciphertext
Decryption
Plaintext
Readable format. Non-encrypted data.
Non-readable format. Encrypted data.
Readable format. Non-encrypted data.
TechTarget

Fig 1: Cryptography Basic Function (Model)

In the above illustration cryptography basic function has some functions to change the message fromone text to other text using encryption and decryption standards for secure the information. According to this figure the plain text is changed into cipher text with the help of encryption, simultaneously the cipher text isagain converted into plaintext (User Readable Form) with the use of decryption.Thisis the basic concept of cryptography.Further, will move to the conceptsof DES, AES and RSA one after another viathis paper.Generally Cryptography has many types to protect the information; some of them are, Symmetric, asymmetric and block cipher & stream cipher cryptanalysis. These are some cryptography key which is also some types of cryptography, commonly used for file protection from the unknown person or from a third party.

IV.Hypothesis of AES, DES and RSA

Cryptography uses many important concepts to protect the message from an unprotected user. From that some of them are discussed below:

4.1 Structure of Data Encryption Standard (DES)

The Data Encryption Standard (DES) was developed by the National Bureau of Standards (NBS) in 1970$_s$. The connotation of DES is to confer a standard and confident method to protect the vulnerable information and unclassified data.Each 64 bits of data is ingeminatefrom 1 to 16 times (which means 16 rounds). The following illustration is the Data Encryption Standard structure which carries 16 rounds to protect the original text from the unsubscribed user using S-boxes.

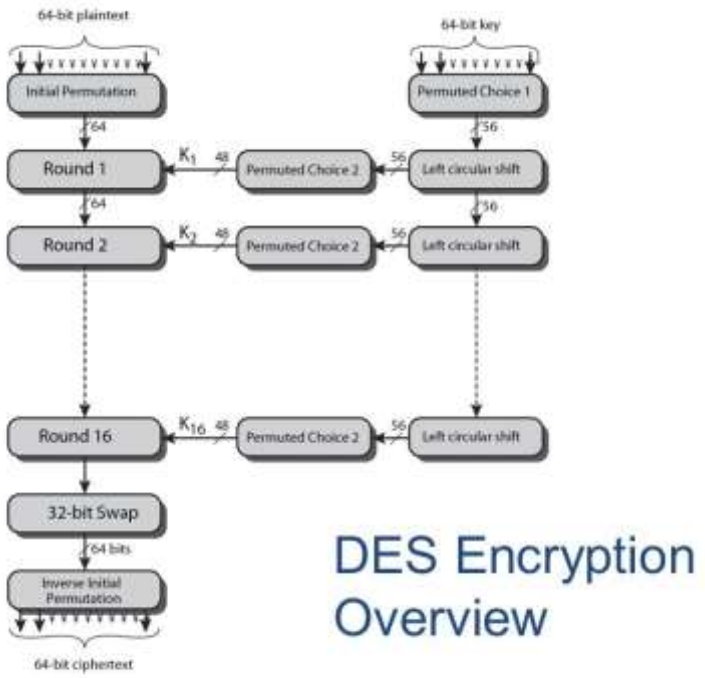


Fig 2: Structure of DES

Inthis diagram first the original message, which is 64-bit PT (Plain Text) was sent to the Initial Permutation after that the message will move on to all the 16-rounds of checking the PT in to bit by bit. Then it will complete their tour.Simultaneously the key will work, remember, 64 bit key is changed into 56-bit key since the actual size of a key is 56-bit. It will appertain through the Permuted function. After this process the key is converted into 48 bit, then it will be passedto the final round and vice versa. After the completion of 16 rounds the plain text is swapped into 32-bit. Finally the swapped message is sent to the Inverse Initial

Permutation (IP^{-1}), after the IP^{-1} function the 64-bit cipher text proclaims to the opponent. Both Permutation and Left Circular Shift uses the sub-key K_i at each and every level.This is the universal view of DES.

4.1.1 Solitary / Single Round of DES

The same procedure is as follows at every round which means every 16 rounds.

Steps:

a. Divide 64-bits into two 32-bit from both sides (left and right).
b. Then the R input is expanding to 48-bits by using Expansion Permutation (E).
c. Expansion Permutation (E) reveals the output in which they perform the XOR operation with key K_i.

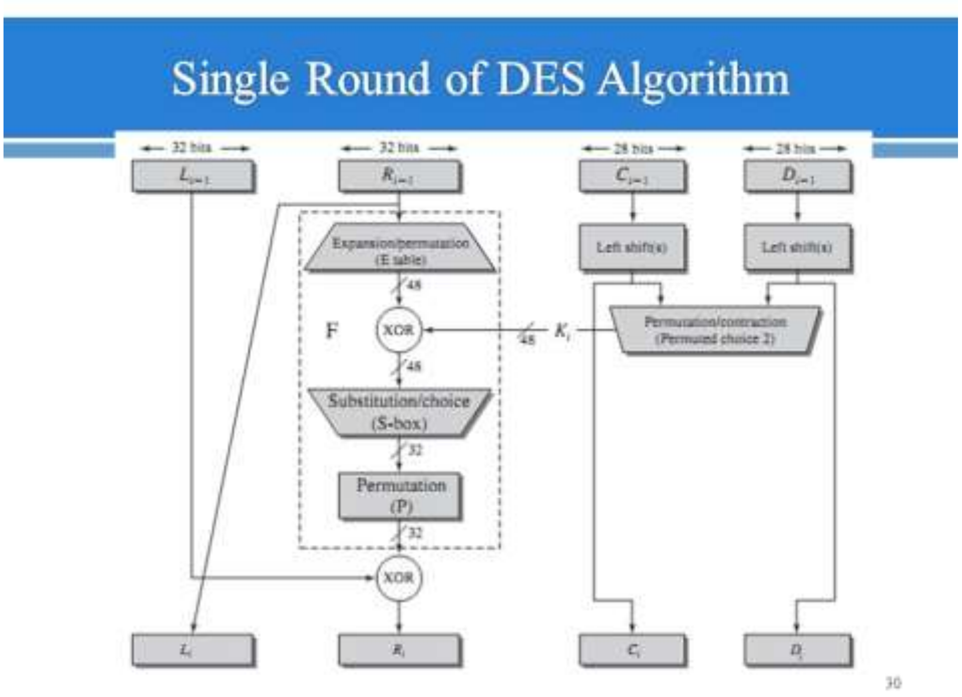


Fig 2(a): Single Round of DES

d. After the XOR operation,it produces the result which has passed to the S-box which givesas 32-bit output.
e. The Permutation Function (P) permuting the resultant value once again for the clarification.
f. Simultaneously 56-bit key will also be divided into two halves as 28-bits which give the performance using Left Circular Shift and Permutation, after thisit will reduce the size of the key and which spread the result into every round.

4.1.2 STRENGTH OF DES:

Some of the important strengths in DES are,

a. Data Encryption Standard has a strong avalanche effect.
b. Using this DES, Brute-force attack is not possible.
c. Timing attack is also difficult.

4.2 OVERVIEW OF ADVANCED ENCRYPTION STANDARD (AES)

AES is a symmetric block cipher also called as private, which uses the same key for the process of encryption and decryption. Advanced Encryption Standard (AES) was suggested by Rijndael. AES will encrypt and decrypt the block size of 128 bits using diverse cipher keys of 128,192, and 256 bits. Generally AES accommodates four alternate transformations such as Sub-Bytes, Shift Rows, Mix-Columns, and Add Round Key. In this paper focuses the concept around thecryptographic function in which how they are officiate in the network area. At first AES will monitor the block and key size of the text, then it will pertain the data into the AES algorithm. Which means it first evaluate the volume to insert the data. Data Encryption Standard (DES) involves six rounds to change from the plain text to cipher text, whereas, Advanced Encryption Standard (AES) involves 10 rounds to change from the plaintext to cipher text. This is the dissimilarity between these two systems.Each round is quite similar to convert the text. AES works under the encryption and decryption manner by the rule of Expand key. For encrypting, the algorithm will work from round 1 to 10. For decrypting, the algorithm will work in a reverse manner.Not only encrypted plain text128-bit key is also constituted as square matrices of bytes. In this section the key is represented as words W_0 to W_{43}. Therefore 44 words, each word contains 4-bytes.Finally the key matrix is stored as words (Expanded Key).

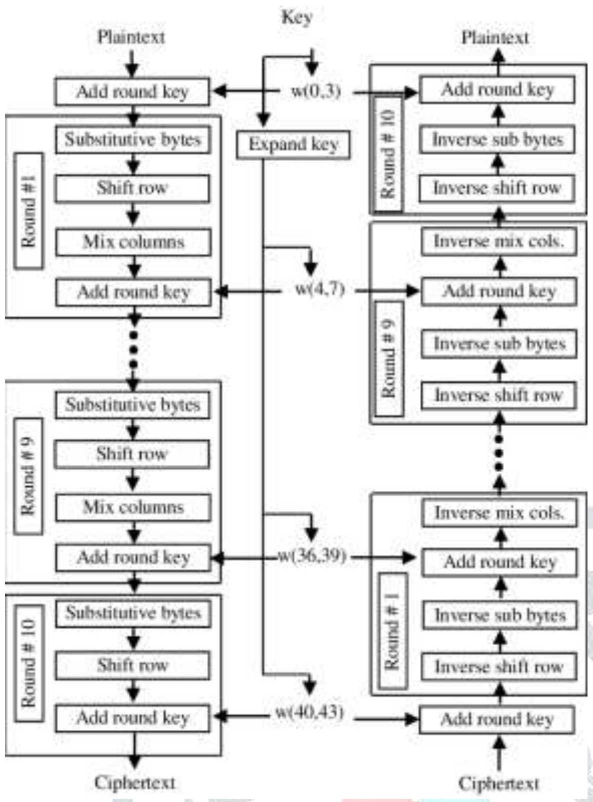


Fig 3: Block Diagram for AES Encryption and Decryption

The above figure (3) suppressesa few blocks or steps to do the encryption. AES takes 128-bit blocks as an input for encrypting the data or plaintext, which can be represented by square matrix. The particular matrix is copied into state array which performsome modification at each level of encryption.Which means the state array is replicated from an input matrix. After this process, the final result is sent to the out matrix. And also out matrix is a depot to save the output or the result.

A. Add Round Key: Add round key performs bitwise XOR operation between the state array and the resulting round key that is the output of the key expansion algorithm.

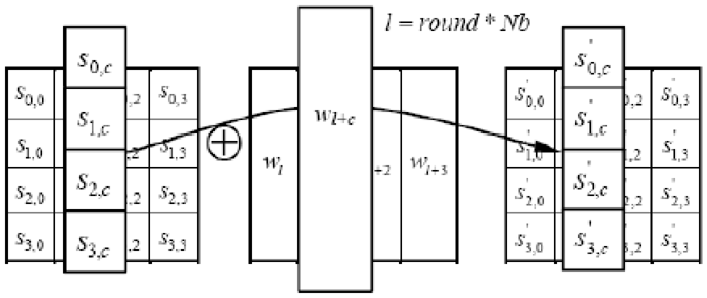


Fig 3(a): AES – Add Round Key

B. **Substitute Bytes:** Substitute Byte is also called as a Sub Bytes transformation which is one of the transformation technique which is a nonlinear byte substitution and also a simple table lookup technique. The implementation of this transformation is simple. Sub Bytes transformation is an S-Box which consist of 16*16 matrix in which the S-Box is used for both forward and inverse transformation.

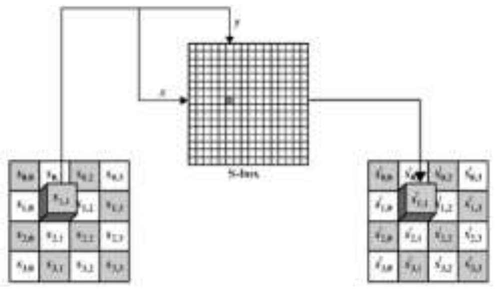


Fig 3 (b): AES – Sub Bytes

C. **Shift Row Transformation:** Which is nothing but, this transformation shifts the rows in a cyclic manner at each and every row to the left. It means each row is shifted to a different offset. Shift row transformation has some rules for shifting rows which are given below:

1. According the rule of Shift Row, the first step, the first row should not be shifted.
2. The shifting process will be started at the second step. In the second row the byte is shifting at one byte position to the left.
3. Likewise, the third row will be shifting at the two byte position to the left.
4. Fourth row is third position and viceversa.

Example: Shifting Rows at Row by Row

8A	**2F**	**16**	**32**
4D	**76**	**28**	**19**
DC	**17**	**0E**	**80**
A8	**DH**	**47**	**04**

$\downarrow$

8A	**2F**	**16**	**32**
76	**28**	**19**	**4D**
0E	**80**	**DC**	**17**
04	**A8**	**DH**	**47**

D. **Mix Column Transformation:**This transformation usesa GF multiplication method for evaluating the bit values. Such transformation is also called as Galoi's Field. GF performs the XOR operation (which is the bitwise operator) to calculate all the bit values using 1-bit left shift operation.

4.2.1 AES Advantages And Disadvantages

Advantages:

1. When comparing to other method the process of AES has more speed.
2. S-Box is used to abolish the symmetric.
3. Comprehensibility of description.
4. More fastened, so no one cannot hack or attack the personal information.

Drawbacks:

1. It uses an excessively potty algebraic structure.
2. Implementing with software is much complicated.

4.3. Application Of RSA

RSA is the first public key cryptosystem which is also referred as a symmetric key cryptosystem in cryptography. RSA involves single round to encrypt and decrypt the original text or a message or plaintext. RSA was invented by Ronald Rivest (Ron Rivest), Adi Shamir, and Leonard Adleman in the year of 1978.It is an algorithm used for public key encryption.Few set of famous security system which is composed of 3 phases: 1) Prime Key Generation, 2) Encryption and 3) Decryption. The below figure is a basic encryption and decryption process which consist some key to encrypt or decrypt the message or to convert from plain to cipher by the use of cryptography algorithm.

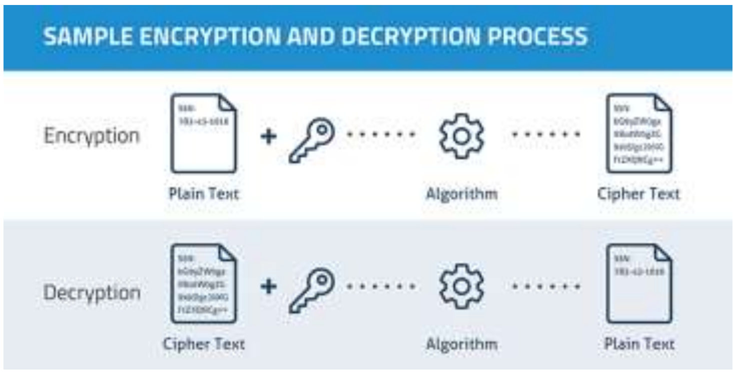


Fig 4 (a): Encryption and Decryption

The RSA system of encryption and decryption is portrayed in the succeeding sketch:

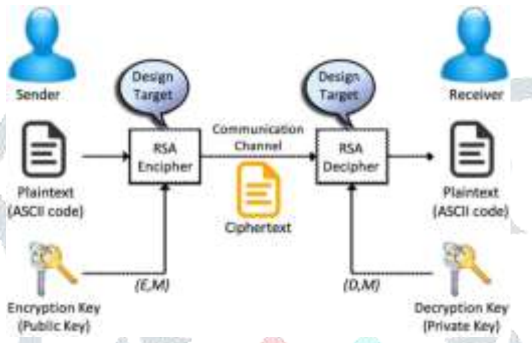


Fig 4 (b): Encryption and Decryption Using RSA

In the above sketch, the sender sends X the confidential message to receiver Y in between the process of encryption and decryption with the safeguard keys through the communication channel. At first, X sends the spontaneous message or plain text is sent to encipher which process the encryption operation using the sender's public key, after encrypting, the encrypted file is sent to decipher for the decryption process via the communication channel.In this (Encipher) process, the filemodifies into another type of text which is called cipher text. The RSA decipher will decrypt the cipher text using the sender's private key. Finally, the originalsafeguard decrypted plain text is received by the user Y.

4.3.1 Asymmetric Key

Asymmetric encryption is one of the discrete images of a crypto system. It has two sections, such as Encryption and Decryption. RSA uses one public and private key for securing and transferring the data from the unauthorized users. It is also called as key-pair. This pair of key is used to transferthe data within a time period. This function is also called as public-key encryption.

4.3.2 Digital Signatures

Digital Signatures (DS) are built from asymmetric cryptography and it can confer credence of manifest to radix, identification and designation of an electronic deed. Digital Signature is used to check or anauthenticate a message. Analogously, DS is a technique which binds an entity to the Digital Data. It will secure message from the third party. In this Digital Signature, hashing function is the one of the main prime to verifying a message and protect the document.Since, DS is invented by private key. Some of the DS significances are–

a. Message Authentication
b. Data Integrity
c. Non-Repudiation

4.3.3 RSA Advantages and Disadvantages

Advantages:

1. Safe and Secure
2. No one should nor crack the file

Disadvantage:

1. The RSA process might be slow while encrypting too long data.

V.SECURITY MANAGEMENT AREAS FOR CLOUD COMPUTING

Security Management (SM) includes functions that control and protect access to organization's resources, information, data, and IT services in order to ensure confidentiality, integrity, and availability. Security management functions are methods for authentication, authorization, encryption, etc. Unfortunately, the expanded definitions and standards around security management do not define a common set of security management areas.

The Security Management Infrastructure approach, which is also called Enterprise Security Management (ESM), is known to serve as comprehensive security architecture for our research. This approach of EU, NATO, the UK, and the USA, contains security management functions, such as Identity Management, Privilege Management Metadata Management, Policy Management, and Crypto Key Management. In addition, there are several sources that describe Cloud computing security areas. However, they differ in their compliance with necessary security management functional areas and collaboration aspects thatcan be used for a comprehensive Cloud security management For example, the management of meta-data or configuration management of security capabilities are not covered. Mainly they focus on Identity, Privilege, Access, and Crypto Key Management.Based on the presented sources above, we present the following ten security management areas for Cloud computing, that can be briefly described as follows:

***Identity Management** is* the ability to confirm and manage the life cycle of an assured identity (human/device/process) Amalgamated Identity Management provides end users with secure access across multiple external applications through coalesce single sign-on.

Credential Managementis the ability to manage the life cycle of digital credentials. Examples of credentials include certificates, private keys, user IDs, and passwords. The credential management is also responsible for verifying the authenticity of credentials.

Attribute Managementis the ability to manage the assigned properties of entities. An attribute is a specification which defines a property of an entity. The key elements are: publishingof attribute requirements, support for user consent, and common attribute policies. Furthermore, it is responsible for requesting the new attribute and associated values from service upon attempts to access the service.

Privilege Managementis the ability to manage permissions to perform an action. It is designed to address the challenges of managing what people can access and giving control of that process to those in the departments who make the decisions.

Digital Policy Management (DPM) mitigates compliance and intellectual property risks through efficient and effective digital policy management. Itis the ability to generate, convert manage and replace digital policies. Digital policies are those that are in machine-specific languages and can be used to guide the behavior of systems in an automated or semi- automated manner.

Configuration Management (CM)is a field ofmanagement that focuses on establishing and maintaining consistency of a system or product's performance and its functional and physical attributes with its requirements, design, and operational information throughout its life. It manages the security-related configuration items, such as defining, controlling, ordering, and loading of configuration data for services.

Cryptographic Key Managementencompasses all of the activities and to provide protocol security services, especially integrity, authentication and confidentiality.

Metadata Managementinvolves storing information about other information. It is an important part of enterprise information management (EIM). It is the ability to generate and manage all security-relevant metadata schema and values over their life-cycle.

Audit Managementis the ability that establishes auditable security-relevant events. Management audits are often necessitated by major changes in a business. Some of the events that call for a management audit are top management changes, mergers and acquisitions, and succession planning. Analysis and assessment of competencies and capabilities of a company's management in order to evaluate their effectiveness, especially with regard to the strategic objectives and policies of the business.

SM Information Managementis the ability to gather and manage security-relevant information of Cloud services (such as region, time, etc.) that are not a native component of the security management areas. A software that automates the collection of event log data from security devices, such as firewalls, proxy servers, intrusion-detection systems, and antivirus software. The SIM (Security Information Management) translates the logged data into correlated and simplified formats

VI. ARCHITECTURE OF NETWORK SECURITY USING DES, AES AND RSA

Below figure showsthe simplification of this research paper.

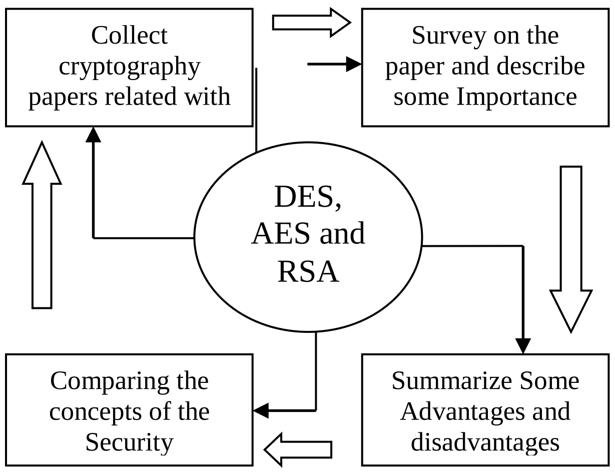

Collect cryptography papers related with
Survey on the paper and describe some Importance
DES, AES and RSA
Comparing the concepts of the Security
Summarize Some Advantages and disadvantages

Fig 5: Architecture of Network Security using DES, AES and RSA

VII. Comparative Analysis Of Des, AES, and RSA

Among those applications they might have some equalize relation and also have some controversy. Some of them are listed below:

Table 1: Comparison between DES, AES and RSA

DES	AES	RSA
DES stands for Data Encryption Standard	AES stands for Advanced Encryption Standard	RSA acronym for Rivest, Adi Shamir
Key length is 56 bits	It uses various lengthsof the key Suchas 128, 192 and 256 bits	>1024 bits
It uses the Symmetric algorithm	It uses the Symmetric algorithm	It uses the Asymmetric algorithm
DES is developed in the year by 1977	AES is developed in the year by 2000	RSA is developed in the year by 1978
DES uses 6 rounds to encrypt and decrypt the text	AES uses 10/12/14 rounds to encrypt and decrypt the text	RSA uses 1 round to encrypt and decrypt the text
Encryption and Decryption process is moderate	Encryption and decryption process is faster	Encryption and Decryption process is slower
Security is not enough	It has excellent security	Low and poor security
Implementation of DES is, when compared with the software,hardware gives the better performance	It will become faster	Not as efficient or competent
Inherent vulnerabilities are Brute-Forced and Cryptanalysis Attack	For this Brute-Force attack only	RSA inherent vulnerabilities are Oracle and Brute-Forced attack

VIII. Conclusion And Future Enhancement

This paper presents a basic literature survey and comparing the work of few selected cryptography concepts and also presents security management areas for cloud computing. The selected crypt theories are DES, AES and RSA. Severaldifferences have been presented in this paper. Through this survey, till now DES performance ispoor and the process is always slow when comparing to another cryptography algorithm and which is having low security. This is the biggest drawback.

In our future work we find some drawback like key problem, file generation error, attacks, etc.and also simply the problem by the use of cryptographyalgorithms to solve thatproblem.

References

[1] Dr.C.Priya et al.,**"Trusted Cloud Computing Platform in IaaS for Closed Box Execution Environment to VM"** in Journal of Advanced Research in Dynamical and Control Systems, volume 10, issue 4, 193-98, 2018, ISSN 1943-023X, SNIP 0.294, UGC Journal No. 26301

[2] R.Ranjani and Dr.C.Priya, **"A Fusion of Image Processing and Neural Networks for Lung Cancer Detection Using SVM In Matlab"** in International Journal of Pure and Applied Mathematics, Volume 119, issue 10, 100-111, ISSN 1311-8080(Print), ISSN 1314-3395(Online) 2018 Impact Factor: 7.19, UGC Journal No. 23425

[3] Prachi V. Bhalerao, et al., "Hardware Implementation of Cryptosystem by AES Algorithm Using FPGA", in IJCMC, Vol. 6, issue 5, pp. 84-89, May 2017.

[4] GurupinderKaur, Dr.Amandeep Singh Sappal. "Implementation of AES Algorithm on FPGA For Low Area Consumption", in International Journal of Advanced Research in Computer Science, Volume 8, No.7, pp. 704-707, July-August 2017.

[5] R.Ranjani and Dr.C.Priya, **"A Survey on Face Recognition Techniques: A Review"** in International Journal of Pure and Applied Mathematics, volume 118, issue 5, 253-74, ISSN 1311-8080(Print), ISSN 1314-3395(Online) 2017 Impact Factor: 7.19, UGC Journal No. 23425

[6] Dr.C.Priya,**"TaaS: Trust Management Model for Cloud-Based on QoS"** in Journal of Advanced Research in Dynamical and Control Systems, volume 9, issue 18, 1336-45, 2017, ISSN 1943-023X, SNIP 0.294, UGC Journal No. 26301

[7] Parson Raghav, et al., "Securing Data in Cloud Using AES Algorithm", in International Journal of Engineering Science and Computing (IJESC), Volume 6, Issue No.4, pp.3672-3675, April 2016

[8] Afolabi, A.O. &Atanda, O.G., "Comparative Analysis of Some Selected Cryptographic Algorithms", in Computing, Information Systems, Development Informatics & Allied Research Journal, Vol. 7, No.2, June-2016.

[9] C.Priya and Dr.R.Latha,**"TaaS: A Framework for Trust Management in Cloud Computing Environments"** in International Journal of Science and Research, volume 5, issue 9, 1402-05, ISSN 2319-7064(Online), DOI: 10.21275/ART20161879, September 2016.

[10] ShrutiSrivastava, A.Y.Kazi, "Design of AES on FPGA Hardware", in International Journal of Industrial Electronics and Electronical Engineering, Volume-4, Issue 9, pp. 152-154, Sep 2016.

[11] Ch.J.L.Padmaja, et al., "RSA Encryption Using Three Mersenine Primes", in Int.J.ChemSci, pp. 2273-2278, 2016.

[12] Ashraf Odeh, et al., "A Performance Evaluation Of Common Encryption Techniques With Secure Watermark System (Sws), in International Journal of Network Security & Its Applications (IJNSA) Vol.7, No.3, pp. 32-38, May 2015.

[13] IsratJahan, et al., " Improved RSA cryptosystem based on the study of number theory and public key cryptosystem, in American Journal of Engineering Research, pp. 143-149, 2015.

[14] C.Priya and Dr.N.Prabakaran, **"A Research on Trusted Computing with Secure Resources for Multiple Clouds using Data Coloring"** in International Journal of Innovative Research in Computer and Communication Engineering, volume 3, issue 4, 3654-62, ISSN 2320-9801(Online),ISSN 2320-9798(Print), April 2015. Impact Factor: 4.447

[15] San San Tint, "Survey On Asymmetric Algorithm Using RSA Different Modified Models", in Computer Applications: An International Journal (CAIJ), Vol.1, pp.27-35, November 2014.

[16] Dr.PrernaMaharajan&AbhishekSachdeva, "A Study Of Encryption Algorithms AES, DES and RSA for Security", in Global Journal of Computer Science and Technology, Volume 13, Issue 15, 2013.

[17] C.Priya and Dr.N.Prabakaran**"A Research on Security prospects for Adopting Multiple Clouds through Cloud Computing"** in International Journal of Engineering Development and Research, volume 1, issue 2, 37-43,ISSN 2321-9939(Online), Sep-Oct 2013. Impact Factor: 1.79

[18] C.Priya and Dr.N.Prabakaran**"Security Management in Inter-Cloud"** in International Journal of Emerging Trends and Technology in Computer Science, volume 1, issue 3, 233-235,ISSN 2278-6856(Online), Sep-Oct 2012. Impact Factor: 4.413

[19] C.Priyaetal.,**"A Trust, Privacy and Security Infrastructure for the Inter-Cloud"** in International Journal of Computer Technology and Applications, volume 3, issue 2, 691-695, 2012, 2229-6093, March 2012.

[20] C.Priya et al., **"The Next Generation of Cloud Computing on Information Technology"** in International Journal of Computer Science and Information Technologies, Volume 2, No 5, 2011, ISSN 0975-9646.

[21] C.Priya et al., "**Monitoring System using Smart Phones**" in International Journal of Computer Engineering and Technology (IJCET), Jan 2011Vol. 1(3), 2011, 0976-6375.

[22] M.Kannan and Dr.C.Priya, **"A survey on fault detection enabled Optimal Load Balancing Technique by efficient utilization of VM in Cloud Computing"** International Conference on Computing Sciences (ICCS), ISBN 978-81-910217-0-9, pp.84

Information Technology Journal 5 (3): 416-421, 2006
ISSN 1812-5638
© 2006 Asian Network for Scientific Information

Hardware Implementation of Instruction Level Parallel Architecture Incorporating Special Functional Units for Image Processing Algorithms

M. Kannan and S.K. Srivatsa
Department of Electronics Engineering, MIT Campus, Anna University,
Chromepet, Chennai-44, Tamil Nadu, India

Abstract: Parallel processing is an efficient form of information processing with emphasis on the exploitation concurrent events in computations. Considering a sequence of assembly instructions for a specific problem it is found that many of the consecutive instructions are independent of each other, without any data dependencies between them. This work exploits such situations and it executes pairs of instructions, which do not have dependencies between them, on two different processing elements, thus enhancing the speed of operations. It is not always true that any two instructions taken from a sequence of instructions could go in parallel. The various types of dependencies that exist among the instructions are the bottleneck in executing instruction in parallel. The various possible data dependencies and control transfers are handled so that most of the instructions are run pairs. The ILP(Instruction Level Parallelism) architecture designed here is to be used for image processing applications. Since specific hardware solutions are always faster that their software counterparts and we have dedicated hardware units for most frequently used image processing problems of finding DFT and DCT. The proposed architecture improves the performance with a speed up factor of more than 1.5 with lesser data dependencies, we can get a higher speed up factor, upper bounded by the value of 2 by the Amdahl's law.

Key words: ILP architecture, DCT, DFT, parallel architecture

INTRODUCTION

Parallel processing has been one of the hottest ideas of computing in which information processing is done with emphasis on the exploitation of concurrent events in computations. Not only architectures, but also compilers and operating systems have been striving for more than two decades to extract and utilise as much parallelism as possible to improve the speed of the systems (Hwang, 1990; Lilja, 1994).

Basically, parallelism is used in two different contexts, available parallelism and utilised parallelism. The available parallelism is present inherently in the problem solutions thus making the implementations easily parallel. The parallelism could be either functional parallelism, which comes from the logic of the solution, or data parallelism, which is due to the type of data structures used (like vectors and matrices).

There are various levels in which the inherent functional parallelism of the solution to the problem can be utilised to complete the solution in a fast manner. The different levels of parallelism are Program level (Coarse-grained parallelism), Procedure level (Middle-grained parallelism), Loop level (Middle-grained parallelism) and Instruction level (Fine-grained parallelism). Among these levels, the highest level of parallelism possible is in the program level, which requires the device of parallel algorithms. An optimising parallel compiler achieves procedure level parallelism and loop level parallelism. By exploiting concurrency between consecutive instructions, which is the processing in the instruction level, an immense speed up can be got. This fine grained processing gives the maximal through put without making changes to the original algorithm.

The implemented ILP processor has instruction set which is a subset of the DLX instruction set (Patterson and Hennessy, 1990), which supports the special instructions used specifically for image processing applications, DCT and DFT instructions. DLX is a simple load store architecture, which had been designed for pipelining efficiency. In order to achieve higher speed up, these frequently used instructions are given to dedicated hardware modules. The performance of the system improves very much when these dedicated hardware units are added.

Corresponding Author: M. Kannan, Department of Electronics Engineering, MIT Campus, Anna University, Chromepet, Chennai-44, Tamil Nadu, India Tel: +91-44-22237276/233

This work proposes the design of a super scalar, instruction level parallel architecture. There are two processing elements and dedicated hardware for DCT and DFT (Schlansker *et al*., 1997, Buehrer and Ekanadham, 1987). The memory is divided into Data memory and program memory thus following the Harvard Architecture of Memory Organisation. For each clock tick, two instructions are fetched from the Program memory and a data dependency check is done on them (Sohi, 1990). If they are independent and both their data are ready, then they are issued to two different processing elements and hence these two instructions run in parallel. The dedicated units can also run in parallel with these two processing elements.

IMPLEMENTATION ILP ARCHITECTURE

The instruction level parallel architecture designed here is a Super Scalar processor (IEEE, 1998 a,b). Two instructions are issued per clock since there are two pipelines in the design. Parallelism between consecutive instructions in the program sequence is exploited in this design. The overall architecture of the processor is shown in Fig. 1. It consists an instruction pre fetch unit that fetches instructions from an interleaved instruction memory, a scheduler which examines the instructions (Arvind and Nikhil, 1986; Pratt, 1976), resolves data dependencies among them and prepares them for execution, a tagged data memory and tagged register file with tags associated with each word to indicate the presence or otherwise of valid data , a set of data structures used by the scheduler, namely the Ready Queue, the Deferred Instruction Queue and the Tag Unit, a set of Processing Elements (PEs) to execute the instructions for which data are available. These PEs can be homogeneous or each could be a specialized functional unit. This design implements the lower order interleaved memory organisation. The memory modules are byte addressable. The processor is 32 bit processor with a register file of 32 registers, supports minimal number of addressing modes to make the design as simple as possible. The addressing modes supported are immediate addressing, Register addressing and Based Indexed addressing. The memory is organized as interleaved memory.

To facilitate simplicity again, DLX basically supports only 3 formats of instructions. They are I Type instructions, R Type instructions and J Type instructions and their formats are given in Fig. 2.

DLX supports the list of simple operations that is supported by almost all the processors. The instructions may be broadly classified as Load Store, ALU, Branches

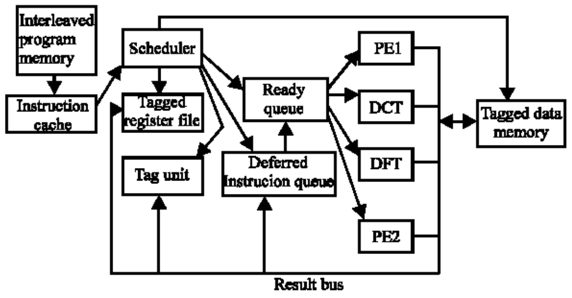


Fig. 1: Overall instruction level parallel architecture

I Type:

Opcode (6)	SRC 1(5)	SRC 2(5)	Immediat (16)

R Type:

Opcode (6)	SRC 1(5)	SRC 2(5)	DST (5)	Function (11)

J Type:

Opcode (6)	Offset (26)

Fig. 2: DLX instruction formats

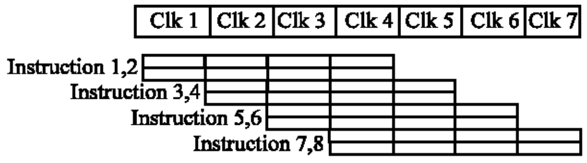


Fig. 3: Pipelined instruction execution in the ILPA

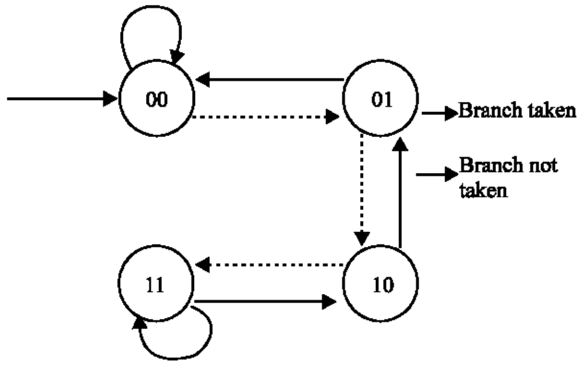


Fig. 4: Branch prediction state diagram

and Jump instructions. For the ILPA, there is one more type of instruction possible, which is the special instruction type for the dedicated hardware units. The opcodes are given in Table 1.

In a program, instructions sometimes depend on each other in such a way that a particular instruction cannot be executed until a preceding instruction or even

Table.1: Opcode of DLX processor (Main Opcodes)

	00	01	02	03	04	05	06	07
00	Special		J	JAL	BEQZ	BNEZ		
08	ADDI		SUB					
10			JR	JALR	ANDI	ORI	XORI	NOP
18	SEQI	SNEI	SLTI	SGTI	SLEI	SGEI		
20	LB	LH	LW	LF	LD	LBU	LHU	LHI
28	SB	SH	SW	SF	SD	SLLI	SRLI	SRAI
Special opcodes (Main Opcode = 00)								
00	01	02	03	04	05	06	07	
00	ADD		SUB		AND	OR	XOR	
08	SEQ	SNE	SLT	SGT	SLE	SGE		
18	SLL	SRL	SRA		DFT	DCT		

two or three preceding instructions have been executed. There are three types of dependencies possible. They are Data dependencies (subsequent instructions are dependent on each other because of data), control dependencies (a conditional jump statement is executed) and resource dependencies (the same resource is being used by both instructions). All these dependencies are taken care appropriately in the scheduler design (Sohi, 1990; Arvind and Iannuei, 1986; Buehrer and Ekandham, 1987).

The ILP architecture is a pipelined architecture like DLX. This design implements a four-stage pipeline, which is different from the standard DLX architecture. The memory access stage in the standard DLX design is removed in this design, since it is considered redundant. The four stages of the pipeline are fetch stage, decode stage, execute stage and write back stage.

As shown in the Fig. 3, initially, in the first clock, the first two instructions are fetched. In the next clock, they are decoded and the fetch unit fetches the next two instructions. In the next clock, the first two instructions are executed in the corresponding execution units, the instructions 3 and 4 are decoded and instructions 5 and 6 are fetched. In the fourth clock, the results of instructions 1 and 2 are written back, instruction 3 and 4 are executed, instruction 5 and 6 are decoded and the instruction 7 and 8 are fetched.

The ILP architecture design implements the branch prediction logic as given below in Fig. 4. Whenever a conditional branch instruction is encountered, it initially assumes that the branch would be taken and starts issuing the instruction from the 'target address' sequence of the program to the instruction pipeline, rather than the 'sequential address' of the program. There are two bits associated with each and every branch instruction that occurs in a program out of which the first bit says if the branch would be taken or not. '0' indicates the branch would be taken and '1' indicates it would not be taken.

When the execution of the branch instruction is over and if the branch is taken as predicted, then the bits

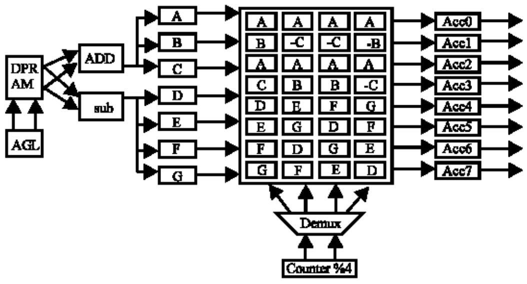


Fig. 5: Architecture of the DCT processor

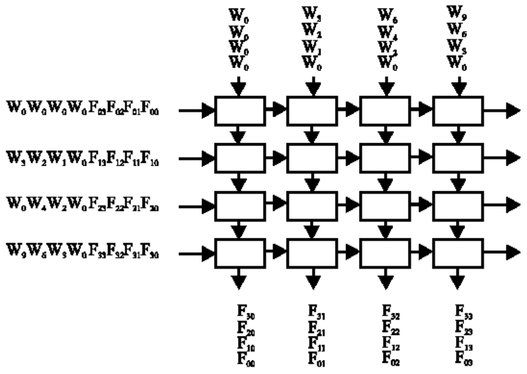


Fig. 6: Systolic array representation of DFT

remain at "00". If the branch is not taken, but was predicted that it would be taken, then the bit transition occurs. So, the current status of the branch is used as history to predict the future branches.

Consider the following example, which adds r5 6 times into r1. In the first clock, 1 and 2 are fetched. In clock 2, they are decoded and 3 and 4 are fetched. In the third clock, 1 and 2 are executed, 3 and 4 are decoded and in the mean time, the fetch unit is free which now uses the branch predicted 'target address' and fetches the instructions 1 and 2 again.

- Loop: Add r1, r1, r5
- Addi r6, r6, 1
- Slei r4, r6, 6
- Bnez r4, loop

The sequence proceeds in the above said manner for 6 times. When the instruction 3 resets the register r4 when r4 becomes greater than 6, the branch would not be taken, which would have been predicted to be taken. In that case, instructions 1 and 2 would have gone into the pipeline already. So, we have to flush the pipeline this time and the status for this branches moves from "00" to "01".

When the different units of ILP Architecture are considered, the Scheduler takes care of scheduling instructions onto available processing elements by coordinating with other units. This includes preparing the instructions and putting in the Ready Queue (RQ) when the data is available. If data is not available the instruction is transferred to Deferred Instruction Queue (DIQ) and transferred from DIQ to RQ once the data is available.

The Tag unit acts as a reservation station for the register file. Whenever a write to a register is initiated, an instance for that register is made in the tag unit. Thereafter, instead of addressing the register by its number, it is addressed by its tag number or the instance number. When one or more of the input operands are not ready at the time of decoding, then that particular instruction has to be deferred. This data structure DIQ is used to store the deferred instructions. The architecture has an extensive set of registers organised as a register file with 32 GPR s. Each Register in the register file has a busy bit associated with it. If the busy bit is set, this indicates that a write into the register has been initiated and an instance created in the tag unit. If it is not set, then the data in the register is valid and that data could be read by the instructions.

The Ready Queue maintains the list of all instructions, which are currently ready to get executed. The Ready Queue gives the instructions to be executed to the scheduler and the scheduler schedules the instructions to the processing elements when it finds the PEs free. On the rise of every clock, the scheduler takes a look at the status of the execution units. If one or more of them are free, then the instructions from the RQ are scheduled to the execution units and the locations in the RQ are made free for further instructions to come.

Related to the Data Dependency Check if the busy bit is set for particular register, the tag unit is searched for the latest tag entry of that register (Espara and Valero, 1997). The corresponding tag index is entered as tag number of the source operand to the DIQ along with the instruction. If the busy bit was reset for the corresponding source register, the data is retrieved from the register file and stored in the corresponding slot of the DIQ. This is repeated for both the source operands. That is, if for one of the sources, the corresponding register has valid data item, this data is stored along with the instruction in the DIQ until the other register gets that data. If both s1 and s2 are available, the instruction is scheduled to any available computational unit and executed. In case of the destination register, a free tag is allocated to this register indicating that an instance has been created for generating this value.

Processing element: The core of any architecture is its processing element. The ILPA processes 32 bit operands. The processing element of the ILP architecture consists of Carry Look Ahead adder, Carry Save Multiplier, Logic units, Comparator, Barrel Shifter etc. (Patterson and Hennessy, 1990). The other two functional units are the DCT and DFT (Pratt, 1976).

Discrete cosine transform: The Discrete cosine transform is a sequence of multiply-accumulate operations as given in the equation. The 2D DCT is a separable function and can be easily achieved by two 1D DCTs operating in a pipelined manner. The fast algorithm for the 8 point DCT is discussed below. The one-dimensional DCT is expressed mathematically as follows.

$$\mathrm{Y(k)} = \sqrt{\frac{2}{\mathrm{N}}}\mathrm{C(k)}\sum_{i=0}^{N-1}\mathrm{x(i)}\cos\frac{(2i+1)k\pi}{2N}$$

where:

$$\mathrm{C(k)} = \frac{1}{\sqrt{2}} \quad \text{when k=0}$$
$$= 0 \text{ Otherwise}$$

For an eight-point DCT, the matrix shown here gives the transform coefficients.

$$\begin{pmatrix} Y(0)\\ Y(1)\\ Y(2)\\ Y(3)\\ Y(4)\\ Y(5)\\ Y(6)\\ Y(7) \end{pmatrix} = 1/2 \begin{pmatrix} A\ A\ A\ A \\ B\ C-C\ B \\ A-A-A\ A \\ C-B\ \mathbf{B}-C \\ D\ E\ F\ G \\ E-G-D-F \\ F-D-G\ E \\ G\ -F\ E\ - \end{pmatrix} \begin{pmatrix} X(0)+x(7)\\ X(1)+x(6)\\ X(2)+x(5)\\ X(3)+x(4)\\ X(0)+x(7)\\ X(1)+x(6)\\ X(2)+x(5)\\ X(3)+x(4) \end{pmatrix}$$

Where,

$$A = \cos\frac{\pi}{4};\ B = \frac{\pi}{8}\cos;\ C = \sin\frac{\pi}{8}$$
$$D = \frac{\pi}{16}\ \cos\frac{3\pi}{16};\ E = \cos;\ F = \sin\frac{3\pi}{16};\ G = \sin\frac{\pi}{16}$$

The block diagram of 1D DCT processing unit is shown in the Fig. 5. This consists of the multiplier unit, the register unit and the pre processing adder and subtractor units, with a post processing accumulator unit.

The input data from the Dual ported RAM is accessed in blocks of 8×8. Two values can be accessed simultaneously from the DPRAM. The first two values accessed are x(0) and x(7) where 0 and 7 indicate the position of the pixel in the current row under consideration. The address generation logic for accessing these values in a specific order as required by the algorithm is given as follows.

The values fetched from the DPRAM are given to the adder and subtractor. The algorithm requires the sum and difference of the two value pairs x(0) and x(7), x(1) and x(6), x(2) and x(5) and x(3) and x(4) for the computation of DCT coefficients Y(0),Y(1),... Y(7) as indicated by the matrix. The adder computes the sum of the two values and the subtractor computes their difference.

The output of the adder should be multiplied with the coefficients A, B and C. Parallel multipliers are used to do this. The algorithm also requires the difference multiplied by the coefficients D, E, F and G, which is also done simultaneously by using 4 multipliers.

Discrete fourier transform: The DFT is also a set of multiply-accumulate operations. The DFT of a two dimensional signal is given as follows:

$$F(u,v) = \sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y) e^{-j2\pi\left(\frac{vy}{N}\right)} e^{-j2\pi\left(\frac{ux}{N}\right)}$$

This mathematical function can be split into two separate MAC operations.

$$F'(x,v) = \sum_{y=0}^{N-1} f(x,y) e^{-j2\pi\left(\frac{vy}{N}\right)}$$

and

$$F(u,v) = \sum_{x=0}^{N-1} F'(x,v) e^{-j2\pi\left(\frac{ux}{N}\right)}$$

The multiplied values are accumulated into a register in the Processing element in the first part of the DFT (finding F' (x, v)) and in the next part, the accumulated values are multiplied and outputs are taken (finding F (u, v)). The DFT block is designed based on the Systolic Array topology.

The PEs are arranged in a 4×4 2D array. When the clock goes high, the process of multiply-accumulate begins. Initially all the registers of all the PEs are cleared. The control input is held low during the first half of the operations. The weights or the twiddle factors for the DFT are fed as vertical inputs whereas the horizontal inputs are the input matrix values.

Table 2: Synthesis results of some units of ILPA

Module	Cell count Combinational	Sequential	Total	Maximum speed
DCT Unit	965	192	1157	22.3 MHz
DFT Unit	124	16	140	34.6 MHz
Adder Unit	215	0	215	84 MHz
Multiplier Unit	1289	0	1289	45.9 MHz
Comparator Unit	177	0	177	29.4 MHz

For the first 4 clocks, the input values are multiplied with their corresponding weights and are stored into the registers Rij. The input value for a row is passed on to all the processors in that row simultaneously and the weight for a column is also broadcasted to all the processors in the column in the same way. When these 4 clocks are counted, the control signal is made high.

RESULTS

The blocks of the ILPA are simulated using Modelsim and the results are obtained. The above blocks are synthesised using Leonardo Spectrum and the results are given in Table 2.

The performance of the ILPA is measured and compared with a single processing systems for standard Matrix Multiplication. The number of instructions executed is 25. Using single processor systems it would have taken 25 clocks to run the same code of matrix multiplication. Since the ILPA takes advantage of parallelism, it could complete the same job with 16 clock cycles, with all data dependencies resolved and producing the same output. Hence a speed up of 1.56 is achieved for this particular program. If we write longer programs and with lesser data dependencies, we can get a higher speed up factor, upper bounded by the value of 2 by the Amdahl's law.

DISCUSSION

The ILP architecture designed here exploits the inherent parallelism in the program and appropriately speeds up the execution. When the design is used for the specific application of image processing, the dedicated units come to help a lot. The architecture proposed here can also be used for any other application not necessarily, Image processing, since all it looks for is the independence of consecutive instructions. The architecture can improve the performance with a speed up factor of maximum 2. For almost all programs, it easily exceeds 1.5.

As mentioned earlier, this study is easily extensible to any specific field. For example, if it is going to be used for a data base specific applications, the same factors of speed up could be gained easily. Since the data base applications will hardly be using the DCT and DFT units, they can be replaced with the functions these applications often use, like sorting and searching. The units can also be replaced by dynamically reconfigurable units, which can do specific hardware functions, giving speed and also be reconFigd, thus giving programmability. Thus reconfigurable hardware units give us a very good hardware-software balance.

REFERENCES

Arvind, R. and S. Nikhil, 1986. Executing a program on the MIT Tagged Dataflow Architecture. Memo 302, MIT Lab, Comput. Sci., Cambridge MA.

Arvind, R. and R.A. Iannuci, 1986. Two fundamentals Issues in Multiprocessing". Memo 226-6, Mit Lab, Computer Sci., Cambridge, MA, pp: 1-21.

Buehrer, R. and K. Ekanadham, 1987. Incorporating data flow ideas into Von Neumann processor for Parallel execution: IEEE Transaction on Computers, 36: 1515-1521.

Espasa, R. and M. Valero, 1997. Exploiting Instruction and Data level parallelism. IEEE Micro, Vol. 17.

Hwang, K., 1993. Advanced Computer Architecture. McGraw-Hill.

IEEE, 1998. Signal Processing Magazine, Special Issue on Multimedia Processors, Vol.15, No. 2.

IEEE, 1998. Symposium on FPGAs for Custom Computing Machines, Napa Valley, California.

Lilja, B.J., 1994. Exploiting the parallelism available in loops. IEEE Computer, 27: 13-26.

Patterson D.A. and J.I. Hennessy, 1990. Computer Architecture A Quantative Approach. Morgan Kaufmann Publishers, Inc San Mateo, California.

Pratt, W., 1991. Digital Image Processing. Springer-Verlag.

Schlansker, M. and T.M. Conte *et al.*, 1997. Compilers for instruction level parallelism. IEEE Computer.

Sohi, G.S., 1990. Instruction issue logic to high performance, interruptible, multiple functional unit, pipelined computers. IEEE Transactions on Computers, 39: 349-358.