Natl. Acad. Sci. Lett.
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

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

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

✉ I. Hariharan
hariharan166@gmail.com

M. Kannan
mkannan@annauniv.edu

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

Published online: 06 April 2019

(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.

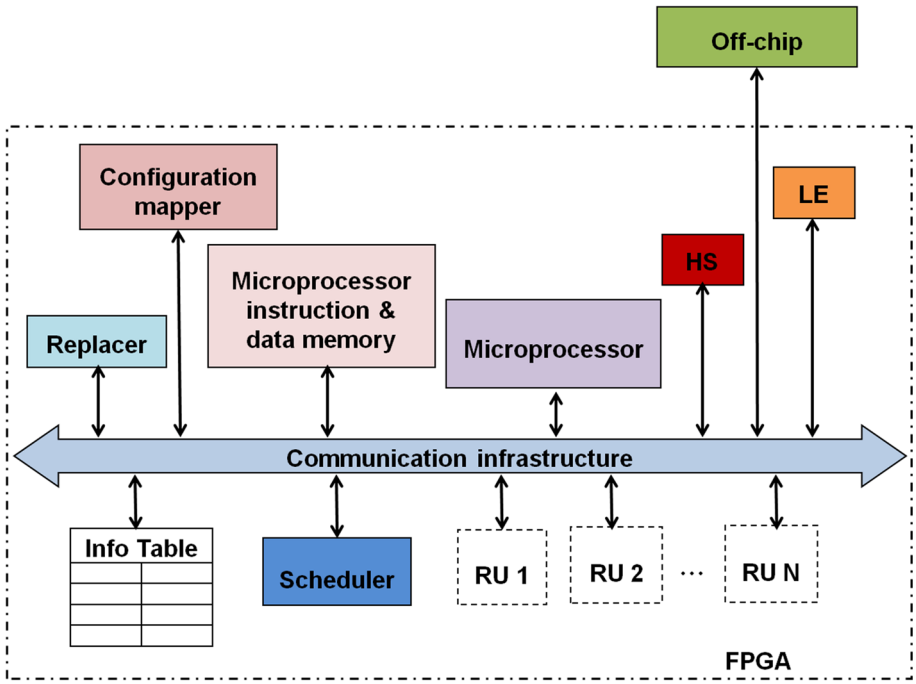

Fig. 1 Proposed architecture

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

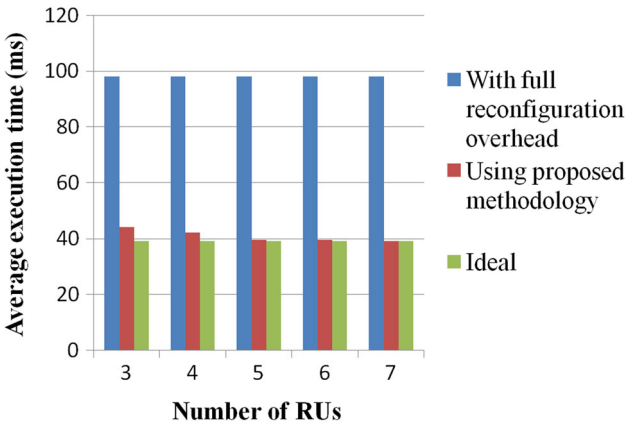

Fig. 2 Performance evaluation of the proposed architecture

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

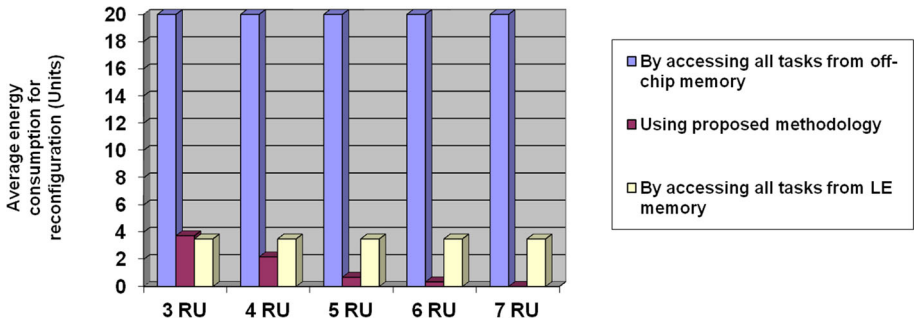

Fig. 3 Reduction in the energy reconfiguration overhead using our proposed architecture

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