Exploiting FPGA-Aware Merging of Custom Instructions for Runtime Reconfiguration

SIEW-KEI LAM, Nanyang Technological University
CHRISTOPHER T. CLARKE, University of Bath
THAMBIPILLAI SRIKANTHAN, Nanyang Technological University

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 performance benefits of reconfigurable processors that support full or partial runtime reconfiguration. The proposed framework achieves this by: 1) providing a means for choosing suitable custom instruction selection heuristics, 2) leveraging FPGA-aware merging of custom instructions to maximize the reconfigurable logic block utilization in each configuration, and 3) incorporating a hierarchical loop partitioning strategy to reduce runtime reconfiguration overhead. We show that performance gain can be improved by employing suitable custom instruction selection heuristics, which in turn depends on the reconfigurable resource constraints and the merging factor (extent that the selected custom instructions can be merged). The hierarchical loop partitioning strategy leads to an average performance gain of over 31% and 46% for full and partial runtime reconfiguration respectively. Performance gain can be further increased to over 52% and 70% for full and partial runtime reconfiguration respectively by exploiting FPGA-aware merging of custom instructions.

Categories and Subject Descriptors: **C [Computer Systems Organization]**: Adaptable Architectures

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Custom instructions, FPGA, full/partial runtime reconfiguration, loop partitioning, reconfigurable processors

1. INTRODUCTION

The computing systems market is increasingly dominated by embedded systems. Non-Recurring Engineering (NRE) costs and Time-To-Market (TTM) will become key factors to market success and future embedded systems will be compelled to reuse off-the-shelf components instead of employing custom chips. At the same time, they need to maintain product differentiation and this will pose major challenges to small companies. In light of this, Field Programmable Gate Arrays (FPGAs) are fast becoming the preferred computing platform dominating the integrated circuit market particularly when concerns over market uncertainties as well as shorter product life cycles of embedded systems cannot be ignored. Today, commercially available FPGAs incorporate reconfigurable processors to provide for high instruction set programmability, while leveraging the computational power of configurable hardware. These reconfigurable processors enable the basic instruction set of the microprocessor to be extended by implementing custom instructions.

Author's addresses: Siew-Kei Lam, Centre for High Performance Embedded Systems, Nanyang Technological University, Singapore; Christopher T. Clarke, Department of Electronic and Electrical Engineering, University of Bath, Bath, United Kingdom; Thambipillai Srikanthan, Centre for High Performance Embedded Systems, Nanyang Technological University, Singapore.

DOI:http://dx.doi.org/10.1145/0000000.0000000

Runtime reconfiguration enables the realization of low cost systems without compromising 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.

In [Lam et. al. 2012], we presented a framework to generate efficient custom instructions for reconfigurable processors that support full or partial runtime reconfiguration. The proposed framework identifies 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 previously proposed in [Lam et al. 2011] to increase the benefits of runtime reconfiguration on reconfigurable processors. We target area-constrained FPGAs with multi-bit logic blocks and bus-based architecture that facilitate configuration memory sharing, which is similar to [Ye et al. 2006]. Experiment results for the Cjpeg application 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.

In this paper, we extend our work in [Lam et. al. 2012] to investigate the effects of different custom instruction selection strategies on runtime reconfiguration. We employed a graph covering algorithm with two widely-used heuristics for custom instructions selection and show that the suitability of the heuristics not only depends on the reconfigurable resource constraints but also on the merging factor (i.e. the extent that the resulting custom instructions can be merged). Specifically, the custom instruction selection heuristic that leads to higher merging factor will result in higher performance gain. For comparisons with the proposed framework, we implemented a knapsack algorithm for custom instruction selection. In addition to the Cjpeg application, we provide experimental results for two other well-known applications (i.e. Sha and BlowfishEnc) to demonstrate the advantages of the proposed framework over the knapsack-based custom instruction selection approach.

2. RELATED WORK

Custom instruction selection aims to select a set of non-overlapping custom instruction instances that best meets the objectives of the design (in terms of area, speed, and/or power consumption). We have previously shown in [Li et. al. 2010] that exact algorithms for custom instruction selection are prohibitive for large sized problems. Hence, approximate solutions, such as heuristic and knapsack-based approaches are often used. The work in [Atasu et. al. 2008][Cong et. al. 2004] formulated the custom instruction selection process as a knapsack problem. The work in [Bonzini et. al. 2008] proposed a hybrid algorithm for recurrence-aware custom instruction selection that combines a greedy covering algorithm and an exact branch and bound algorithm. The method in [Guo et. al. 2003] employs a graph-covering algorithm to maximize the number of covered nodes using a minimum number of custom instructions. Our recent work in [Prakash et. al. 2013] demonstrated that

results of custom instruction selection can be improved by incorporating FPGA architecture characteristics in the selection heuristics.

The work in [Bauer et al. 2007] 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 nullify the speedup obtained through hardware acceleration. This overhead is significant. For example, partial reconfiguration on Xilinx Virtex FPGA and the Stretch processor [Stretch] 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. [Ye et al. 2006]) is an attractive proposition.

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. Integer Linear Programming (ILP) has been used for temporal partitioning of application task graph in [Kaul et al. 1999]. This is accompanied by a loop transformation strategy that aims to increase the throughput while minimizing the reconfiguration overhead. The framework in [Li et al. 2000] presented a strategy that traverse the loop graph in a hierarchical top-down fashion, while recursively combining nested loops. The work in [Mehdipour et al. 2006] presented a method that partitions and modifies custom instructions so that they can be mapped onto coarse-grained functional units. The authors in [Huynh et al. 2009] 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 [Lam et al. 2010].

3. OVERVIEW OF FRAMEWORK

Figure 1 shows an overview of the proposed framework. The framework relies on the Trimaran compiler infrastructure [Trimaran] to generate the Intermediate Representation (IR) of C-application in the form of a Data Flow Graph (DFG). The IR serves as input to the Custom Instruction Identification and Selection stage to determine a set of custom instructions. We have employed a graph covering algorithm that can adopt different objective functions for custom instruction selection. This will be described in Section 4.

Cluster merging is then performed on the selected custom instructions to determine the merged clusters. As discussed in Section 5, 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. 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 and the hierarchical loop partitioning strategy in Section 6. 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.

The target FPGA model, which is described in detail in [Lam et. al. 2008] consists of a set of multi-bit logic blocks that is organized around an interconnection network. Each multi-bit logic block incorporates programmable fine-grained logic elements

that are similar to those found in commercial FPGA architectures. In particular, these logic elements consist of 4-input LUTs that are accompanied by fast carry propagation structure. It is noteworthy that in our framework, the use of the variable to indicate the number of direct inputs to the logic block makes the approach applicable in situations where the number of inputs is different. Unlike commercial architectures, the logic elements within each multi-bit block shares the same configuration memory, which leads to reduce runtime reconfiguration overhead. We assume that the smallest possible configuration unit is a multi-bit logic block. If the computation resource requirement of the custom instructions exceeds the number of available logic blocks in the reconfigurable logic, then the custom instructions are mapped to different configurations. At runtime, a reconfiguration manager automatically loads the required configurations onto the logic blocks for computing the custom instructions.

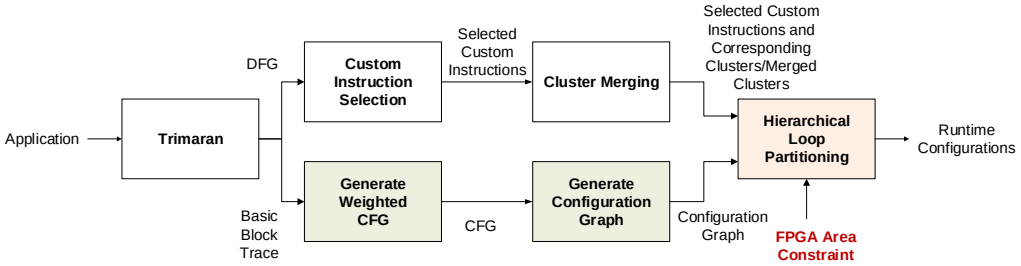

Figure 1: Framework for generating runtime reconfigurations.

4. CLUSTER INSTRUCTION SELECTION

We have used the exhaustive custom instruction enumeration proposed in [Pozzi et. al. 2006] to identify custom instruction instances from the pre-register allocated IR. The same constraint set described in [Lam et. al. 2009] is used in the enumeration process. In particular, only connected integer operations are allowed in the custom instruction instances and the maximum number of input/output ports is 5/2. Previous work has shown that input-output ports more than this range result in little performance gain. Finally, only convex sub-graphs are allowed in the instances.

The custom instruction selection problem can be formulated as follows: Given an application DFG G, a unique set of custom instructions $T = \{T_1, T_2, \ldots, T_i\}$ and the instances of each custom instruction T_i, $I_i = \{I_{i,1}, I_{i,2}, \ldots, I_{i,j}\}$, find a subset of the set I that covers G.

We have adopted the graph covering algorithm presented in [Guo et. al. 2003] to find a set of non-overlapping nodes from the conflict graph based on some objective function. A conflict graph is an undirected graph $G_u(V_u,E_u)$. Each vertex represents a custom instruction instance $I_{i,j}$ that is associated with a unique custom instruction T_i. An edge $e \in E_u$ between two instances signifies that the instances have at least one overlapping node. The number of nodes in an instance $I_{i,j}$ is denoted as $size(I_{i,j})$. The covering algorithm starts by taking all the custom instruction instances and constructing a conflict graph with them. For each unique template T_i, the Maximum Independent Set (MIS) (referred to hereinafter as MIS_i) is the largest subset of instances in T_i for which those instances do not share any common edges (they are mutually non-adjacent). This is established using an iterative approach. The term $size(MIS_i)$ is used in this paper to indicate the number of instances that are contained within MIS_i . The MIS_i with the largest objective function ($w(MIS_i)$) is then selected. All instances that match the selected MIS then become selected instances. After selection, these instances and their neighbors can be removed from the conflict graph.

This algorithm is repeated until the conflict graph is empty. The computation of the MIS can be implemented in time linear in the number of vertices and edges of G_u [Halldórsson et. al. 1994].

The choice of objective function (i.e. $w(MIS_i)$) will have significant impact on the custom instruction selection process. In this paper, we have evaluated two commonly used objective functions: 1) Most-Frequent-Largest-Fit-First (MLFF), and 2) Largest-Fit-First (LFF). The objective function for MLFF is: $w(MSI_i) = size(v_x) \times size(MSI_i)$, which takes into account both the frequency of custom instruction occurrence and the size of the custom instructions. The objective function for the LFF heuristic is: $w(MSI_i) = size(v_x)$. The LFF approach attempts to select the MIS with the largest instances first. The selected custom instructions using the objective function MLFF or LFF will then undergo the cluster merging process described in the next section.

5. CLUSTER MERGING

In [Lam et. al. 2011], we proposed the cluster merging technique to generate area-time efficient custom instructions. Figure 2 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. Each custom instruction consists of a set of primitive integer arithmetic (e.g. addition, subtraction, multiplication), logical (and, or, xor), and relational (e.g. logical and arithmetic shift) operations.

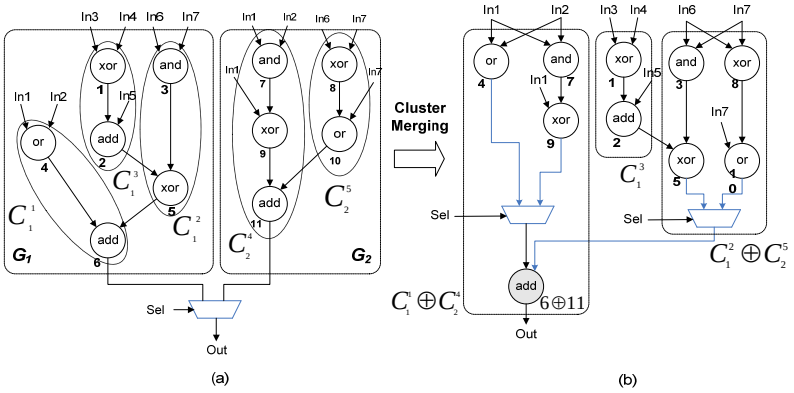

Figure 2: Example of cluster merging for custom instructions G_1 and G_2.

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. This resembles the technology mapping process, where a set of clusters that effectively cover each custom instruction is identified. In Figure 2(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 2(b) is capable of performing the functionality of the original custom instructions ($x \oplus y$ denotes x and y have been merged). A heuristic is used to select a unique set of merged clusters with the aim to maximize the area utilization of the FPGA resources. As discussed in [Lam et. al. 2011], the time complexity of cluster merging is $O(|V_u|^2)$, where $|V_u|$ is the number of clusters that are evaluated in each iteration.

The results of cluster merging also provide 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 2(b) utilizes three FPGA logic blocks and has a critical path delay that is equivalent to the latency of three FPGA logic blocks. In the next section, we describe how the cluster merging technique can lead to performance benefits for runtime reconfiguration.

6. TEMPORAL PARTITIONING

6.1 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. The basic block trace lists the actual execution sequence of the basic blocks for a given input dataset.

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. This is achieved with a simple program that parses the basic block trace and constructs an adjacency matrix which records the control edges of each basic block. 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 *C1* in the configuration graph consists of basic blocks *BB1*, *BB2*, ... *BB7*, configuration *C3* in the configuration graph consists of basic blocks *BB8*, *BB9*, ... BB13, and configuration *C2* in the configuration graph consists of basic blocks *BB14*, *BB15*, ... *BB18*. It is noteworthy that the basic blocks in each configuration belong to application loops, which are the most frequently executed segments of embedded applications.

We have used transitive closure to identify the existence of cycles between each pair of basic block in the weighted CFG. 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.

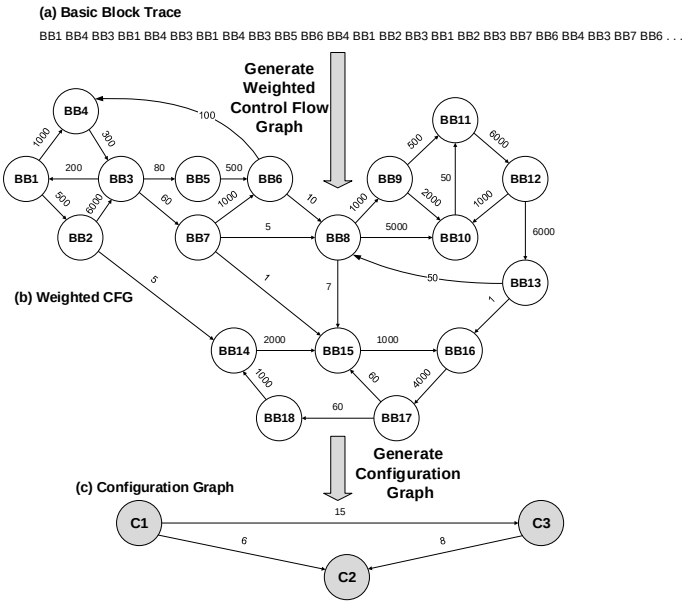

Figure 3: Generating configuration graph from basic block trace.

6.2 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 *C1*, *C2* and *C3* of the configuration graph (see Figure 3c), 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 using Eq. (1). Details of the performance gain computation will be discussed later.

In the subsequent iterations of the partitioning process, each configuration is partitioned into two new configurations. For example in Figure 4, *C1* is partitioned to *C1.1* and *C1.2*; *C2* is partitioned to *C2.1* and *C2.2*; and *C3* is partitioned to *C3.1* and *C3.2*. We have used the multilevel 2-way partitioning algorithm in [Karypis et al. 1998a] to partition each configuration into two 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.

For each partition solution, the total performance gain (calculated using Eq. (1)) 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. In particular, *C1.2*, *C2.1*, *C2.2* and *C3.2* do not undergo further partitions as doing so will not lead to improved performance gain. For example, consider that *C1.2* is further partitioned to two

configurations (e.g. *C1.2.1* that consists of BB5, and *C1.2.2* that consists of BB6 and BB7). The partitioning process will now result in additional overhead that is required to reconfigure *C1.2.1* to *C1.2.2*. In particular, we need to take into account the time required for the additional 500 reconfigurations. Assuming that this overhead obviates the benefits of the partitioning process, we have to discard the configurations *C1.2.1* to *C1.2.2* and retain the original partition *C1.2*.

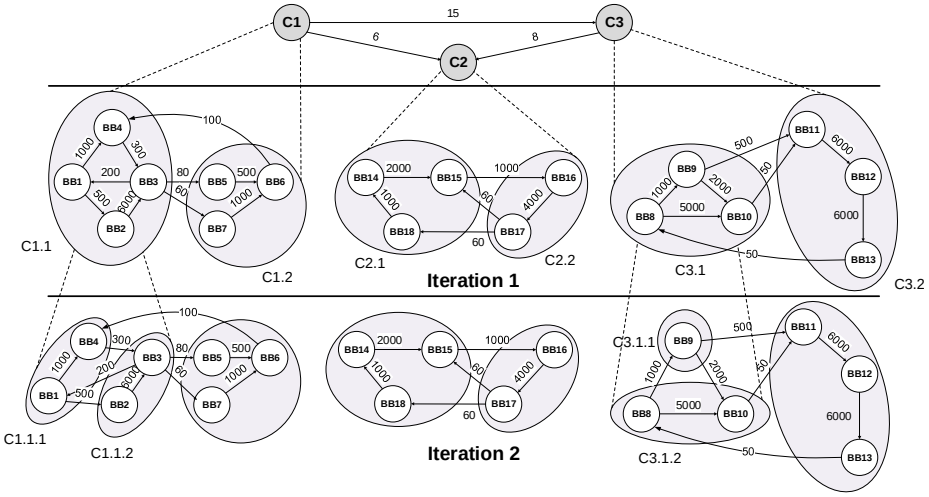

Figure 4: Example of hierarchical loop partitioning.

The partition process is repeated until no new partitions are formed in a particular iteration. [Karypis et al. 1998b] has shown that the k-way graph partitioning can be achieved in $O(|E|)$ time, where $|E|$ is the number of edges in the graph. Since each configuration may be partitioned to two in an iteration, the total number of partitions evaluated is at most twice the number of resulting partitions. 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. The final set of partitions is the runtime configurations.

ALGORITHM 1 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).

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, and T_{RTR}^x is the reconfiguration cost of x. Note that Eq. (1) makes the following assumptions: each operation executes in one clock cycle on the soft-core processor and at any one time, only the base instruction or custom instruction is executed.

ALGORITHM 1. Hierarchical Loop Partitioning

```
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
```

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.

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

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

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

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 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.

7. 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. 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 BlowfishEnc, Sha [Guthaus et. al. 2001] and Cjpeg application [EEMBC]. The second and third column lists the number of selected custom instructions, the fourth and fifth column lists the average size of the selected custom instructions (in terms of number of operations), the sixth and seventh column lists the number of basic clusters that are obtained using the clustering technique, the eighth and ninth column lists the number of unique basic clusters, and the final two columns report the number of unique basic/merged clusters after the cluster merging. Results for the MLFF and LFF approaches are reported. The unique clusters in the last four columns of Table I are the set of non-isomorphic clusters (i.e. each of the clusters are unique in their operations and interconnectivity between operations) before and after cluster merging. In the last two columns (i.e. unique cluster after merging), the *merging factor* (calculated as the ratio of basic clusters and unique clusters) is reported in brackets. The merging factor signifies the extent that the custom instructions can be merged.

It can be observed that the merging factor of Cjpeg is the highest among the three applications. In addition, the merging factor of the LFF approach in Cjpeg is significantly higher that the MLFF approach. It can also be observed that on average over 34% and 49% of the basic clusters in the three applications are isomorphic for MLFF and LFF approaches respectively. The number of unique clusters can be further reduced by an average of over 45% and 39% through cluster merging for MLFF and LFF approaches respectively. The notable number of isomorphic clusters found in these applications provides a strong justification for adopting the cluster-based runtime reconfiguration approach.

Table I. Cluster Statistics

Application	Custom Instructions		Average Size		Basic Clusters		Unique Clusters (Before Merging)		Unique Clusters (After Merging)	
	MLFF	LFF	MLFF	LFF	MLFF	LFF	MLFF	LFF	MLFF	LFF
BlowfishEnc	7	8	3.14	3.38	13	15	8	9	5 (2.6)	6 (2.5)
Sha	8	8	2.27	3.01	11	17	9	11	5 (2.2)	7 (2.4)
Cjpeg	52	90	3.13	5.00	81	191	43	54	20 (4.1)	27 (7.1)

In the following sections, 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 the different custom instruction selection approaches for increasing the performance gain of runtime reconfigurable processors. In the experiments we have chosen $r = 3$ in Eq. (1) based on the area-optimized configuration of the MicroBlaze soft-core processor [Mattson et. al. 2004].

7.1 Full Reconfiguration

The full reconfiguration model requires the complete reprogramming of the entire configuration memory during runtime reconfiguration. The charts in the left of Figure 5 compares the performance between the hierarchical loop partitioning

without cluster merging (No Merging) and hierarchical loop partitioning with cluster merging (Merging) for the full reconfiguration model. The results for MLFF and LFF approaches are shown. 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 order to obtain the performance of No RTR, a custom instruction selection algorithm based on the knapsack approach is used to select a set of custom instructions that lead to the highest performance while meeting the area constraint. It is noteworthy that No RTR outperforms the baseline processor (where no custom instructions are deployed) by 20.6%, 35.8% and 14.8% for BlowfishEnc, Sha and Cjpeg respectively. Hierarchical loop partitioning is not employed for No RTR. These values are obtained for varying FPGA area constraints in terms of percentage of the maximum FPGA logic blocks that is required to implement all the selected custom instructions for each application. Larger area constraints are not shown as they will not lead to notable performance gains in the approaches considered.

It can be observed that for the LFF approach, No Merging either does not outperform No RTR (i.e. BlowfishEnc) or only outperforms No RTR for a few cases when the area constraint is less than or equal to 6% (i.e. Sha and Cjpeg). Thereafter, there is no significant difference between the performance of No Merging and No RTR. On the other hand, Cluster Merging (denoted as Merging) outperforms both No RTR and No Merging for: 1) BlowfishEnc when area constraint is 14%, 2) Sha when area constraint is 36% and 42%, and 3) Cjpeg when area constraint is less than 30%.

The MLFF approach performs significantly better than the LFF approach in BlowfishEnc and Sha when the area constraint is small. For example, in BlowfishEnc, MLFF outperforms No RTR for area constraint up to 27% and in Sha, MLFF outperforms No RTR for area constraint up 53%. Cluster Merging provides additional performance gain in certain cases for these applications. However for Cjpeg, the LFF approach is more favorable across all area constraints. These results clearly demonstrate that the choice of custom instruction selection approach is essential to obtain high performance gain in runtime reconfiguration. MLFF usually produces smaller custom instructions compared to LFF (which has a preference for selecting larger custom instructions first) and hence the MLFF approach is typically more favorable when the area constraint is tight. This is evident in BlowfishEnc and Sha where MLFF performs better than LFF for small area constraints but becomes less favorable when the area constraints are relaxed. However for Cjpeg, LFF is preferred even when the area constraints is small due to its high merging factor (see Table 1). The high merging factor enables higher utilization of the FPGA space, which in turns result in higher performance per unit area for the LFF approach.

We define *performance threshold* as the point at which the hierarchical loop partitioning strategy (using the best custom instruction selection approach) is no longer feasible when the area constraint is increased further. For area constraints higher than the performance threshold, runtime reconfiguration does not lead to any benefits. The performance threshold for BlowfishEnc, Sha and Cjpeg occurs when the area constraint is 27%, 53% and 28% respectively. Note that beyond these performance thresholds, No RTR gives comparable performance with the hierarchical loop partitioning approach. Hence, one of the benefits of the framework is that it provides a means for identifying the minimal area that is required for achieving maximal performance through custom instructions. The experimental results clearly demonstrate this as the performance gain of runtime reconfiguration exceeds the performance gain of No RTR in many cases when the area constraint is less than the performance threshold.

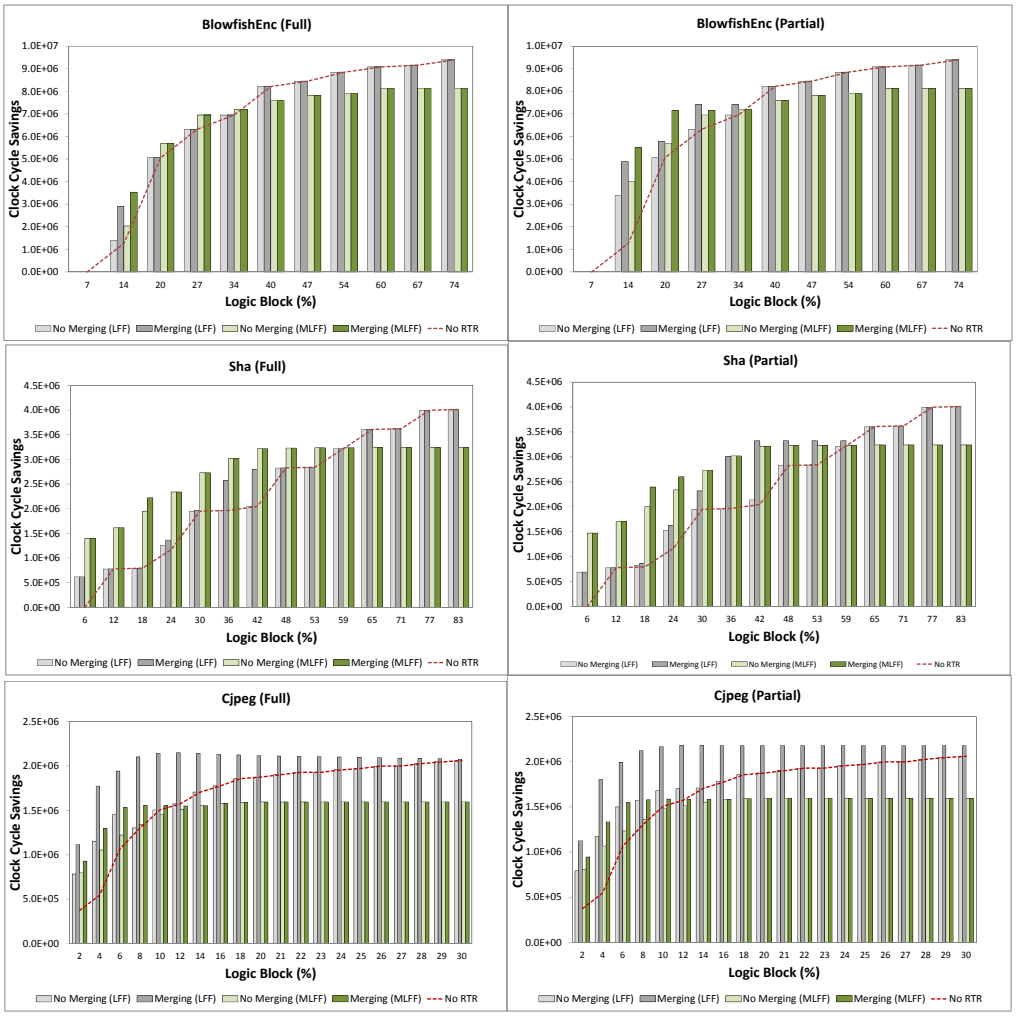

Figure 5: Performance gain for full (left) and partial (right) reconfiguration models

The average performance gain can then be calculated for all the area constraints up to (and including) the performance threshold. As shown in Figure 5, when compared to No RTR, the proposed hierarchical loop partitioning strategy (No Merging) leads to an average performance gain of 20.8% for BlowfishEnc (using MLFF approach), 59% for Sha (using MLFF approach), and 13.1% for Cjpeg (using LFF approach). When cluster merging is employed (Merging), the average performance gain over No RTR increases to 50.5% for BlowfishEnc, 62.9% for Sha, and 42.8% for Cjpeg.

The charts on the left of Figure 6 shows the total runtime reconfiguration cost (represented with lines), which is calculated using Eq. (3), and the number of configurations (represented by columns) for the full reconfiguration model. It can be observed that when the area constraint is relaxed, the number of configurations generally reduces to a point where it will not change anymore. In BlowfishEnc and Sha, the number of configurations reduces to 1 for all the methods considered when the area constraint is 20% and 48% respectively. When the area constraint is further

relaxed, runtime reconfiguration no longer leads to any performance gain. It can also be observed that the number of configurations in No Merging is always equal to or less than Merging for both the LFF and MLFF approaches. This is due to the fact that when cluster merging is taken into account during hierarchical loop partitioning, more configurations could be generated as the overall performance gain compensates for the runtime reconfiguration cost. The reconfiguration cost for LFF (Merging) in Cjpeg gradually increases due to the increase in the number of logic blocks that undergo runtime reconfiguration. This shows the LFF (Merging) approach for Cjpeg can effectively increase the utilization of the configurations, which in turn lead to the generation of more configurations.

7.2 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. The charts on the right of Figure 5 compare the performance between No Merging, Merging and No RTR for the partial reconfiguration model.

Compared to full reconfiguration, partial reconfiguration leads to higher performance gain in No Merging. This is evident for BlowfishEnc and Cjpeg where No Merging outperforms No RTR for area constraints up to 14% and 12% respectively. Similar to the full reconfiguration model, the MLFF approach is generally more favorable for BlowfishEnc and Sha when the area constraints are tight, while the LFF approach is preferred for Cjpeg. This is explained in the previous sub-section whereby custom instruction selection approaches that leads to higher merging factor results in better performance. However, partial reconfiguration results in performance advantage for larger number of design points when compared to full reconfiguration. In particular, the performance threshold of the partial reconfiguration model increases to 34%, 59% and 30% for BlowfishEnc, Sha and Cjpeg respectively. Based on the performance threshold of the full reconfiguration model, the proposed hierarchical loop partitioning strategy (No Merging) outperforms No RTR by an average of 60.1% for BlowfishEnc (using MLFF approach), 61.1% for Sha (using MLFF approach), and 16.8% for Cjpeg (using LFF approach). With cluster merging (Merging), the average performance gain over No RTR increases to 97.6% for BlowfishEnc, 69.1% for Sha, and 45.5% for Cjpeg.

The charts on the right of Figure 6 shows the total runtime reconfiguration cost (represented with lines), which is calculated using Eq. (3), and the number of configurations (represented by columns) for the partial reconfiguration model. Similar to the full reconfiguration method, the number of configurations obtained with No Merging is always lower than or equal to the number of configurations obtained with Merging. An exception to this is a few cases in the LFF method for Cjpeg when the area constraints are 8%-12%. The number of configurations generated is also generally higher than the full reconfiguration model. For example, in BlowfishEnc and Sha, the area constraint at which the number of configurations reduces to 1 for all the methods considered increases to 40% and 65% respectively. In addition, the runtime reconfiguration cost of the LFF (Merging) for Cjpeg is evidently smaller than the full reconfiguration model as the area constraint is relaxed. This is due to the fact that unlike full reconfiguration, the partial runtime reconfiguration cost is not dependent on the area constraint but on the common clusters/merged clusters in consecutive configurations. 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.

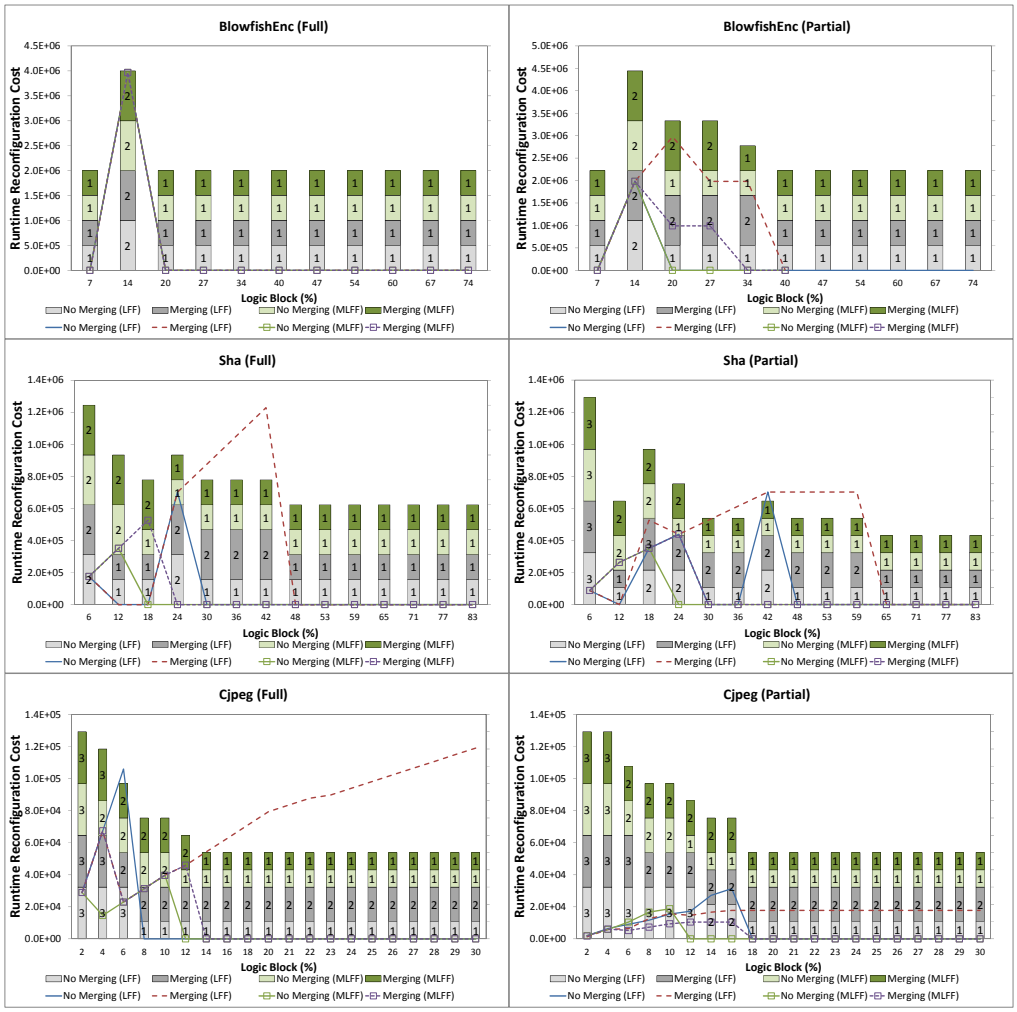

Figure 6: Runtime reconfiguration cost and number of configurations

8. 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 that employs cluster merging to enable a larger number of profitable custom instructions to be implemented in each configuration. In particular, cluster merging can effectively increase the utilization of the configurations, resulting in higher performance per unit area. Cluster merging also plays an important role to determine the best custom instruction selection strategy for runtime reconfiguration. Our analysis reveals that performance gain can be improved by employing a custom instruction selection heuristic that results in a higher merging factor. Experiment results show that both the full and partial runtime reconfiguration can benefit notably from the proposed cluster merging based hierarchical loop partitioning strategy when appropriate cluster selection strategy is adopted.

REFERENCES

Atasu, K., Özturan, C., Dündar, G., Mencer, O., and Luk, W. 2008. CHIPS: Custom Hardware Instruction Processor Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 27, No. 3*, 528-541.

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

Bonzini, P. and Pozzi, L. 2008. Recurrence-Aware Instruction Set Selection for Extensible Embedded Processors. *IEEE Transactions on Very Large Scale Integration Systems, Vol. 16, No. 10*, 1259-1267.

Cong, J., Fan, Y., Han, G. and Zhang, Z. 2004. Application-Specific Instruction Generation for Configurable Processor Architectures. *Proceedings of the ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, 183-189.

EEMBC: The Embedded Microprocessor Benchmark Consortium, Online: http://www.eembc.org

Guo, Y., Smit, G.J.M., Broersma, H., and Heysters, P.M. 2003. A Graph Covering Algorithm for a Coarse Grain Reconfigurable System. *Proceedings of the ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, 199-208.

Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., and Brown, R.B. 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. *IEEE International Workshop on Workload Characterization*, 3-14.

Halldórsson M. and Radhakrishna J. 1994. Greed is Good: Approximating Independent Sets in Sparse and Bounded-Degree Graphs. *Proceedings of the Annual ACM Symposium on Theory of Computing*, 439-448

Huynh H.P., Sim J.E., and Mitra, T. 2009. An Efficient Framework for Dynamic Reconfiguration of Instruction-Set Customization. *Design Automation for Embedded Systems*, 91-113.

Karypis, G., and Kumar, V. 1998a. A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Orderings of Sparse Matrices. University of Minnesota.

Karypis, G., and Kumar, V. 1998b. Multilevel k-way Partitioning Scheme for Irregular Graphs. Journal of Parallel and Distributed Computing, Vol. 48, pp. 96-129.

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

Lam, S.K., Krishnan, B.N., and Srikanthan T., 2006. Efficient Management of Custom Instructions for Run-Time Reconfigurable Instruction Set Processors. IEEE International Conference on Field Programmable Technology, 261-264.

Lam, S.K., Huang, F., Srikanthan, T., and Wu, J. 2008. Run-Time Management of Custom Instructions on a Partially Reconfigurable Architecture. *IEEE International Conference on Electronic Design*.

Lam, S.K., and Srikanthan, T. 2009. Rapid Design of Area-Efficient Custom Instructions for Reconfigurable Embedded Processing. In *Journal of Systems Architecture*, Vol. 55, No. 1, 1-14.

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

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

Lam, S.K., Srikanthan, T., and Clarke, C.T. 2012. Exploiting FPGA-Aware Merging of Custom Instructions for Runtime Reconfiguration. In *7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip*.

Li T., Wu J., Lam S.K. and Srikanthan T. 2010. Selecting Profitable Custom Instructions for Reconfigurable Processors, *Journal of Systems Architecture, Vol. 56, No. 8*, 340-351.

Li Y., Callahan T., Darnell E., Harr R., KurkureU. and Stockwood J.. 2000. Hardware-Software Co-Design of Embedded Reconfigurable Architectures. In *Design Automation Conference*, 507-512.

Mattson, D., and Christensson, M. 2004. Evaluation of Synthesizable CPU Cores. M.S. thesis, Chalmers University of Technology, Gothenburg, Sweden.

Mehdipour, F., Noori, H., Zamani, M.S., Murakami, K., Sedighi, M., and Inoue, K. 2006. An Integrated Temporal Partitioning and Mapping Framework for Handling Custom Instructions on a Reconfigurable Functional Unit. In *Asia-Pacific Computer Systems Architecture Conference*, 219-230.

Prakash, A., Lam, S.K., Clarke, C.T., and Srikanthan, T. 2013. FPGA-Aware Techniques for Rapid Generation of Profitable Custom Instructions, In *Microprocessors and Microsystems, Vol. 37, No. 3*, pp. 259–269

Stretch Inc. S6000 Family Software Configurable Processors. Online: http://www.stretchinc.com/products/s6000.php

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

Ye, A.G., and Rose, J. 2006. 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, 462-473.