Reducing reconfiguration overheads of a reconfigurable dynamic system using active run-time prediction

Hariharan. I [1], Kannan. M [2]
Department of Electronics Engineering
MIT Campus, Anna University
Chennai, Tamil Nadu, India
[1] hariharan166@gmail.com, [2] mkannan@annauniv.edu

***Abstract*—The performance of the Field-Programmable Gate Array (FPGA) is largely affected by its reconfiguration overheads. For a dynamic system in which the nature of the system is unpredictable at design-time, these overheads are expensive in terms of performance. To reduce these overheads, architecture along with two algorithms is proposed which dynamically predicts and schedules the configuration. As a result, the time reconfiguration overhead is reduced to improve the performance. In most cases, the scheduling result obtained exactly matches with the system whose configurations are fetched from HS memory. This could be the near maximum achievable performance for any FPGA architecture, executing a dynamic application.**

Keywords—Dynamic systems; Prediction based algorithm; Reconfiguration overheads; Configuration mapping; field-programmable gate array (FPGA)

I. INTRODUCTION

Reconfigurable architecture finds its place in most of the application that was previously ruled by the other general purpose processor (GPP) and digital signal processor (DSP) systems. It is because of their high flexibility without compromising much on performance [1], [2]. Some of the features like partial reconfiguration [3] and run-time reconfiguration [4], [5] make field-programmable gate array (FPGA) unique when compared with other processor systems. When the configurations are loaded onto the available resources on an FPGA, reconfiguration overheads are generated [6]. These overheads can degrade the system performance to a larger extent. To reduce these overheads, proper management of configurations is very important.

In our paper, we focus mainly on a dynamic system. A dynamic system may execute more than one application at a time. Its main drawback is that the future tasks are unpredictable in nature. Without predicting the future tasks, it is very difficult to manage the configurations needed for partial reconfiguration. Improper management of configuration results in configuration thrashing problem. Therefore, in this paper, we present a prediction based scheduling where most of the reconfiguration overheads are reduced and the system achieves a near maximum performance.

II. RELATED WORKS

Reconfiguration overhead occurs when the configurations are improperly managed. During partial-reconfiguration, configuration data are fetched from the off-chip memory and loaded into the desired Reconfigurable Unit (RU). This fetching of configurations from the off-chip memory is costly as it carries a significant amount of both the time and energy reconfiguration overheads. In paper [7], a configuration memory hierarchy is proposed along with two configuration mapping algorithms. One algorithm is for dynamic systems. The authors tried to keep only a limited number of tasks inside the on-chip memories, to avoid configuration thrashing problems. Authors from the paper [8] and [9], used prefetch approach to reduce the reconfiguration overhead. Prefetching the configurations in advance hides most of the time reconfiguration overheads. It improves the system performance. The size of the configuration is directly proportional to the amount of reconfiguration overhead generated. So, authors in paper [10] used intra-bitstream compression technique where they exploit the redundancies available between successive configurations and reuse them. The regularities between different configurations are analyzed and the configurations are compressed to reduce the reconfiguration overhead in the paper [11]. Paper [12] used compression, prefetching, caching, and allocation services to reduce the reconfiguration overheads in heterogeneous multicore RSoC systems.

Authors of the paper [13], proposed a fast reconfiguration manager named FaRM which uses bitstream compression, direct memory access, and scheduling techniques to reduce the reconfiguration overheads. Multi-context FPGAs are used to hide the time reconfiguration overhead [14]. When the active Virtual ConFiguration (VCF) is working in the foreground, a configuration is being loaded in the background. Configuration context swapping takes very short time compared to the execution time. Multiple configuration controller concepts are introduced in [15]. Previous works considered that task can be executed in parallel, but the configurations are loaded only in sequence. Here, the configurations are loaded in parallel with the help of having multiple tiles and each tile carrying its own configuration SRAM. It reduces the configuration overheads by 21% compared with a system having a single configuration controller. All the above techniques can be used efficiently in a static system. But, for a dynamic system, these techniques fails. Because the future is unpredictable. Our paper tries to address this lacuna and reduces most of the reconfiguration overheads generated in any dynamic systems.

III. Proposed Architecture

The proposed architecture is having a memory hierarchy which consists of high speed (HS), low energy (LE) and off-chip memories shown in figure 1. The task graphs to be configured at run-time are present inside the off-chip memory. Mapping analyzer, future task predictor, and replacer together work on the current scenario at run-time to give an efficient mapping on HS and LE memories. Since the system is a dynamic system, future tasks are predicted at every instant and assigned to either of the on-chip memories. RUs in the architecture are blocks which are reconfigured with the tasks as per the requirement of the system dynamically. For every new task arrival, the future task predictor forecast the next task which may get reconfigured. This task is assigned to HS or LE memory based on the decision taken by mapping analyzer. In some cases, as the number of tasks presents inside the HS or LE memory is already full and still, a new task has to be accommodated, then the replacer has to check with the previously stored tasks and replace the non-vital task with the current one. All the details about reconfiguration and execution are updated instantaneously in the info table. The scheduler organizes the reconfiguration and execution of individual tasks in the available RUs. The microprocessor controls every operation which is performed by the proposed architecture. Communication infrastructure can be made using buses or network-on-chip (NOC) [16]. This kind of architecture can be realized using last generation FPGA's of Xilinx [17], [18] and Altera [19], [20].

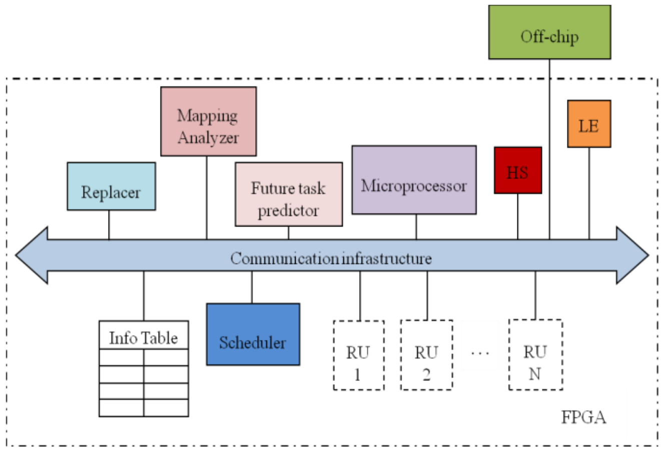

Fig. 1. Proposed architecture

The architecture consists of a number of homogeneous reconfigurable tiles as it appears in the paper [21]. At run-time, these tiles are loaded with the configuration and perform an execution. Dynamic reconfiguration changes the functionality of the individual RU. When the functionality changes, it is necessary to alter the interconnections available between them. This involves a tedious and time-consuming process of placement and routing. Hence, the interconnection network model is used where each tile is surrounded by fixed interconnections that avoid run-time placement and routing [22]. In this architecture, the applications are modeled using the Task Concurrency Management (TCM) [23]. TCM uses two levels of hierarchies. The top level consists of applications represented as task graphs and the lower level consists of tasks (capable of performing individual functions). The dynamic nature of the model is restricted only to the top level.

IV. Motivational Example

A static system can be scheduled efficiently by reducing most of the reconfiguration overheads. It is achieved, because, the future tasks are easily predictable. For a dynamic system, it is not possible to decide the nature of the task graph execution at the design-time phase. But, during the run-time phase, the future task can be dynamically predicted. When the system dynamically starts predicting, the performance of the dynamic system can be improved. Consider a dynamic system, consisting of only two task graphs, namely MPEG-1 and JPEG [7] is shown in figure 2. Each task is represented with specific notations and is given outside the task graph. Whereas, the ideal execution time of individual tasks are specified inside. The nature in which these two task graphs get executed at run-time is shown in figure 3.

At design-time, the system is unaware of the nature of task graph execution. So, it is impossible to predict the task graph execution at design-time. Without prediction, if we allocate some of the tasks to on-chip memories, it may not be the optimum one regarding performance. Hence, the scheduler result is shown for all the tasks assigned to off-chip memory in figure 4. Prefetching of configuration is not possible as the future tasks are unpredictable. Consider, if the nature of the task graph execution is already known and we have only HS memory with ten configurations of memory space. Now, the same scheduler output is shown in figure 5. In figure 5, the scheduler achieves the execution in just 122 ms. Figure 4 and 5 gives the scheduling for only 5 RUs and it is represented as R1 to R5. This is just half the total execution time achieved when all the tasks were kept inside the off-chip memory. The methodology proposed in this paper focuses mainly on run-time prediction and suitable memory allocation (either HS or LE) for every task. This is carried out dynamically at every instant of the application so that the time reconfiguration overhead generated is less.

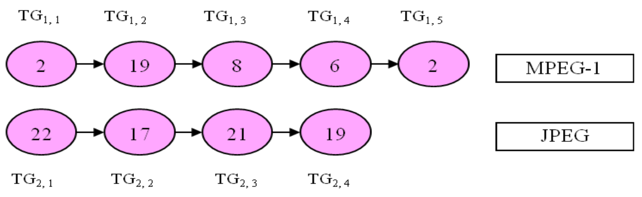

Fig. 2. Task Graphs of MPEG-1 and JPEG

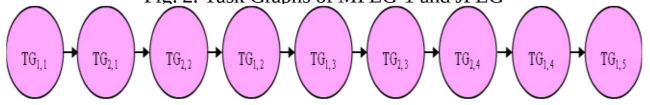

Fig. 3. Execution of MPEG-1 and JPEG at run-time

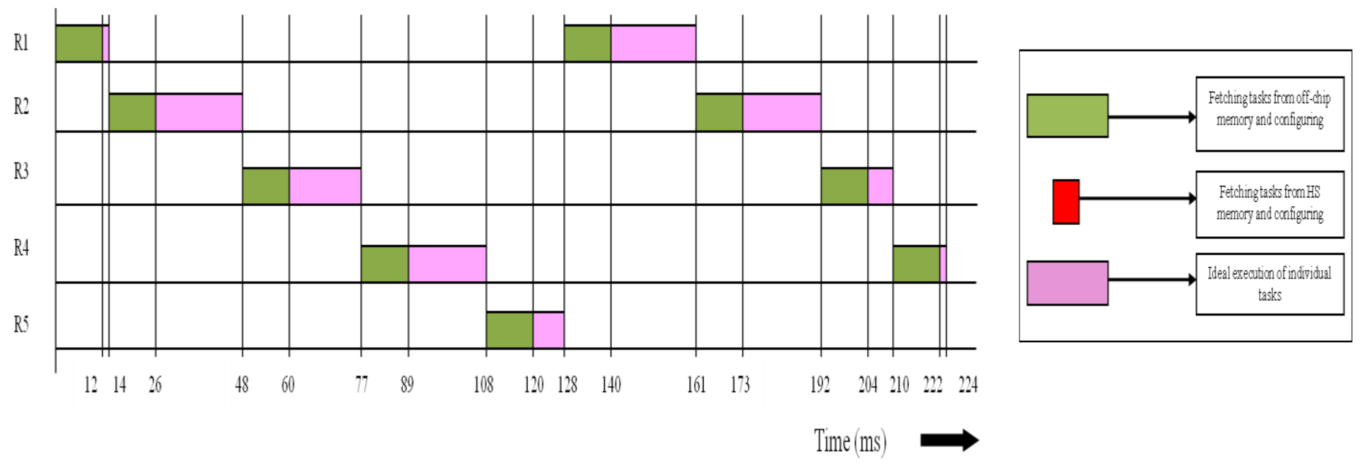

Fig. 4. Tasks scheduled only from off-chip memory

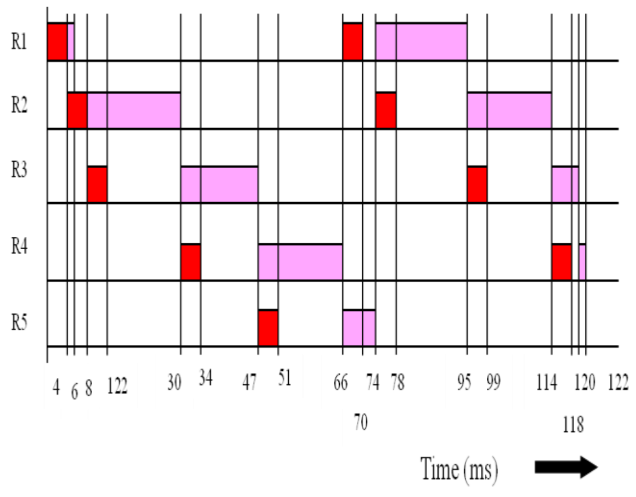

Fig. 5. Tasks scheduled only from HS memory

V. Proposed Methodology

Our methodology is divided into design-time and run-time phases. The complex parallel task graphs are converted into their simpler forms at design-time and stored in the off-chip memory. The run-time phase performs a prediction based scheduling using future task predictor, mapping analyzer, and replacer. The prediction is also dynamic in order to match with the dynamic nature of the application.

A. Design-time phase

A complex task graph is converted into a simple one for easier processing at run-time. This conversion is performed by assigning weights to individual tasks. The most weighted task is kept at the top and the least weighted task is kept at the bottom, while the remaining tasks are distributed between first and the last task as per their weight. Also, the levels of the task graphs are maintained for any task execution. Until the execution of all the tasks present in any level is finished, the execution of next level tasks is not performed. The weight of any task is the sum of their ideal execution time and the maximum weight of their successive nodes. For a leaf node, as there is no successive node, the weight is equal to its ideal execution time.

Consider the task graph given in figure 6. This task graph can be converted into a simpler one as shown in figure 7. The converted task graph is stored in the off-chip memory. This is repeated for all the task graphs.

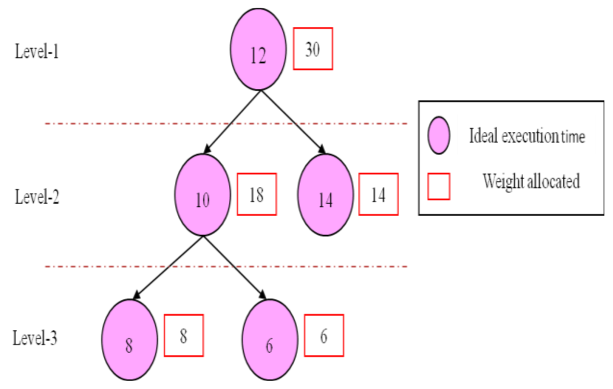

Fig. 6. Complex task graph with weight allocated to individual tasks

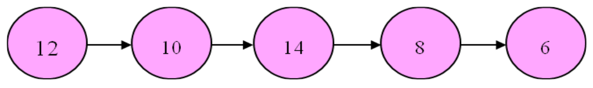

Fig. 7. Converted simpe task graph

B. Run-time phase

The main aim of our methodology is to reduce the reconfiguration overheads of a dynamic system at run-time. The dynamic nature of the system is predicted (next task) for every task execution using the Future Task Predictor (FuTP). The availability of the next task inside the on-chip memory is verified with the help of info table. If the next task is already available inside the on-chip memory, it is loaded (using prefetch) and executed after the current task's execution. If the next task is missing, then it is checked with Mapping Analyzer (MP) to store in either of the on-chip memories. Once it is stored in the on-chip memory, the scheduler performs the loading of the configuration on the available RU and its execution.

Two scenarios are considered in a dynamic system. In one scenario, the number of task graphs is lesser than the total size of on-chip memories (including both HS and LE) and in another, the total number of task graphs is greater than or equal to that of the available total number of on-chip memories. In the latter case, the algorithm-1 used in the first case is used with slight modification to avoid configuration thrashing problems. The modification is followed till the number of task graphs becomes lesser than the total number of on-chip memories available. When it reaches the above condition, the same algorithm used for scenario 1 is used.

1) *Algorithm-1*

This algorithm tries to reduce most of the reconfiguration overheads. It consists of three steps. The main aim is to predict the next task when the current task is being executed and storing the next task inside the on-chip memory based on the hiding value.

a) Step-1

Assign all the first tasks of every task graph to on-chip memories. This is done in an ascending order, starting from HS to LE. When the size of HS equals the total size of HS tasks, the remaining tasks are kept in LE memory. These information is stored in the info table and it will be updated regularly.

b) Step-2

From the available tasks inside the on-chip memory, one task ($TG_{x,\ f}$) is selected by the processor as per the user interrupt and it is loaded in the first available RU. The loaded task will be executed by the scheduler. In $TG_{x,f}$, 'x' represents the task graph number and 'f' represents the task number that is present in the task graph 'x'. The task reconfiguration and execution of the first task is simulated using (1) and (2). For every next task memory assignment, the current task updating is required.

$$R_1 = RTG_{x,f} = HS \text{ (or) } LE \tag{1}$$

$$E_1 = ETG_{x,f} = ietg_{x,f} + R_1 \tag{2}$$

Where R_1 is the simulated reconfiguration time of the first task

E_1 is the simulated execution time of the first task

$RTG_{x,f}$ is the simulated reconfiguration time of the task '$TG_{x,f}$'

$ETG_{x,\ f}$ is the simulated execution time of the task '$TG_{x,f}$'

$ietg_{x,f}$ is the ideal execution time of the task '$TG_{x,f}$'

HS (or) LE is the time needed to load the configuration from high speed or low energy respectively

c) Step-3

While the current task is being loaded (or) executed, the next task must be predicted. For next task prediction, FuTP is used. Keeping the current task as a reference, the FuTP updates the task set with all the possible tasks that may be needed for the application execution. Consider the example given in figure 2 where both MPEG-1 and JPEG are executed simultaneously. If task $TG_{1,\ 1}$ is selected for the execution, then FuTP will have {$TG_{1,\ 2}$ and $TG_{2,1}$} as their updated set.

After the future task set prediction, the FuTP and the info table are compared. If all the tasks present inside the FuTP is already available in the info table, then there is no need for any other task to be kept inside the on-chip memory. Otherwise, the task needs to be kept inside the on-chip memory is checked with mapping analyzer. Before checking with the MA, the next task must be allocated to any of the available RU. The RU allocation is followed in the clockwise direction. Mapping analyzer checks the hiding value 'V_n' using the equations (3) (or) (4). Equation (3) is used before the replacement scenario and equation (4) is used after the replacement scenario to calculate the hiding value. Depending upon the value 'V_n', the task is kept either in HS or LE. If 'V_n' is positive, assign it to LE and otherwise, assign it to HS. After assignment, it is updated using equations (5) and (6) before the occurrence of replacement scenario. And using equations (7) and (8), after the occurrence of replacement scenario. This updating is necessary, to calculate the 'V_n' of the next task.

$$V_n = E_{n-1} - (R_{n-1} + LE) \tag{3}$$

$$V_n = E_{n-1} - (\text{Max }\{E_m \text{ (or) } R_{n-1}\} + LE) \tag{4}$$

Where V_n is the hiding value of the n^th task

E_{n-1} is the simulated execution time of the $(n-1)^{th}$ task

R_{n-1} is the simulated reconfiguration time of the $(n-1)^{th}$ task

E_m is the simulated execution time of the m^{th} task

m is the previous task's number executed in the same RU

Hiding value calculates the exact amount by which a task must be kept in either of the on-chip memory for the current situation in a dynamic system. A negative 'V_n' represents that the task to be kept in HS. Otherwise, a performance overhead occurs. A replacement scenario is the one in which all the available RUs are already occupied, and still there is a next task to be assigned in any of the available RUs. During this scenario, Least Recently Used (LRU) policy is used to select the RU.

$$R_n = RTG_{x,f} = R_{n-1} + HS \text{ (or) } LE \tag{5}$$

$$E_n = ETG_{x,f} = \text{Max}\{E_{n-1} \text{ (or) } R_n\} + ietg_{x,f} \tag{6}$$

$$R_n = RTG_{x,f} = \text{Max }\{E_m \text{ (or) } R_{n-1}\} + HS \text{ (or) } LE \tag{7}$$

$$E_n = ETG_{x,f} = \text{Max}\{E_{n-1} \text{ (or) } R_n\} + ietg_{x,f} \tag{8}$$

Where R_n is the simulated reconfiguration time of the nth task

E_n is the simulated execution time of the nth task

After a task execution, the same task is removed from the on-chip memory. If the MA assigns the next task in HS or LE and in the assigned memory there is no space for the new task accommodation, then the replacer selects anyone of the task from the same memory and assigns it to the other available on-chip memory. The step-3 is repeated till all the tasks from every task graph is executed.

2) Algorithm-2

Algorithm-2 is used when the number of task graphs is greater than or equal to the total size of on-chip memories. At first, hiding value is calculated for every task presents in a task graph and is shown in figure 8. For every task graph, lines 1 – 15 is performed to calculate the 'V_n'. In lines 2 – 4, the first task is assumed to occupy HS and the values of 'R_1' and 'E_1' are updated using equations (9) and (10) respectively.

```
# Hiding value calculation
# Condition : Total number of task graphs >= Total size of on-chip memories

1  : for every task graph do
2  :     assign first task to HS;
3  :     n = 1;
4  :     Update using equations (9) and (10);
5  :     for 2 to i
6  :         n = n+1;
7  :         calculate $V_n$ using equation (11);
8  :         Update using equations (12) and (13);
9  :     end
10 :     for (i+1) to k
11 :         n = n+1;
12 :         calculate $V_n$ using equation (14);
13 :         update using equations (15) and (16);
14 :     end
15 : end
```

Fig. 8. Hiding value calculation

In lines 5 – 9 from the second task to 'ith' task, V_n is calculated using equation (11) and is updated using (12) and (13). 'i' is the available number of RUs. After replacement scenario, the same equation (11), (12), and (13) are modified to equations (14), (15), and (16) respectively to find the value 'V_n'. This is repeated from (i+1)th task to kth task in lines 10 – 15, where k is the total number of tasks present inside the task graph.

```
# Keep only highly crucial task inside the on-chip memory

1  : for every task graph arrival do
2  :     if (size of on-chip memory = size of tasks present inside the on-chip memory);
3  :         assign first task to off-chip memory;
4  :     else
5  :         assign first task to on-chip memory;
6  :     end
7  :     for 2 to k
8  :         sort $V_n$;
9  :     end
10 :     if (size of on-chip memory = size of tasks present inside the on-chip memory)
11 :         assign least $V_n$ to off-chip memory;
12 :     else
13 :         assign least $V_n$ to on-chip memory;
14 :     end
15 : end
```

Fig. 9. Algorithm for placing only crucial tasks in on-chip memory

$$R_1 = HS \tag{9}$$

$$E_1 = R_1 + iet_1 \tag{10}$$

$$V_n = E_{n-1} - (R_{n-1} + LE) \tag{11}$$

$$R_n = R_{n-1} + HS \tag{12}$$

$$E_n = \text{Max}\ \{E_{n-1}\ (\text{or})\ R_n\} + iet_n \tag{13}$$

$$V_n = E_{n-1} - (\text{Max}\ \{E_m\ (\text{or})\ R_{n-1}\} + LE) \tag{14}$$

$$R_n = \text{Max}\ \{E_m\ (\text{or})\ R_{n-1}\} + HS \tag{15}$$

$$E_n = \text{Max}\ \{E_{n-1}\ (\text{or})\ R_n\} + iet_n \tag{16}$$

Since the number of task graphs is greater or equal to that of a number of on-chip memories present, it is difficult to shortlist only one task from the predicted set. Even though the prediction is correct, it is not always possible to place all the tasks inside the on-chip memories. Selection of tasks from the predicted set plays a crucial role and if the selection becomes wrong it leads to configuration thrashing. To avoid this, only a lesser number of tasks are allowed to occupy the on-chip memory using our algorithm-2 given in figure 9. For every task graph, assign the first task to on-chip memory (priority is always given to HS) and select the task having least 'V_n' and assign it to on-chip memory or off-chip memory (based on the availability of memory space and priority is given to on-chip memory). As the number of task graphs is greater than or equal to that of the amount of on-chip memories, algorithm-2 is repeated. And, when the number of task graphs becomes lesser than a number of on-chip memories present, the algorithm-1 is followed.

VI. RESULTS AND DISCUSSION

To analyze our results, a simulation environment is created. CACTI tool is used to model HS, LE, and off-chip memories [24] and [25] which is given in table I. The data provided in table I is only relative values, because as the technology varies their absolute values also vary.

TABLE I

ACCESS TIME AND ENERGY CONSUMPTION FOR DIFFERENT MEMORY MODULES

Memory module	Memory access time for each configuration	Normalized energy consumption
HS	4 ms	1
LE	6 ms	0.7
Off-chip	12 ms	4

Three groups of task graphs are created to analyze the proposed methodology. The first group will have JPEG and MPEG-1 task graphs [7] and the second group will have TG-1, TG-2, and, TG-3 task graphs executed simultaneously. These three task graphs are randomly generated. The third group is having both group-1 and group-2 together. Since the system is a dynamic system, multiple task graphs are competing for the resources. The experiments are carried out for a thousand iterations and the average values are found. The size of HS and LE memories are kept at three.

TABLE II

EXECUTION TIME FOR DIFFERENT GROUPS OF TASK GRAPHS

Task graphs	Execution time (ms)			
	HS	LE	Off-chip	Our methodology
Group-1	122	122	128	122
Group-2	84	107	204	84
Group-3	200	210	305	200

Table II and table III provide execution time and energy consumption values when all the tasks are fetched from HS, LE, and Off-chip memories. The last column gives the execution time and energy consumption values for our methodology. From the tables II and III, it is clear that the performance of the dynamic system is exactly matching with the performance of the system in which all its tasks are preloaded from the HS memory. The energy consumption of our methodology is high. This is because our architecture focuses mainly on the performance and it is not concerned with the energy consumption. Group-3 consists of maximum tasks compared to other groups. By keeping the size of HS and LE memories to 3 configurations each, it is possible to accommodate all the first tasks of every TG (from all the three groups) to on-chip memories. Therefore, the performance of the proposed methodology exactly matches with the performance of the system when all its tasks are fetched from HS alone. If more number of task graphs is considered simultaneously, our methodology still provides the highest possible performance that a dynamic system could achieve.

TABLE III

NORMALIZED ENERGY CONSUMPTION FOR DIFFERENT GROUPS OF TASK GRAPHS

Task graphs	Normalized energy consumption			
	HS	LE	Off-chip	Our methodology
Group-1	9	6.3	36	49.37
Group-2	16	11.2	64	88.68
Group-3	25	17.5	100	136.34

VII. CONCLUSION

The dynamic nature of the application is addressed dynamically using our architecture. Two algorithms are proposed in this paper. When both the proposed algorithms are used along with our proposed architecture, the performance obtained is maximum. As the performance of the system gets maximized, energy reconfiguration overheads become very high in this method. The main objective of this paper is improving the performance of the dynamic system which is achieved. Hence, an increase in the energy reconfiguration overhead as a consequence of improving the performance may be neglected.

REFERENCES

[1]. Tessier, Russell, and Wayne Burleson. "Reconfigurable computing for digital signal processing: A survey." *The Journal of VLSI Signal Processing* 28.1 (2001): 7-27.

[2]. Compton, Katherine, and Scott Hauck. "Reconfigurable computing: a survey of systems and software." *ACM Computing Surveys (csuR)* 34.2 (2002): 171-210.

[3]. Koch, Dirk, et al. "Partial reconfiguration on FPGAs in practice—Tools and applications." *ARCS Workshops (ARCS), 2012*. IEEE, 2012.

[4]. Liu, Ming, et al. "Run-time partial reconfiguration speed investigation and architectural design space exploration." *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. IEEE, 2009.

[5]. Shoa, Alireza, and Shahram Shirani. "Run-time reconfigurable systems for digital signal processing applications: A survey." *Journal of VLSI signal processing systems for signal, image and video technology* 39.3 (2005): 213-235.

[6]. Liu, Shaoshan, et al. "Minimizing the runtime partial reconfiguration overheads in reconfigurable systems." *The Journal of Supercomputing* (2012): 1-18.

[7]. Clemente, Juan Antonio, et al. "Configuration mapping algorithms to reduce energy and time reconfiguration overheads in reconfigurable systems." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22.6 (2014): 1248-1261.

[8]. Hauck, Scott. "Configuration prefetch for single context reconfigurable coprocessors." *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. ACM, 1998.

[9]. Li, Zhiyuan, Katherine Compton, and Scott Hauck. "Configuration caching management techniques for reconfigurable computing." *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*. IEEE, 2000.

[10]. Pan, Ju Hwa, Tulika Mitra, and Weng-Fai Wong. "Configuration bitstream compression for dynamically reconfigurable FPGAs." *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*. IEEE, 2004.

[11]. Li, Zhiyuan, and Scott Hauck. "Configuration compression for virtex FPGAs." *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*. IEEE, 2001.

[12]. Chevobbe, Stéphane, and Stéphane Guyetant. "Reducing Reconfiguration Overheads in Heterogeneous Multicore RSoCs with Predictive Configuration Management." *International Journal of Reconfigurable Computing* (2009).

[13]. Duhem, François, Fabrice Muller, and Philippe Lorenzini. "Farm: Fast reconfiguration manager for reducing reconfiguration time overhead on fpga." *International Symposium on Applied Reconfigurable Computing*. Springer Berlin Heidelberg, 2011.

[14]. Liu, Ming, et al. "Reducing FPGA reconfiguration time overhead using virtual configurations." *Proceedings of the International Workshop on Reconfigurable Communication Centric System-on-Chips*. 2010. Liu, Ming, et al. "Reducing FPGA reconfiguration time overhead using virtual configurations." *Proceedings of the International Workshop on Reconfigurable Communication Centric System-on-Chips*. 2010.

[15]. Qu, Yang, Juha-Pekka Soininen, and Jari Nurmi. "A parallel configuration model for reducing the run-time reconfiguration overhead." *Proceedings of the conference on Design, automation and test in Europe: Proceedings*. European Design and Automation Association, 2006.

[16]. L. Benini and G. De Micheli, "Networks on chip: A new paradigm for systems on chip design," in *Proc. DATECE*, Mar. 2002, pp. 418–419.

[17] *7 Series FPGAs Overview, DS180 (v1.11)*, Xilinx, San Jose, CA, USA, 2012.

[18] *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 Extensible Processing Platform, User Guide, UG850 (v1.0)*, Xilinx, San Jose, CA, USA, 2012.

[19] Altera. (2011). *Stratix V Device Datasheet*, San Jose, CA, USA [Online]. Available: http://www.altera.com/literature/hb/stratix-v/stx5_53001.pdf

[20] 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

[21]. Clemente, Juan Antonio, Javier Resano, and Daniel Mozos. "An approach to manage reconfigurations and reduce area cost in hard real-time reconfigurable systems." *ACM Transactions on Embedded Computing Systems (TECS)* 13.4 (2014): 90.

[22]. T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins, "Interconnection Network enable Fine-Grain Dynamic Multi-Tasking on FPGAs", *Proceedings of the Field Programmable Logic Conference (FPL)*, pp.795-805, 2002

[23]. Chun Wong, Paul Marchal, and Peng Yang, "Task Concurrency Management methodology to schedule the MPEG4 IM1 player on a highly parallel processor platform", *Proceedings of the Ninth International Symposium on Hardware/Software Codesign, CODES 2001*

[24]. Muralimanohar, Naveen, Rajeev Balasubramonian, and Norman P. Jouppi. "CACTI 6.0: A tool to understand large caches." *University of Utah and Hewlett Packard Laboratories, Tech. Rep* (2009).

[25]. Thoziyoor, Shyamkumar, et al. "A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies." *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*. IEEE, 2008.