Exploiting FPGA-Aware Merging of Custom Instructions for Runtime Reconfiguration

Siew-Kei Lam*, Thambipillai Srikanthan
Centre for High Performance Embedded Systems,
Nanyang Technological University,
SINGAPORE.
Email: assklam@ntu.edu.sg*

Christopher T. Clarke
Department of Electronic and Electrical Engineering,
University of Bath,
Bath, United Kingdom.

Abstract— **Runtime reconfiguration is a promising solution for reducing hardware cost in embedded systems, without compromising on performance. We present a framework that aims to increase the advantages of runtime reconfiguration on reconfigurable processors that support full or partial runtime reconfiguration. The proposed framework incorporates a hierarchical loop partitioning strategy that leverages FPGA-aware merging of custom instructions to: 1) maximize the reconfigurable logic block utilization in each configuration, and 2) reduce the runtime reconfiguration overhead. Experimental results show that the proposed strategy leads to over 39% average reduction in runtime reconfiguration overhead for partial runtime reconfiguration. In addition, the proposed strategy leads to an average performance gain of over 32% and 34% for full and partial runtime reconfiguration respectively.**

Keywords- Custom instructions, FPGA, full/partial runtime reconfiguration, loop partitioning, reconfigurable processors.

I. INTRODUCTION

Future system-on-chip platforms are expected to incorporate reconfigurable processors [1] to leverage the computational power of hardware while providing for high instruction set programmability to meet the increasingly tight time-to-market requirements. Reconfigurable processors enable the basic instruction set of the microprocessor to be extended by implementing custom instructions on the reconfigurable space (e.g. Field Programmable Gate Arrays (FPGAs)).

Runtime reconfiguration enables the realization of low cost systems without compromising on performance by allowing the configuration of the hardware to change dynamically during program execution. Although runtime reconfiguration is possible in commercial FPGAs, the fine-grained programmable structure of commercially available reconfigurable architectures results in large reconfiguration overhead. In addition, there is a lack of tools and methodologies to support runtime reconfiguration in commercial FPGAs.

A framework is presented in this paper to rapidly identify a suitable set of runtime configurations or temporal partitions from a given application. Rapid area-time estimation of the custom instructions in the temporal partitions are undertaken to evaluate the benefits of runtime reconfiguration early in the design cycle. The proposed framework incorporates a hierarchical loop partitioning strategy that reduces the search space complexity for determining full and partially reconfigurable custom instructions. The framework leverages the cluster merging technique that we proposed in [2] to increase the benefits of runtime reconfiguration on reconfigurable processors. In this paper, we target area-constrained FPGAs with multi-bit logic blocks and bus-based architecture that facilitate configuration memory sharing, which is similar to [3]. We assume that the smallest possible configuration unit is a multi-bit logic block. In this paper, we assume that the logic blocks consist of 4-input LUTs that are accompanied by fast carry propagation structure. Such logic blocks can be found in commercial FPGA architectures. Experiment results show that both the full and partial reconfiguration models of the target FPGA can benefit notably from the proposed cluster merging based hierarchical loop partitioning strategy.

The paper is organized as follows: In the next section, we discussed related work in runtime reconfiguration for reconfigurable processors and temporal partitioning. Section 3 provides a brief description of the cluster merging technique. In Section 4, we introduce the proposed framework for generating runtime custom instruction configurations for area-constrained reconfigurable processors. We will also provide detailed description of the proposed hierarchical loop partitioning strategy. Next, experimental results will be shown to demonstrate the viability of the proposed strategy for generating runtime configurations based on full and partial reconfiguration models. We conclude the paper in Section 6.

II. RELATED WORK

Previous work has shown the benefits of runtime reconfiguration on commercial reconfigurable processors. For example, the work in [4] has demonstrated runtime reconfiguration for JPEG and H.264 encoder/decoder on Xilinx Virtex FPGA based reconfigurable processors. However, the fine-grained programmable structure in commercial FPGAs necessitates high reconfiguration overhead which may override the speedup obtained through hardware acceleration. This overhead is significant. For example, partial reconfiguration on Xilinx Virtex FPGA [4] and the Stretch processor [5] is in the order of milliseconds. Hence, FPGA architectures with multi-bit logic blocks and bus-based architecture that facilitate configuration memory sharing (e.g. [3]) is an attractive

proposition as they can fully exploit the advantages of runtime reconfiguration.

Tools and methodologies also play an essential role to select custom instructions that can mitigate the high reconfiguration overhead in order to exploit the benefits of runtime reconfiguration in reconfigurable processors. There are many reported works in custom instruction selection but we will not be discussing them here as it is not the focus of this paper. The authors in [6] have provided a good review of the work in this area. Tools and methodologies supporting runtime reconfiguration on reconfigurable processors must also incorporate efficient temporal partitioning strategies that take into account the reconfiguration overhead. Temporal partitioning is required to partition the application into mutually exclusive configurations such that the area requirement of each configuration is within the reconfigurable resource capacity.

The following describes some previously reported work in temporal partitioning. Integer Linear Programming (ILP) has been used for temporal partitioning of application task graph in [7]. This is accompanied by a loop transformation strategy that aims to increase the throughput while minimizing the reconfiguration overhead. The framework in [8] presented a strategy that traverse the loop graph in a hierarchical top-down fashion, while recursively combining nested loops. The work in [9] presented a method that partitions and modifies custom instructions so that they can be mapped onto coarse-grained functional units. The authors in [10] presented a framework which performs temporal partitioning of frequently executed application loops. The framework assumes that custom instruction versions and their corresponding hardware area-time measures are available prior to the partitioning process. Recently, we proposed a hierarchical partitioning strategy that heuristically determines whether the application loops can be merged with existing configurations or unfolded for further evaluation in order to obtain a set of runtime configurations that contain profitable custom instructions [11].

A. Our Contribution

We present a framework that performs temporal partitioning on application loops, which constitute the most frequently executed segments of embedded applications. Unlike the framework in [10], our work does not assume the availability of the hardware area-time measures prior to the partitioning process. Instead, the proposed framework is capable of rapidly estimating the hardware area-time information of the custom instructions in the temporal partitions without undergoing time consuming hardware implementation. Unlike [8], the proposed hierarchical loop partitioning strategy aims to maximize the performance gain of each configuration by increasing the utilization of each configuration and reducing the reconfiguration cost. In particular, the proposed strategy employs cluster merging to maximize the performance gain of the configurations. Finally, unlike our previously reported work in [11], the proposed hierarchical loop partitioning strategy in this paper employs *k*-way partitioning approaches to maximize the performance gain of each configuration and minimize the reconfiguration overhead. In addition, our previous work does not exploit cluster merging for runtime reconfiguration. We will also demonstrate the advantages of the proposed framework for both full and partial runtime reconfiguration in this paper.

III. Cluster Merging

In [2], we proposed the cluster merging technique to generate area-time efficient custom instructions. Figure 1 illustrates an example of cluster merging of two custom instructions G_1 and G_2, with the assumption that there is only one available output port. The cluster merging method first partitions the custom instructions into a set of clusters such that each cluster can be mapped onto a single FPGA logic block. In Figure 1(a), G_1 is partitioned into clusters C_1^1, C_1^2 and C_1^3, and G_2 is partitioned into clusters C_2^4 and C_2^5. Next, clusters from different custom instructions are merged if the resulting merged cluster can still be mapped onto a single FPGA logic block. This process takes into account the architectural constraints of the FPGA device for generating area-time efficient custom instructions. It can be observed that the merged data-path in Figure 1(b) is capable of performing the functionality of the original custom instructions ($x \oplus y$ denotes x and y have been merged).

Unlike the widely-used resource sharing method for area optimization, the proposed cluster merging process does not maximize sharing of common resources and this leads to less reliance on multiplexers for implementing custom instructions. The proposed method has been shown to achieve significantly lower area-delay products when compared to commercial tools and efficient resource sharing methods in the literature. For example, when compared to commercial tools, the proposed cluster merging technique can achieve over 40% average reduction in area costs for certain FPGA devices. Further details of the cluster merging technique can be found in [2].

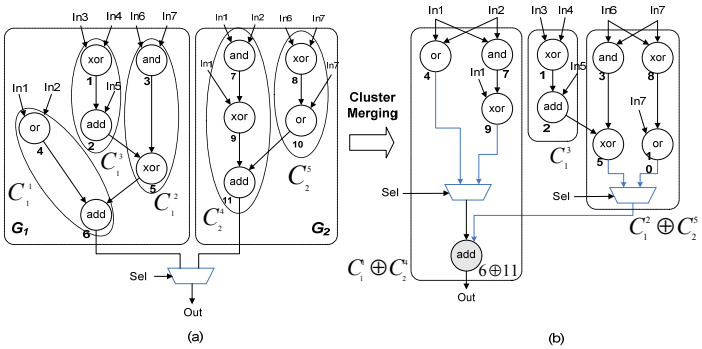

Figure 1: Example of cluster merging for custom instructions G_1 and G_2

It is noteworthy that the results of cluster merging also provide an area-time estimation of FPGA realization due to the architecture-aware nature of the cluster merging process. For example, the merged data-path in Figure 1(b) utilizes three FPGA logic blocks and has a critical path delay that is equivalent to the latency of three FPGA logic blocks. The cluster generation technique used to partition the custom instructions into clusters is formulated based on certain rule-

sets that define the architectural constraints of the target FPGA logic blocks. Investigations show that the cluster generation technique can estimate the average critical paths and area measures of 150 custom instructions from sixteen applications, to be within 3% and 1% respectively of those obtained using hardware synthesis.

In the next section, we describe how the cluster merging technique can lead to performance benefits for runtime reconfiguration.

IV. Proposed Method

Figure 2 shows an overview of the proposed framework. The framework relies on the Trimaran compiler infrastructure [12] to generate the Intermediate Representation (IR) of the C-application in the form of a Data Flow Graph (DFG). The IR serves as input to the Custom Instruction Selection stage to select a set of custom instructions. There are a large number of previously reported works in Custom Instruction Selection [6], all of which can be incorporated in the proposed framework.

Cluster merging is then performed on the selected custom instructions to determine the merged clusters. As discussed in Section 3, the results of cluster merging provides an indication of the area costs and critical path delays of the custom instructions when they are implemented on the reconfigurable multi-bit logic blocks.

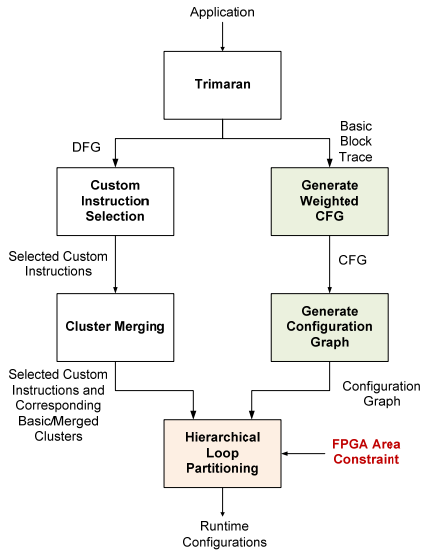

Figure 2: Framework for generating runtime configurations

A configuration graph is then generated to enable temporal partitioning of loops using the proposed hierarchical loop partitioning strategy. We will discuss the generation of the configuration graph in the following sub-section. Note that the partitioning strategy relies on the hardware estimation results from the cluster merging process in order to obtain a set of custom instruction configurations. In addition, the partitioning strategy also utilizes the results from cluster merging to increase the performance gain of the configurations and to reduce reconfiguration cost.

A. Generating the Configuration Graph

The configuration graph is intended to provide visibility of sections of the application that run together and hence would be considered as a group for custom instruction reconfiguration. Figure 3 shows an example of configuration graph generation from the basic block trace of an application obtained from Trimaran.

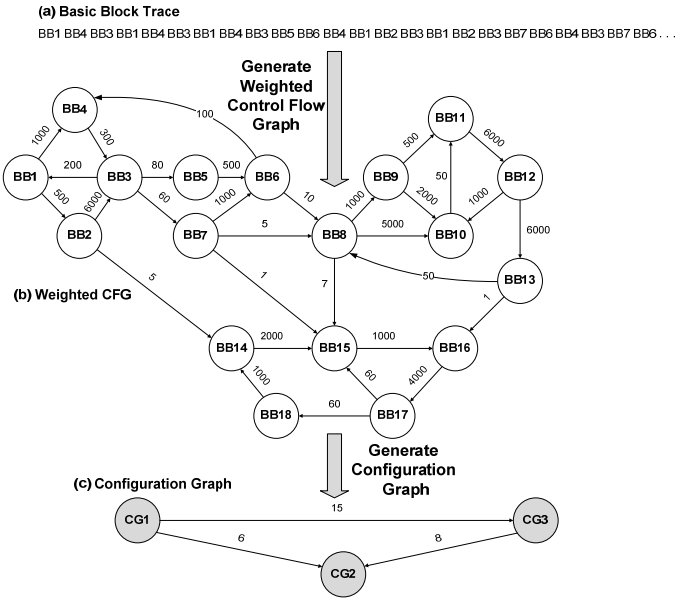

Figure 3: Generating configuration graph from basic block trace

We first convert the basic block trace into a weighted Control Flow Graph (CFG), which encapsulates the control flow between unique basic blocks and the corresponding frequency. In particular, the weighted CFG is a directed graph $G(V,E,w)$, where V is a set of vertices that represent the unique basic blocks in the basic block trace. An edge $e \in E$ is an ordered pair (u,v), where u, $v \in V$, that represents the control flow between basic blocks u and v. Each edge (u,v) is associated with a weight w that represents the frequency of the control flow between u and v.

The configuration graph is a directed graph $G_c(V_c,E_c,w_c)$ that is generated from the weighted CFG. Each vertex $u_c \in V_c$ in the configuration graph, denoted as a configuration, is a set of basic blocks (i.e. $u_c = \{u_1, u_2, ..., u_k\} \in V$) that are reachable from one another. In other words, a cycle can be found between any pair of basic blocks in a configuration. In addition, there are no duplicated basic blocks in different configurations (i.e. $u_c \cap v_c = \emptyset$, where u_c, $v_c \in V_c$ and $u_c \neq v_c$). For example in Figure 3, configuration *CG1* in the configuration graph consists of basic blocks *BB1*, *BB2*, ... *BB7*, configuration *CG3* in the configuration graph consists of basic blocks *BB8*, *BB9*, ... *BB13*, and configuration *CG2* in the configuration graph consists of basic blocks *BB14*, *BB15*, ... *BB17*. It is noteworthy that the basic blocks in each configuration belong to application loops, which are the most frequently executed segments of embedded applications.

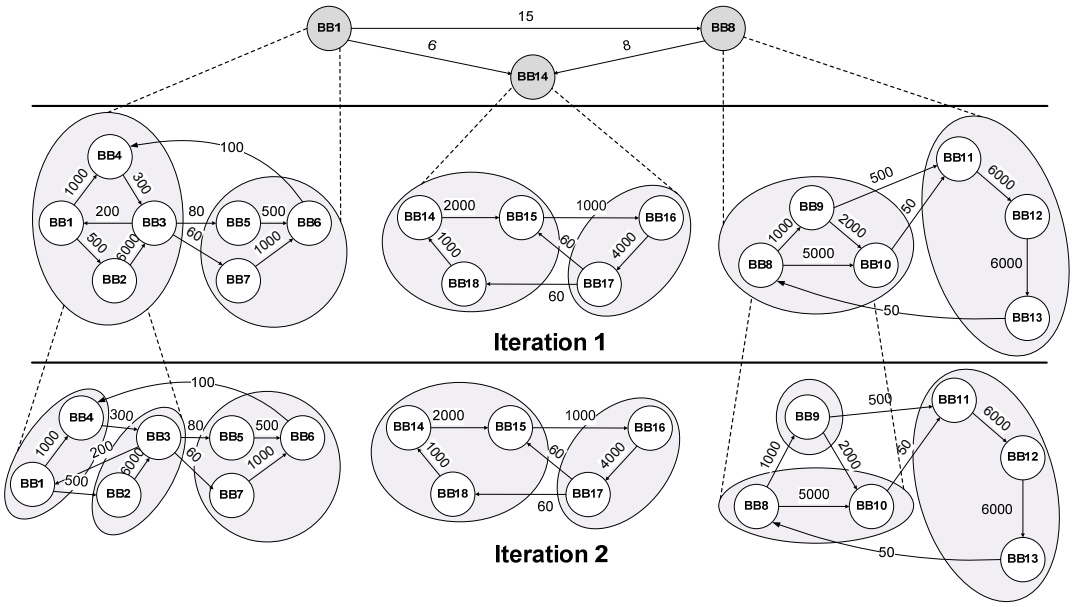

Figure 4: Example of hierarchical loop partitioning

We have used transitive closure to identify the existence of cycles between each pair of basic block in the weighted CFG. Other more efficient methods such as [13] can also be used for identifying cycles. The acyclic graph is then generated by collapsing the basic blocks into the corresponding configurations. It can be observed that the edges of the configuration graph are associated with a weight, which is the sum of edge weights between basic blocks in different configurations. Note that weights of the edges in the configuration graph are typically very small, as these edges represent the less occurring control flow between disjoint loops in the application. Each configuration in the initial configuration graph is a potential runtime configuration candidate. Hence, the weight of an edge in the configuration graph $w_c(u_c,v_c)$, where u_c, $v_c \in V_c$, represent the number of times configuration u_c is reconfigured to v_c.

B. Hierarchical Loop Partitioning

The proposed hierarchical loop partitioning temporally partitions the application loops, in a top-down fashion starting from the initial acyclic configuration graph, into one or more configurations such that the overall performance gain of runtime reconfiguration is maximized. The final output of the partitioning process is a set of configurations and the selected custom instructions in each configuration.

Figure 4 shows an example of the proposed method. In the initial step, the performance gain of the custom instructions in each configuration is calculated. The performance gain is computed by selecting the set of custom instructions in each configuration that leads to the highest software cycle savings while meeting the FPGA area constraint. In the subsequent iterations of the partitioning process, each configuration is partitioned into two new configurations. We have used the multilevel 2-way partitioning algorithm in [13] to partition each configuration into two equal-size parts with the objective to minimize the edge-cut. The edge-cut is defined as the sum of the weight of the straddling edges between the partitions. Each new partition can be represented by a new vertex in G_c, which represents a possible runtime configuration candidate. Note that the partitioning also introduces additional edges in the configuration graph which represents the straddling edges between the basic blocks in the various partitions.

The effective performance gain for each new configuration $x \in V_c$ (in terms of software cycle savings) is computed as shown in Eq. (1), where G_i^x is a custom instruction in configuration x, $F(G_i^x)$ is the execution frequency of instruction G_i^x, $T_S(G_i^x)$ denotes the number of operations in G_i^x, $T_H(G_i^x)$ is the estimated critical path delay of G_i^x (inferred from cluster merging), r is the ratio between the clock frequency of the FPGA and base processor (r is chosen based on the area-optimized configuration of the soft-core processor in [13]), and T_{RTR}^x is the reconfiguration cost of x. The area utilization of all the custom instructions G_i^x in x cannot exceed the FPGA area constraint A_{FPGA} (in terms of number of logic blocks) as shown in Eq. (2). In our work, G_i^x is selected from the set of custom instructions in configuration x that leads to the highest software cycle savings while meeting the FPGA area constraint.

$$SCS(x) = \sum_i^c F\left(G_i^x\right) \cdot \left(T_S\left(G_i^x\right) - r \cdot T_H\left(G_i^x\right)\right) - T_{RTR}^x \quad (1)$$

$$A(x) = \sum_i^c A\left(G_i^x\right) \leq A_{FPGA} \quad (2)$$

$$T_{RTR}^x = \begin{cases} \sum w_c(u_c,x) \times T_{RTR}^{lb} \times A_{FPGA} & \text{if full RTR} \\ \sum w_c(u_c,x) \times T_{RTR}^{lb} \times (A_{FPGA} - n_c) & \text{if partial RTR} \end{cases} \quad (3)$$

The reconfiguration cost of configuration x is computed differently for the full and partial reconfiguration model as shown in Eq. (3). $\sum w_c(u_c, x)$ is the sum of weights of the incoming edges of x in the configuration graph. In other words, $\sum w_c(u_c, x)$ represents the number of times configuration x will be reconfigured on the FPGA at runtime. T_{RTR}^{lb} is the reconfiguration cost of a single multi-bit logic block [3] and is measured in terms of software clock cycles. Finally, n_c is the number of common clusters/merged clusters in configuration x and the previous configuration u_c, i.e. $(u_c, x) \in E_c$. For partial reconfiguration, we can avoid reconfiguring logic blocks with common clusters/merged clusters in two consecutive configurations.

For each partition solution, the total performance gain of the resulting partitions is compared to the performance gain of the initial configuration. If the post-partition performance is less than the initial performance, then the new partitions are discarded and the initial configuration is restored. This can be observed in Iteration 2 of Figure 4, where some of the configurations in Iteration 1 do not lead to any further partitions. The partition process is repeated until no new partitions are formed in a particular iteration. The final set of partitions is the runtime configurations. Note that the proposed hierarchical partitioning strategy reduces the search space by avoiding further partitioning if the resulting partitions do not lead to higher performance.

```
PARTITION-LOOP (G_c, A_FPGA)
 1. partition_exist := true
 2. while (partition_exist = true) {
 3.     partition_exist := false
 4.     for each node u_c ∈ G_c {
 5.         SCS(u_c) = CAL-GAIN (u_c, A_FPGA)
 6.         remove u_c from G_c
 7.         u_c^1, u_c^2 = 2-WAY-PARTITION (u_c)
 8.         insert u_c^1 and u_c^2 in G_c
 9.         SCS(u_c^1) = CAL-GAIN (u_c^1, A_FPGA)
10.         SCS(u_c^2) = CAL-GAIN (u_c^2, A_FPGA)
11.         if SCS(u_c^1) + SCS(u_c^2) < SCS(u_c) {
12.             restore u_c in G_c
13.             remove u_c^1 and u_c^2 from G_c }
14.         else partition_exist := true }}
15. return G_c
```

Figure 5: Pseudo code for hierarchical loop partitioning

Figure 5 shows the pseudo code for the proposed hierarchical loop partitioning strategy. In each iteration (lines 4-14), the performance gain of each existing configuration in the configuration graph G_c is first evaluated (line 5) using the function CAL-GAIN and temporarily removed from G_c (line 6). The existing configuration is then partitioned into two smaller configurations using the 2-WAY-PARTITION function (line 7) and the new configurations are inserted into G_c along with the corresponding edges (line 8). The performance gain of the two new configurations is evaluated (lines 9-10) and compared to the performance gain of the initial configuration (line 11). In the event that the partitioning has led to less favorable performance gain, the initial partition is restored (line 12) in the configuration graph and the new configurations are removed from the configuration graph (line 13). When no new partitions are generated in an iteration (evaluated in line 2), the algorithm returns the configuration graph consisting of the final set of configurations (line 15).

V. EXPERIMENTAL RESULTS

Runtime reconfiguration on reconfigurable processors is only feasible for applications where the performance of the custom instructions can mitigate the high reconfiguration overhead of the FPGA architecture. The proposed framework employs cluster merging to increase the utilization of each configuration by packing larger number of profitable custom instructions in each configuration. In addition, cluster merging may lead to reduction in the partial reconfiguration cost due to larger number of common basic/merged cluster realizations in consecutive runtime configurations. Hence, the proposed strategy can lead to high performance benefits if most of the profitable custom instructions in the application have common clusters.

Table I reports the cluster statistics from Cjpeg application [15]. The first column lists the number of selected custom instructions, the second column lists the number of basic clusters that are obtained using the clustering technique, the third column lists the number of unique basic clusters, and the final column reports the number of unique basic/merged clusters after the cluster merging. The unique clusters in the third and fourth column of Table I is the set of non-isomorphic clusters before and after cluster merging respectively. It can be observed that on average over 71% of the basic clusters in the application are isomorphic. The number of unique clusters can be further reduced by 50% through cluster merging. The notable number of isomorphic clusters found in Cjpeg provides a strong justification for adopting the cluster-based runtime reconfiguration approach.

TABLE 1: CLUSTER STATISTICS FOR CJPEG APPLICATION

Custom Instructions	Basic Clusters	Unique Clusters (Before Cluster Merging)	Unique Clusters (After Cluster Merging)
90	191	54	27

In this section, we will evaluate the proposed hierarchical loop partitioning strategy for both full reconfiguration and partial reconfiguration models. In addition, we will also investigate the impact of cluster merging on increasing the performance gain of runtime reconfigurable RISPs.

A. Full Reconfiguration

The full reconfiguration model requires the complete reprogramming of the entire configuration memory during runtime reconfiguration. Figure 6 shows the total runtime reconfiguration cost (primary axis), which is calculated using Eq. (3), and the number of configurations (secondary axis) for the full reconfiguration model. These values are obtained for

varying FPGA area constraints (in terms of number of logic blocks), i.e. 2% to 20% of the maximum FPGA area that is required to implement all the selected custom instructions.

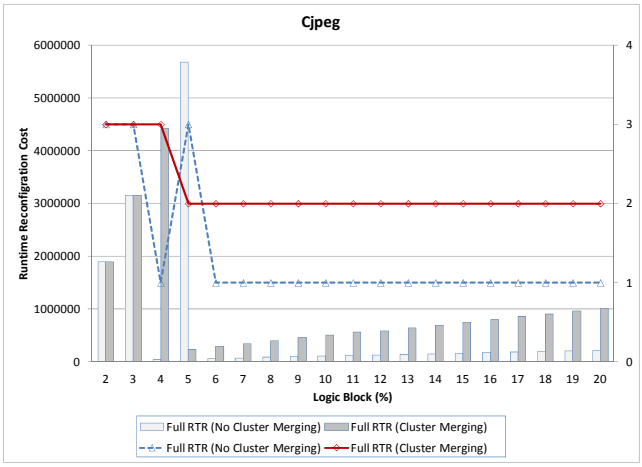

Figure 6: Runtime reconfiguration cost and number of partitions for full reconfiguration model

It can be observed that in general (except for cases where the area constraint is less than 6%), the runtime reconfiguration cost where cluster merging is not considered during hierarchical loop partitioning (denoted as *No Cluster Merging*), is lower than the case where cluster merging is considered during hierarchical loop partitioning (denoted as *Cluster Merging*). This is due to the fact that when cluster merging is taken into account during hierarchical loop partitioning, more configurations are generated as shown in Figure 6. Hence, the number of times the FPGA undergoes reconfiguration at runtime also increases for *Cluster Merging*. The gradual increase in reconfiguration cost for area constraint larger than 5% is due to the increase in the number of logic blocks that undergo runtime reconfiguration for the full reconfiguration model.

Figure 7 compares the performance between *No Cluster Merging* and *Cluster Merging* for the full reconfiguration model. The performance is calculated by summing up the software cycle savings of all the configurations (calculated using Eq. (1)). In addition, the performance without runtime reconfiguration (*No RTR*) is also shown in Figure 7. In order to obtain the performance of *No RTR*, a greedy algorithm is used to select a set of custom instructions that lead to the highest performance while meeting the area constraint. Hierarchical loop partitioning is not employed for *No RTR*.

It can be observed that *No Cluster Merging* outperforms *No RTR* only for a few cases when the area constraint is less than 6%. Thereafter, there is no significant difference between the performance of *No Cluster Merging* and *No RTR*. On the other hand, *Cluster Merging* outperforms both *No RTR* and *No Cluster Merging* for almost all the cases (except for the case where the area constraint is 20%). In particular, on average, *Cluster Merging* outperforms *No RTR* and *No Cluster Merging* by 45.6% and 32.9% respectively. It is noteworthy that *Cluster Merging* can outperform *No Cluster Merging* by over 77% (i.e. for an area constraint of 4%). In addition, *Cluster Merging* outperforms *No RTR* by two times or more for area constraints 2%, 3% and 5%. The performance gain of *Cluster Merging* over the *No RTR* and *No Cluster Merging* gradually decreases when the area constraint is more relaxed due to the increase in reconfiguration cost as shown in Figure 6.

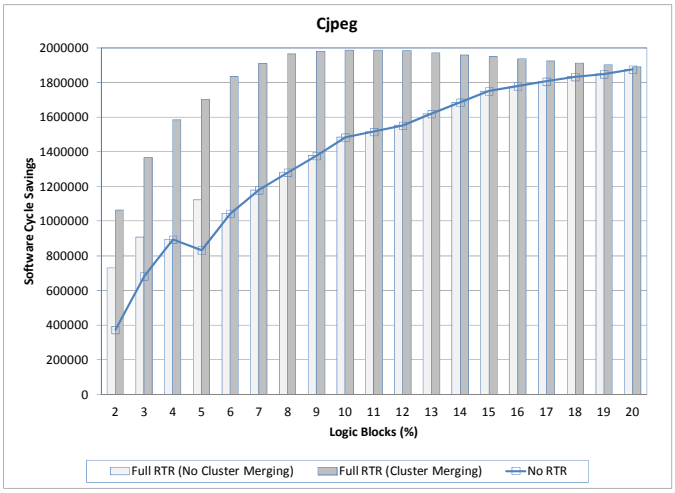

Figure 7: Performance gain for full reconfiguration model

These results show that cluster merging can effectively increase the utilization of the configurations, which has led to the generation of a larger number of configurations. This in turn has resulted in higher performance benefits for the full reconfiguration model.

B. Partial Reconfiguration

Partial reconfiguration enables a portion of the configuration memory to be programmed during runtime reconfiguration and hence this can lead to higher savings in the runtime reconfiguration cost. Figure 8 shows the total runtime reconfiguration cost (primary axis), which is calculated using Eq. (3), and the number of configurations (secondary axis) for the partial reconfiguration model. The range of area constraint is the same as the previous sub-section.

It can be observed that similar to the full reconfiguration method, the number of configurations obtained using the proposed method without considering cluster merging (denoted as *No Cluster Merging*) is generally lower than the number of configurations obtained using the proposed method that takes into account cluster merging (denoted as *Cluster Merging*). However, unlike the full reconfiguration model, the runtime reconfiguration cost of *Cluster Merging* is lower than *No Cluster Merging* for all the area constraints considered. On average, *Cluster Merging* has 39.6% lesser reconfiguration cost compared to *No Cluster Merging*. The maximum percentage of reduction in configuration cost is 49.4% when the area constraint is 6% of the maximum FPGA area. These results imply that cluster merging is capable of reducing the runtime

reconfiguration cost due to the increase in common basic/merged clusters in consecutive configurations.

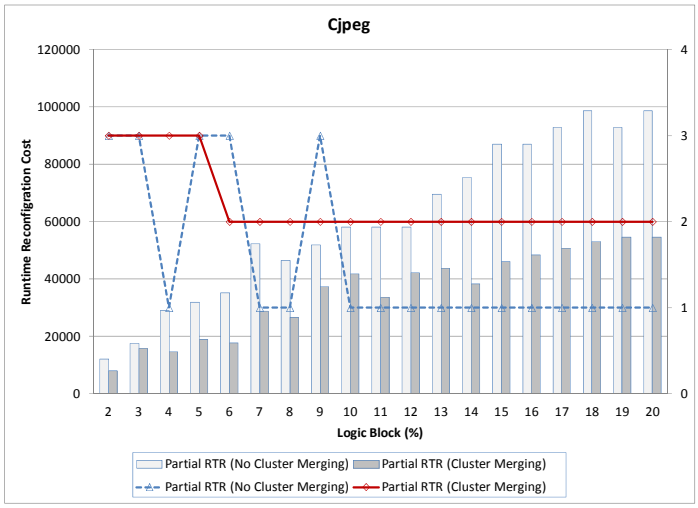

Figure 8: Runtime reconfiguration cost and number of partitions for partial reconfiguration model

Figure 9 compares the performance between *No Cluster Merging* and *Cluster Merging* for the partial reconfiguration model. The performance of *No RTR* is also shown. Firstly, it can be observed that the partial reconfiguration model leads to higher performance than the full reconfiguration model. For example, unlike the full reconfiguration model, *Cluster Merging* in the partial reconfiguration model still outperforms *No RTR* when the area constraint is 20% of the maximum FPGA area. However *No Cluster Merging* outperforms *No RTR* for only a few cases when the area constraint is less than 6%. This shows the significance of cluster merging for increasing the performance of runtime reconfiguration for the partial reconfiguration model. It is also evident that *Cluster Merging* outperforms both *No RTR* and *No Cluster Merging* for all cases. In particular, on average, *Cluster Merging* outperforms *No RTR* and *No Cluster Merging* by 52.2% and 34.9% respectively. In addition, *Cluster Merging* can outperform *No Cluster Merging* by over 86% (i.e. for area constraint of 4%). Similar to the full reconfiguration model, *Cluster Merging* outperforms *No RTR* by two times or more for area constraints 2%, 3% and 5%. Specifically, a maximum performance gain of up to 2.94 times can be observed when the area constraint is 2% of the maximum FPGA area.

These results show that cluster merging can lead to higher performance benefits for the partial reconfiguration model in two ways: 1) increasing the utilization of the configurations, and 2) reducing the runtime reconfiguration cost.

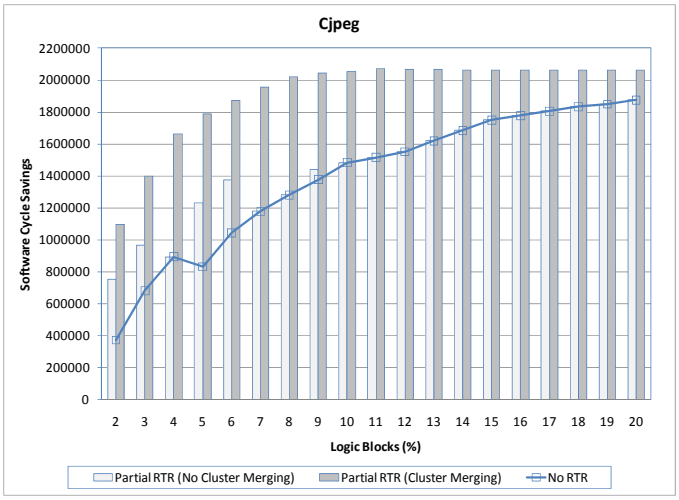

Figure 9: Performance gain for partial reconfiguration model

I. Conclusion

A framework which aims to maximize the performance of custom instructions through runtime reconfiguration, while minimizing the reconfiguration overhead has been presented. The proposed framework incorporates a hierarchical loop partitioning strategy which employs cluster merging to enable a larger number of profitable custom instructions to be implemented in each configuration. Experimental results show that this also leads to the generation of more configurations containing profitable custom instructions. In addition, hierarchical loop partitioning with cluster merging can achieve significant reduction in the reconfiguration cost for the partial reconfiguration model. This is due to the fact that cluster merging results in a larger number of common basic/merged cluster realizations in consecutive runtime configurations. Experiment results show that both the full and partial runtime reconfiguration can benefit notably from the proposed cluster merging based hierarchical loop partitioning strategy. Finally, the benefits of cluster-based runtime reconfiguration for a given application can be easily determined by analyzing the percentage of isomorphic clusters in the selected custom instructions.

References

[1] Francisco Barat, Rudy Lauwereins and Geert Deconinck, "Reconfigurable Instruction Set Processors from a Hardware/Software Perspective", *IEEE Transactions on Software Engineering*, Vol. 28, No. 9, September 2002, pp. 847-862

[2] S.K. Lam, T. Srikanthan and C.T. Clarke, "Architecture-Aware Technique for Mapping Area-Time Efficient Custom Instructions onto FPGAs", *IEEE Transactions on Computers,* Vol. 60, No. 5, May 2011, pp. 680-692

[3] A.G. Ye and J. Rose, "Using Bus-Based Connections to Improve Field-Programmable Gate-Array Density for Implementing Datapath Circuits", *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 14, No. 5, May 2006, pp. 462-473.

[4] L. Bauer, M. Shafique, S. Kramer and J. Henkel, "RISPP: Rotating Instruction Set Processing Platform", *ACM/IEEE/EDA 44th Design Automation Conference*, June 2007, pp.791-796.

[5] Stretch Inc., "S6000 Family Software Configurable Processors", Online: http://www.stretchinc.com/products/s6000.php

[6] Carlo Galuzzi and Koen Bertels, "The Instruction-Set Extension Problem: A Survey", *International Workshop on Applied Reconfigurable Computing (ARC)*, March 2008, pp. 209-220.

[7] M. Kaul, R. Vemuri, S. Govindarajan and I. Ouaiss, "An Automated Temporal Partitioning and Loop Fission Approach for FPGA based Reconfigurable Synthesis of DSP Applications", *Design Automation Conference*, 1999, pp. 616-622.

[8] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure and J. Stockwood, "Hardware-Software Co-Design of Embedded Reconfigurable Architectures", *Design Automation Conference*, 2000, pp. 507-512.

[9] Farhad Mehdipour, Hamid Noori, Morteza Saheb Zamani, Kazuaki Murakami, Mehdi Sedighi and Koji Inoue, "An Integrated Temporal Partitioning and Mapping Framework for Handling Custom Instructions on a Reconfigurable Functional Unit", *Asia-Pacific Computer Systems Architecture Conference*, August 2006, pp. 219-230.

[10] H.P. Huynh, J.E. Sim and T. Mitra, "An Efficient Framework for Dynamic Reconfiguration of Instruction-Set Customization", *Design Automation for Embedded Systems*, Vol. 13, No. 1-2, June 2009, pp. 91-113.

[11] S.K. Lam, Y. Deng, J. Hu, X. Zhou and T. Srikanthan, "Hierarchical Loop Partitioning for Rapid Generation of Runtime Configurations", 6th International Symposium on Applied Reconfigurable Computing, March 2010, pp. 282-293.

[12] Trimaran: An Infrastructure for Research in Instruction-Level Parallelism, Online: http://www.trimaran.org

[13] G. Ramalingam, "Identifying Loops in Almost Linear Time", ACM Transactions on Programming Languages and Systems, Vol. 21, No. 2, March 1999, pp. 175-188.

[14] George Karypis and Vipin Kumar, "A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Orderings of Sparse Matrices", University of Minnesota, September 1998.

[15] Daniel Mattson and Marcus Christensson, “Evaluation of Synthesizable CPU Cores”, M.S. thesis, Chalmers University of Technology, Gothenburg, Sweden, 2004.

[16] The Embedded Microprocessor Benchmark Consortium, Online: http://www.eembc.org/home.php.