Architecture-Aware Technique for Mapping Area-Time Efficient Custom Instructions onto FPGAs

Siew-Kei Lam, *Member, IEEE*, Thambipillai Srikanthan, *Senior Member, IEEE,* and Christopher T. Clarke

Abstract— Area-time efficient custom instructions are desirable for maximizing the performance of reconfigurable processors. Existing data path merging techniques based on resource sharing can be deployed to improve area efficiency of custom instructions. However, these techniques lead to large increase in the critical path delay. In this paper, we propose a novel strategy that takes into account the architectural constraints of the FPGA device in order to realize custom instructions with low area-delay product. The proposed strategy is based on partitioning the custom instruction data-paths into a set of basic clusters such that they can be combined using a heuristic based cluster merging process to maximize the utilization of FPGA logic blocks. Unlike the resource sharing method, the proposed cluster merging process does not maximize sharing of common resources and this leads to lesser reliance on multiplexers for implementing custom instructions. Resource sharing is only applied sparingly at the final stage to increase utilization of logic blocks. We show that the proposed technique leads to more than 34%, 34% and 42% average reduction in area costs for Spartan-3, Virtex-4 and Virtex-5 architectures respectively when compared to optimizations achieved through commercial synthesis tool. We have also shown that the proposed technique leads to more than 18%, 17% and 13% average reduction in area costs for Spartan-3, Virtex-4 and Virtex-5 respectively when compared to results obtained using one of the most efficient resource sharing based method reported in the literature. In addition, the proposed technique outperforms the resource sharing based method in terms of area-delay product, with average reductions of more than 27%, 34% and 19% for Spartan-3, Virtex-4 and Virtex-5 respectively.

Index Terms— Automatic synthesis, data-path design, real-time and embedded systems, reconfigurable hardware

1 Introduction

Future embedded systems must continue to meet the shrinking time-to-market window and lower NRE (Non-Recurring Engineering) costs. Product differentiation needs and increasing complexity of applications will demand customization in the form of hardware acceleration. To this end, FPGAs (Field Programmable Gate Arrays) are gaining popularity as the increasing NRE costs of Application-Specific Integrated Circuits begin to outweigh the per-unit-cost of FPGAs for high-volume applications 0-[2].

Platforms which consist of a microprocessor core that is tightly coupled with a RFU (Reconfigurable Functional Unit) are defined as reconfigurable processors. The instruction set extension capability of reconfigurable processors (e.g. [3]-[5]) that facilitate critical parts of the application to be implemented in hardware, provides an attractive means to meet the flexibility, performance, and cost demands of embedded computing devices [6]. The use of FPGA-based embedded processing is growing and it is estimated that by 2010, more than 40% percent of all FPGA designs will contain an embedded microprocessor [7]. Strategies for maximizing the area utilization of FPGA based implementations will be an important step for satisfying the tight design constraints in future embedded System-On-a-Chip designs.

In this paper, we propose a novel high-level optimization strategy for realizing area-time efficient custom instructions on FPGA devices. In particular, we show that the well-accepted notion of resource sharing for achieving area-efficient designs does not necessarily lead to best results for FPGA-based realizations. The proposed approach is targeted towards commercial FPGA architectures (i.e. [8]-[10]) that typically consist of logic elements with LUT (Look-Up Tables) of any arbitrary input *K* (i.e. *K*-LUT) and a carry-logic structure.

The remainder of this paper is organized as follows: in the following sub-section, we discuss existing high-level area optimization approaches based on resource sharing. Section 2 provides an overview of the proposed strategy and highlights the differences between our proposed method and existing approaches. Section 3 describes the proposed technique in detail. Experimental results are provided in Section 4 and the paper concludes in Section 5.

- *S.K. Lam and T. Srikanthan are with the Centre for High Performance Embedded Systems, Nanyang Technological University, Singapore 637553. E-mail: assklam@ntu.edu.sg, astsrikan@ntu.edu.sg.*
- *C.T. Clarke is with the Department of Electronic and Electrical Engineering, University of Bath, BA2 7AY Bath, U.K. E-mail: c.t.clarke@bath.ac.uk..*

Manuscript received 15 December 2009.

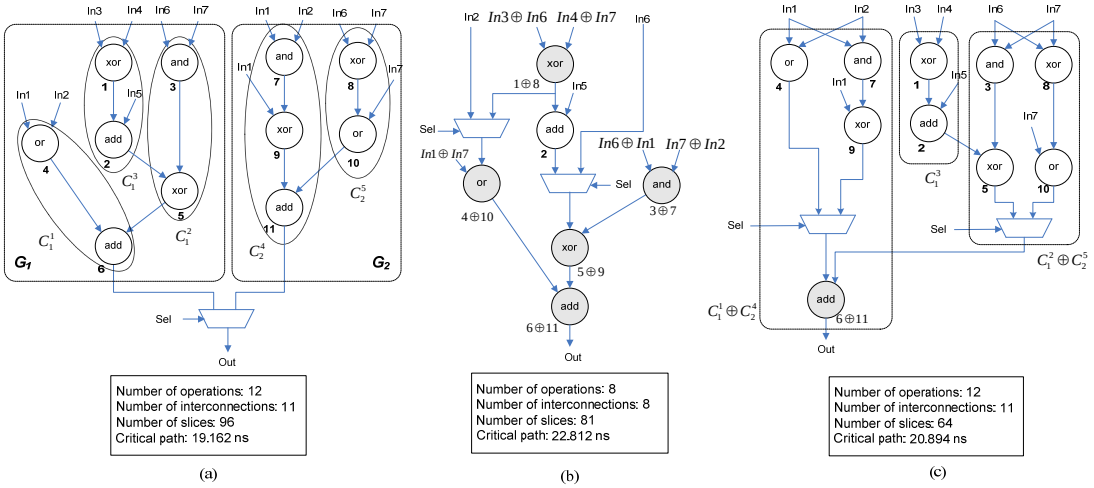


Fig. 1. (a) Original, (b) Resource sharing, (c) Cluster merging

1.1 Related Work

Most of the reported works in custom instruction generation [11]-[16] do not focus on efficient area-time mapping of the custom instructions onto the architecture. Area-time optimization can be performed during high-level [17] or gate level synthesis (i.e. technology mapping [18]-[19]). In contrast to gate level synthesis which operates on the available logic gates of a target architecture library, high-level synthesis operates on the primitive operations derived from the behavioral/algorithmic representations. Since designs at higher level of abstractions are less confined to the physical architecture, optimizations at these levels usually lead to high quality results.

High-level area minimization has often relied on strategies to maximize resource sharing of the data-paths. These approaches aim at maximizing the reuse of operations and interconnections by identifying similarities between two original data-paths. Commercial FPGA tools also adopt the resource sharing approach as one of their main area optimization strategies. For example, the Xilinx synthesis tool supports resource sharing for a limited number of hardware resources (e.g. adders, subtractors and multipliers), by implementing one single arithmetic operator for similar operations that are never used concurrently [20].

Figure 1 illustrates an example of resource sharing of two custom instruction data-paths (i.e. G_1 and G_2 in Figure 1(a)). In this example, we assume that there is only one available output port on the RFU and, hence, the outputs of G_1 and G_2 have been multiplexed. The two custom instruction data-paths in Figure 1(a) are combined into a single data-path in Figure 1(b) by merging similar operations and interconnections between the two data-paths. $x \oplus y$ denotes that operations/inputs x and y has been merged. The resulting data-path in Figure 1(b) is capable of performing the functionality of the original data-paths. It can be observed that the resulting number of operations and interconnections has been reduced. Figure 1 also reports the FPGA implementation results which shows that the area (in terms of number of slices) have been reduced due to resource sharing.

Resource sharing has also been employed for reducing the system reconfiguration overhead of reconfigurable architectures [21]-[22]. The work in [21] minimizes the run-time reconfiguration time by identifying common components in two successive configurations. A weighted bipartite graph is constructed from two successive configurations, and an algorithm that performs graph matching and combining is employed to produce a combined configuration. In [22], a data-path merging algorithm has been presented to reduce the reconfiguration overhead of an architecture template with a reconfigurable interconnection network. In order to optimize interconnect-sharing, the data-paths are merged by solving the maximum bipartite matching problem.

The work in [23], which was later extended in [24], represents custom instruction data-paths as path sequences, and computes the longest common subsequence to identify possible resource sharing between the sequences. In [25], the left-edge binding algorithm was employed to minimize the number of functional units and registers through resource sharing. Other methods for improving the interconnection mapping include iterative improvement and ILP (Integer Linear Programming) approaches [26].

In [27], a data-path merging algorithm is presented to merge several DFGs (Data-Flow Graphs) in order to produce a reconfigurable data-path with minimum hardware operations and interconnection. The algorithm first constructs a compatibility graph that represents all the possible mappings of common operations and interconnectivity between two DFGs. The maximum weight clique problem is then solved to maximize resource sharing between the two DFGs. The authors in [27] demonstrated that their technique outperforms other approaches based on bipartite matching, on iterative improvement and on ILP.

1.2 Main Contribution

Conventional area optimization algorithms based on resource sharing typically merges graph representations of two or more custom instructions that contain similar subgraphs. Our results reveal that resource sharing based approaches (e.g. [21]-[27]) often do not lead to the most

efficient FPGA resource utilization and can result in high critical path delay.

This paper presents a novel strategy for generating area-time efficient FPGA realization of custom instructions on commercial architectures. The proposed strategy first maps the custom-instructions onto the logic blocks to maximize the area utilization of FPGA resources. This process resembles technology mapping, and it is treated as a covering problem rather than a merging problem. The mapped custom instructions are then merged to maximize the utilization of the logic blocks prior to judiciously considering resource sharing to avoid increasing the area-delay product. The proposed strategy can be completed in the order of milliseconds and hence does not incur an overhead in existing design flows.

2 Overview of Proposed Method

The proposed method consists of the following three main steps: 1) cluster identification, 2) cluster merging, and 3) data-path combination and resource sharing. We describe the proposed method by using the example in Figure 1.

Definition 1: A custom instruction data-path is a directed graph $G_i = (V_i, E_i)$ for $i = 1,2,\ldots,n$ and n is the number of custom instructions obtained using the methodology in [28], where:

- A vertex $v \in V_i$ for $1 \le i \le n$ is a primitive integer operation in a compiler's IR (Intermediate Representation). These operations can be categorized as 1) arithmetic i.e. addition (*add*), subtraction (*sub*), multiplication (*mul*), 2) logical (*and*, *or*, *xor*), and 3) relational e.g. logical/arithmetic shift by a constant/non-constant factor (*shl*, *shr*, *shra*). Each vertex is also associated with at most two input ports and one output port.
- An arc $e = (u, v) \in E_i$ indicates a data transfer from vertex *u* to vertex *v*, whereby the output port of *u* is connected to one of the input ports of *v*.

Definition 2: A basic cluster $C_i^j = (V_i^j, E_i^j)$ is a subgraph of a custom instruction G_i, which can be implemented either 1) on a single *FPGA logic group* (a group of logic elements that share the same hardware configuration), or 2) using embedded FPGA IP (Intellectual Property) cores. The size of the logic group is equivalent to the custom instruction bit-width. None of the basic clusters in G_i overlap, i.e. $V_i^j \bigcap V_i^k = \varnothing$ and $E_i^j \bigcap E_i^k = \varnothing$ for $j \ne k$. In addition $\bigcup_{j=1}^{c} V_i^j = V_i$ and $\bigcup_{j=1}^{c} E_i^j = E_i$, where c is the number of basic clusters in G_i.

Cluster identification partitions the custom instruction data-paths into a set of basic clusters. For example in Figure 1(a), G_1 consists of basic clusters C_1^1, C_1^2 and C_1^3, and G_2 consists of basic clusters C_2^4 and C_2^5. Cluster identification resembles the technology mapping process, where a set of basic clusters that effectively covers the custom instruction data-path is identified. It is worth mentioning that unlike existing works in technology mapping, cluster identification operates on the high-level representation of the custom instructions.

Definition 3: Let $C_x^j = (V_x^j, E_x^j)$ and $C_y^k = (V_y^k, E_y^k)$ be basic clusters. $\overline{G} = (\overline{V}, \overline{E})$ is known as the *merged cluster* of C_x^j and C_y^k, denoted as $\overline{G} = C_x^j \oplus C_y^k$, if and only if:

- $x \ne y$, i.e. only the basic clusters in different data-paths can be merged.
- The merged cluster $\overline{G}$ can be implemented on a single FPGA logic group.
- $\overline{V} = V_x^j \bigcup V_y^k \bigcup V^o$, where V^o is a set of extra vertices and $0 \le |V^o| \le 1$. The extra vertex consist of a *m*-1 multiplexer (*mux*) to facilitate time-multiplexed computations of the basic clusters C_x^j and C_y^k. For example in Figure 1(c), the basic clusters C_1^1 and C_2^4, and the basic clusters C_1^2 and C_2^5 are merged to produce the merged clusters $C_1^1 \oplus C_2^4$ and $C_1^2 \oplus C_2^5$ respectively. The resulting merged clusters require an additional component, which is a 2-1 multiplexer (*m* = 2).
- $\overline{E} = E_x^j \bigcup E_y^k \bigcup E^o$, where E^o is a set of extra arcs and $0 \le |E^o| \le 1$. The extra arcs are introduced along with the *mux*. For example in Figure 1(c), the introduction of a 2-1 *mux* in the merged cluster $C_1^1 \oplus C_2^4$ has resulted in an additional arc from the output port of the *mux* to the input port of *add*. Note that the output ports of vertex $u \in V_x^j$ and $v \in V_y^k$ are assigned to the input ports of the *mux*. In addition, a select signal *Sel* is required for the *mux*. The same can be observed for $C_1^2 \oplus C_2^5$.

Cluster merging first identifies all combinations for merging the basic clusters in order to further increase the utilization of the FPGA resources by maximizing the functionality of the logic groups. Basic clusters can be merged only if they belong to different data-paths and the resulting merged cluster can be implemented onto a single FPGA logic group. Note that the generation of a merged cluster may introduce an additional input for the multiplexer select pin as shown in Figure 1(c). This poses a limitation to the combination of basic clusters that can be merged, as the number of inputs of the resulting merged cluster cannot violate the input constraint of the logic group. In cases where a multiplexer is introduced in a merged cluster, the multiplexer select signal is used to select the desired functionality between the corresponding basic clusters in a time-multiplexed manner. This is possible as the basic clusters that are associated with the merged clusters do not execute concurrently as they belong to different data-paths. In order to ensure that each basic cluster can only be merged in a unique fashion, a

heuristic is used to select a unique set of merged clusters with the aim to maximize the area utilization of the FPGA resources.

In the final stage (*data-path combination and resource sharing*), the basic clusters in the custom instruction data-paths are replaced with the selected merged clusters. Custom instruction data-paths that incorporate common merged clusters are then combined to construct a final data-path such as that shown in Figure 1(c). Note that multiplexers may be inserted to facilitate interconnect sharing between the clusters. In the example, the resulting data-path consists of a set of merged clusters (i.e. $C_1^1 \oplus C_2^4$ and $C_1^2 \oplus C_2^5$) and one basic cluster (i.e. C_1^3). In order to further minimize the area costs, we can perform resource sharing on the clusters of separate data-paths (data-paths that have not been combined) only if the resulting data-path does not lead to higher area-delay product. Since the example in Figure 1(c) has only one single resulting data-path, resource sharing is not performed.

It can be observed that the resulting data-path in Figure 1(c) contains equivalent number of operations and interconnections as the original data-paths in Figure 1(a). Hence, in contrast to the resource sharing based method, our approach may not lead to lesser number of operations and interconnections. However, it can be seen in Figure 1(c) that the actual FPGA implementation results of our approach has higher area reduction when compared to the resource sharing based approach (i.e. Figure 1(b)). In addition, our approach leads to a lower critical path delay.

The proposed method only attempts to merge basic clusters that can be mapped uniformly onto logic elements to form logic groups that share the same hardware configuration. This is applicable for most of the operations in our experiments. Operations, which cannot be mapped uniformly onto the logic groups, form basic clusters that will not be considered for merging. Note that these operations are usually realized using embedded FPGA IP cores. The current method also does not consider optimized circuits (e.g. carry-select adder, compressor tree, etc.), or specialized operations (e.g. counters) that exploits the carry-chain and multiplexers within the logic elements for efficient realization. These circuits/operations cannot be directly identified from the high-level representation of the applications [29] and, therefore, are not considered in the proposed method.

3 Cluster Merging for Area Optimization

In this section, we begin by briefly describing the cluster identification stage. Details of this work can be found in [28]. In this paper, we focus on cluster merging. The resource sharing algorithm employed in the last stage of the proposed method is adopted from [27].

3.1 Cluster Identification

Cluster identification incorporates the following steps: 1) cluster enumeration and 2) cluster selection. The *cluster enumeration* process decomposes the template into a list of basic cluster instances. Two sets of legality checks have been used to determine a basic cluster. The legality checks have been formulated based on our understanding on how the operations are mapped onto the logic elements of the target FPGA.

The first set of legality checks determines the primitive operations that can be included in the basic cluster. For example, only a single arithmetic operation is allowed in a basic cluster. A logical operation must execute before an *add/sub* operation in a basic cluster but certain relational operations (i.e. shift-by-constant) can execute after the *add/sub* operation in the basic cluster. More complex arithmetic operations such as multiplications cannot co-exist in a basic cluster with other operations.

The second set of legality checks evaluates whether the operations that have been included in a basic cluster conform to the input-output constraints of the FPGA logic element. For example, the number of inputs of the operations cannot exceed K and the number of outputs is 1. This legality check also takes into consideration the input constraints for implementing addition/subtraction using logic elements in certain FPGA technology (see [28]).

After all the basic clusters have been enumerated, a set of basic clusters is then selected in the *cluster selection* process to effectively cover the original data-path. The basic clusters can be categorized into three cluster groups as shown in Figure 2. The *comb* group (Figure 2(a)) consists of only logical and/or relational operations (i.e. *shl*, *shra*). The *ari* group (Figure 2(b)) consists of an *add/sub* operation that may co-exist with a relational operation. Relational operations in the *ari* group (i.e. *shl*, *shra*) must execute after the *add/sub* operation (refer to the legality check for including an operation in the cluster). Finally, the third group consists of a combination of the *comb* and an *ari* group (Figure 2(c)). Operations in the *comb* group must execute before the operations in *ari*.

3.2 Cluster Merging

Figure 3 shows the algorithm for cluster merging. Initially, the basic clusters that are obtained from the cluster identification stage are stored in both Cs_i and Cs_0 (line 1). In the first iteration (lines 4-19), each pair of basic clusters are evaluated for cluster merging. The resulting merged clusters are stored in Cs_{i+1} (line 14). In subse-

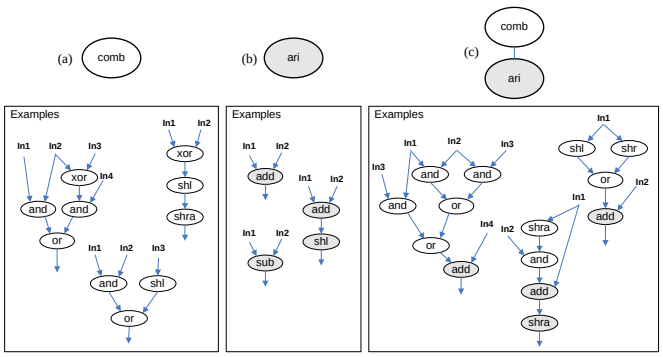


Fig. 2. Cluster groups and their corresponding examples

Algorithm 1: Cluster Merging

Input: Set of basic clusters from the cluster identification stage Cs_0
Output: Unique set of merged clusters

```
1   Cs_1 = Cs_0
2   i = 1;
3   merge_cluster_pair_found := true;
4   while merge_cluster_pair_found = true
5       merge_cluster_pair_found := false;
6       for all clusters C_x^i, where C_x^i ∈ Cs_i
7           for all clusters C_y^0, where C_y^0 ∈ Cs_0
8               success := MergeClustersWithoutMux(C_x^i, C_y^0);
9               if success = false then
10                  success := MergeClustersWithMux(C_x^i, C_y^0);
11              end
12              if success = true then
13                  merge_cluster_pair_found := true;
14                  Cs_{i+1} = StoreMergeCluster(C_x^i, C_y^0);
15                  i++;
16              end
17          end
18      end
19  end
20  solution := SelectMergeClusters(Cs_i, Cs_{i-1}, ..., Cs_0);
21  return solution;
```

Fig. 3. Cluster merging algorithm

quent iterations, the newly computed merged clusters from the previous iteration (i.e. Cs_i) are evaluated for cluster merging with the original set of basic clusters (i.e. Cs_0). This iterative process is repeated until no merged clusters are found in a particular iteration (i.e. condition expression equates to *false* in line 4). In the second part of the algorithm, a unique set of merged clusters is selected from the merged cluster sets (i.e. from the sets $Cs_i, \ldots, Cs_1$) and the original set of basic clusters (i.e. Cs_0) (line 20).

There are two ways for merging a pair of clusters: with or without introducing a multiplexer. The pair of clusters is first evaluated to determine if they can be merged without incurring a multiplexer (line 8). If this is not possible, then the cluster pair is evaluated to determine if they can be merged by introducing a multiplexer (line 10). Preference is given to the former as introducing a multiplexer requires additional input pins (i.e. the multiplexer select pin) and this may violate the input constraints of the cluster. In the following sections, we will discuss these two cluster merging methods.

1) Merging Clusters Without Multiplexer

Figure 4 describes the algorithm for determining if a cluster pair can be merged without introducing a multiplexer. As mentioned earlier, a pair of clusters can be merged if the resulting merged cluster can still be implemented using a single logic group. The first step of the algorithm partitions the cluster pair into their corresponding *comb* and *ari* components (lines 2-3).

Figure 5 shows two approaches to merge clusters without introducing a multiplexer. Let C_x be a cluster, and $comb_x$ / ari_x the corresponding *comb*/*ari* components. In Figure 5(a), $comb_x$ consists of a multiplexer that has been introduced in previous iterations. Note that there is an external input to the multiplexer (i.e. In_1) that is not connected to any internal logic within $comb_x$. We denote such an input as an *unused-mux-input*. For simplicity, we have omitted all inputs/outputs of the clusters that are irrelevant to the current discussion. Cluster merging of C_x and C_y is achieved by connecting the output port of $comb_y$ to *unused-mux-input*. The second approach to merge C_x and C_y is shown in Figure 5(b). In this scenario, C_x does not have a *comb* component and C_y does not have an *ari* component. Cluster merging is achieved by simply connecting the output port of $comb_y$ to the input port of ari_x.

The cluster merging approaches discussed above can only be achieved when certain conditions are satisfied. These four conditions, which are evaluated in lines 4-21 of Algorithm 2, are described below. When the necessary Conditions 1 and 2 are met, Conditions 3 and 4 form the sufficient conditions to merge a pair of clusters. Note that

Algorithm 2: Merge Clusters Without Mux

Input: Cluster pair to be evaluated: C_x, C_y
LUT input constraint: K
Output: Outcome of evaluation (success or fail)

```
1   success := true;
2   (comb_x, ari_x) := partition_cluster(C_x)
3   (comb_y, ari_y) := partition_cluster(C_y)
4   if (ari_x ≠ φ) and (ari_y ≠ φ)
5       If (ari_x ≠ ari_y)
6           success := false;
7       end
8   end
9   if (ari_x ≠ φ) and (ari_y = φ)
10      If ShiftByConstantExist(ari_x)
11          success := false;
12      end
13  end
14  if success = true and (C_x ≠ C_y)
15      If (comb_x ≠ φ) and (comb_y ≠ φ)
16          success := UnusedMuxPinExist (comb_x)
17      end
18      If (ari_x ≠ φ) and (comb_y ≠ φ)
19          success := IsIdentity (comb_y)
20      end
21  end
22  if InputConstraintViolated (K, comb_x, ari_x, comb_v, ari_v)
23      success := false;
24  end
25  return success;
```

Fig. 4. Algorithm to evaluate if a cluster pair can be merged without a multiplexer

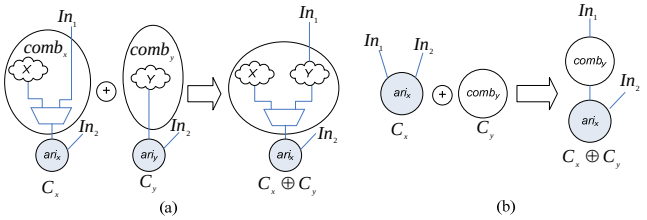


Fig. 5. Two approaches for merging clusters without introducing an additional multiplexer

the conditions for merging clusters C_x and C_y are commutative.

- **Necessary Condition 1**: *If C_x and C_y consist of ari components (e.g. Figure 5(a)) and they can be merged, then $ari_x = ari_y$.* **Proof**: We proof this by contradiction. Suppose that C_x and C_y can be merged and they both consist of different *ari* components ($ari_x \neq ari_y$). The merged cluster $C_x \oplus C_y$ is implemented in a single logic group (according to Definition 3), which can perform the functionalities of C_x and C_y including the arithmetic functions ari_x and ari_y. However, this contradicts the fact that each logic group can only implement one single *ari* component (see Section 3.1). Hence, for C_x and C_y to merge, ari_x and ari_y must be the same. This condition is evaluated in lines 4-8 of Algorithm 2.
- **Necessary Condition 2**: *If only one of the clusters C_x or C_y has an ari component (e.g. Figure 5(b)) and they can be merged, then the ari component cannot have a shift-by-constant operation*. **Proof**: We proof this by contradiction. Suppose that C_x and C_y can be merged and only C_x has an *ari* component (i.e. ari_x), which includes a shift-by-constant operation. The merged cluster $C_x \oplus C_y$ is implemented in a single logic group, which can perform the functionalities of C_x and C_y. In order for the merged cluster to perform the functionalities of C_y, the *ari* component must be convertible to an identity function (for example in Figure 5(b), by assigning an identity operand to In_2 (e.g. In_2 = 0) in $C_x \oplus C_y$, we can obtain the result of $comb_y$ if ari_x can be converted to an identity function). However it is not possible to convert ari_x with a shift-by-constant operation (assuming the constant is non-zero) to an identity function as the shift operation will always alter the input value (except when the input value is zero). Hence, this contradicts the assumption that C_x and C_y can be merged. This condition is evaluated in lines 9-13 of Algorithm 2, where the ShiftByConstantExist function checks if the *ari* component has a shift-by-constant operation.
- **Sufficient Condition 3**: *If Conditions 1 and 2 are met, and if both C_x and C_y have a comb component, then cluster merging is possible if one of the clusters have an unused-mux-input* (e.g. Figure 5(a)). This condition is evaluated in lines 15-17 in Algorithm 2 using function UnusedMuxPinExist.
- **Sufficient Condition 4**: *If Conditions 1 and 2 are met, and if at least one of the clusters has a comb component (e.g. $comb_y$ in Figure 5(a) and Figure 5(b)) while the other clusters have an ari component, then cluster merging can be achieved if the comb component can be converted to an identity function.* This enables the merged cluster to perform the functionality of the *ari* component. This condition is evaluated in lines 18-20 of Algorithm 2 using the IsIdentity function.

Figure 6 provides examples to show how two clusters can be merged without introducing a multiplexer. In Figure 6(a), conditions 1, 2 and 4 are not applicable as none of the clusters have an *ari* component. The clusters C_x and C_y can be merged as they satisfy condition 3, i.e. C_x has an *unused-mux-input*. In Figure 6(b), conditions 1 and 3 are not applicable. Clusters C_x and C_y can be merged as they satisfy condition 2 and 4, i.e. the *ari* component in C_x does not have a shift-by-constant operation and the *comb* component in C_y can be converted to an identity function. Note that the latter condition allows the merged cluster to perform the arithmetic operation in C_x by assigning In_4 to '1', and In_5 to '0'. In Figure 6(c), condition 1 is not applicable. Clusters C_x and C_y satisfy conditions 2, 3 and 4. In this case, the clusters are merged by connecting the output of C_y to the *unused-mux-input* of C_x. In addition, the merged cluster $C_x \oplus C_y$ can perform the arithmetic operation in C_x by assigning In_5 to '1', and In_6 to '0'. Finally in Figure 6(d), condition 2 is not applicable. Both clusters have an *ari* component and hence they must first satisfy condition 1. In addition, they also satisfy condition 3 so that the *comb* component of C_y can be connected to the *unused-mux-input* of C_x. Finally, condition 4 is also satisfied and the merged cluster $C_x \oplus C_y$ can perform the arithmetic operation in C_x by assigning $In_2 \oplus In_6$ to '0'.

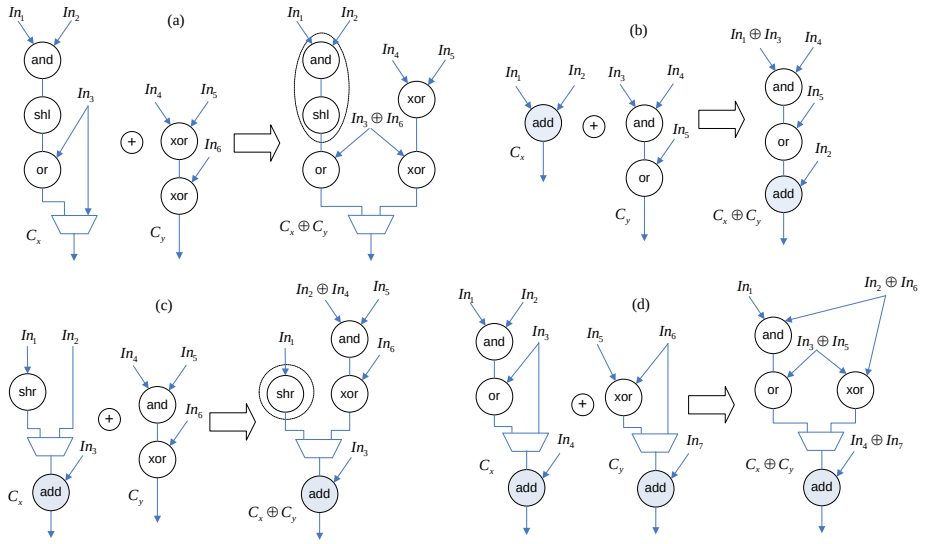


Fig. 6. Examples of cluster pairs that can be merged without a multiplexer

The evaluation of the conditions for cluster merging relies on three functions i.e. ShiftByConstantExist, UnusedMuxPinExist and IsIdentity. The first two functions are trivial as long as the program maintains a record of all the operations and the connectivity information between the operations in each cluster. Hence, we will only describe the implementation of the IsIdentity function.

Implementation of the IsIdentity Function

The IsIdentity function aims at evaluating whether the *comb* component of a cluster can produce an output that is identical to one of the inputs. In order to evaluate whether a particular input can be produced at the output, the remaining inputs are first assigned to '0's or '1's. Then the DFG of the *comb* component is progressively simplified as the inputs are propagated to the output by means of dataflow identity and dominance laws [30]. If the input is directly connected to the output when the process completes, then the *comb* component can be converted to an identity function.

The proposed approach to determine if a DFG is an identity function draws certain similarities with the method presented in [31] to transform a complex DFG pattern to a simpler pattern by applying identity operands to the nodes in order to eliminate them from the pattern. However, the method presented in [31] assumes that each node in the DFG have an external input that can be assigned to an identity operand. In addition, the method in [31] does not take into account shift-by-constant operators, which cannot be converted into an identity operation. The proposed method overcomes these limitations.

Figure 7 shows the proposed algorithms for determining if $comb_x$ can be converted to an identity function. We

Algorithm 3: IsIdentity

Input: $comb_x$
Output: Outcome of evaluation (success or fail)

1 Identify shift-by-constant operations and store them in *roots;*
2 **for** all shift operations *i* in *roots*
3 perform depth-first-search to obtain reverse sub-tree *S* rooted at *I;*
4 remove all vertices/edges $(v,e) \in S$, where $(v,e) \notin S \cap C_x$;
5 assign '0' to edges $e' = (i,v')$, where $(v',e') \in C_x - S$;
6 **end**
7 *success* := false;
8 **for** all inputs *u of* $comb_x$
9 find next input *v*, where $u \neq v$;
10 *success* := **AssignOperand** $(u,v,0,comb_x)$;
11 **if** *success* = true **then** break; **end**
12 *success* := **AssignOperand** $(u,v,1,comb_x)$;
13 **if** *success* = true **then** break; **end**
14 **end**
15 **return** *success*;

Algorithm 4: AssignOperand

Input: Input *u*, Input *v*, *op_value*, $comb_x$
Output: Outcome of evaluation (success or fail)

1 *success* := false;
2 assign *op_value* to edges that are incident to *v*;
3 find next input *w*, where $w \neq u \neq v$;
4 **if** $w = \phi$ **then**
5 **EliminateOp** $(comb_x)$
6 If input u is connected directly to output of $comb_x$ **then**
7 *success* := true;
8 **end**
9 **else**
10 *success* := **AssignOperand** $(u,w,0,comb_x)$;
11 **if** *success* = true **then** return *success*; **end**
12 *success* := **AssignOperand** $(u,w,1,comb_x)$;
13 **end**
14 **return** *success*;

Fig. 7. Algorithm to identify if a *comb* component can be converted to an identity function

will describe the algorithms based on the example in Figure 8. The first part of the algorithm (lines 1-6 of Algorithm 3) removes all the nodes in the reverse sub-trees rooted at the shift-by-constant operations (e.g. node 4 in Figure 8(a)). The function first identifies all the root nodes in $comb_x$ (line 1 of Algorithm 3) before finding the reverse sub-tree members of the corresponding root nodes using the depth-first-search algorithm (line 3 of Algorithm 3). Each sub-tree can only have one shift operation that must be a root node. Note that only the nodes that are exclusive to the sub-tree are removed (line 4 of Algorithm 3).

The rationale for removing the nodes in the reverse sub-trees rooted at the shift-by-constant operations are as follows. As discussed in [28], the inputs of these reverse sub-trees are assigned dedicated input pins (e.g. In_1^{shr} in Figure 8(b)), which are hardwired to the required shifted operands. Each sub-tree can then be treated as a hypothetical node with dedicated input pins. As the hypothetical nodes cannot be converted to an identity function (due to the fact that the shift operations will alter any non-zero input values), they are removed from $comb_x$ by assigning the dedicated input pins to '0's. This produces '0's at the output of the hypothetical nodes as shown in Figure 8(c) (line 5 of Algorithm 3).

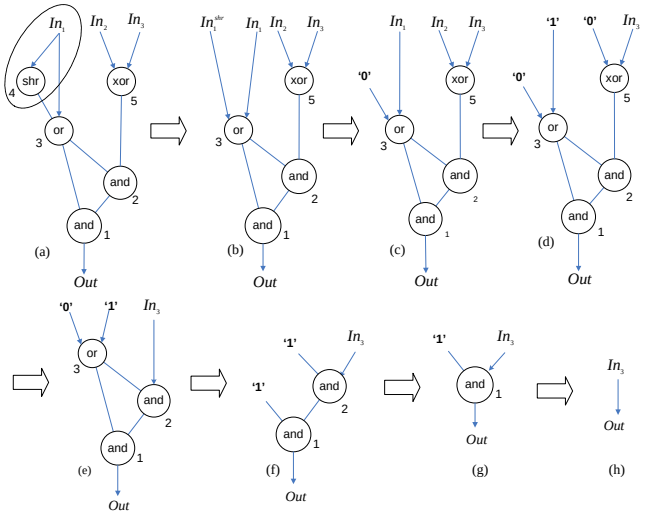


Fig. 8. Identifying if a *comb* component can be converted to an identity function

Next, the inputs of $comb_x$ are evaluated one at a time to determine if they can be produced at the output after eliminating all the operations within $comb_x$ by means of dataflow identity and dominance laws. This is achieved by recursively assigning '0's and '1's to the remaining inputs (using the AssignOperand function in lines 10 and 12 of Algorithm 3 and Algorithm 4) and propagating these values through the operators in $comb_x$ (using the function EliminateOp in line 5 of Algorithm 4). The AssignOperand function considers all the 2^n Boolean assignments of the remaining n number of external inputs to the *comb* component. For example in Figure 8(d), to evaluate whether In_3 can be produced at the output, In_1 and In_2 are assigned to '1' and '0' respectively (using the AssignOperand function). In the subsequent steps (Figure 8(e)-(h)), these values are propagated to the operations one at a time beginning from the predecessor nodes (using the EliminateOp function). As the operations in the *comb* component have been sorted according to their order of dependency, the operations are progressively removed in the order of the dependency graph by means of dataflow identity and dominance laws. At each step, the output of an operation is produced based on whether the operands are identity and dominating operands. The operation is then removed. In this example, In_3 is directly connected to the output after all the operations have been removed and, hence, $comb_x$ can be converted to an identity function.

In the worst case, the time complexity of AssignOperand for a single iteration of Algorithm 3 (lines 8-14) is $O(2^{n_{in}-1})$, where n_{in} is the number of external inputs to the *comb* component. In practice, n_{in} is usually very small (e.g. 4-6 depending on the target FPGA). The execution time complexity of EliminateOp is $O(n_{op})$, where n_{op} is the number of operations in the *comb* component. Note that the linear time complexity can be achieved as the operations in the *comb* component have been sorted according to their order of dependency.

Checking for Input Constraints Violation of Merged Cluster

As no additional multiplexer has been introduced in the merging process, the input constraints of the merged cluster will not be violated if all the input pins of the original cluster with fewer inputs can be merged. For example, in Figure 6(d), all the input pins of C_y have been merged with those of C_x. Since both the original clusters already conform to the input constraints, the number of input pins of the resulting merged cluster will also not exceed *K*. However, there is a possibility that the input pins of the original clusters cannot be merged. This is shown in Figure 6(a) and (c), whereby only a single input pin of the original clusters is merged (i.e. $In_3 \oplus In_6$ in Figure 6(a) and $In_2 \oplus In_4$ in Figure 6(c)). The reason for this is that shifted values of the inputs affected by the shift-by-constant operation (e.g. *and-shl* in Figure 6(a) and *shr* in Figure 6(c)) will be hardwired to the logic elements and, hence, cannot serve as valid inputs to other logic functions (that do not require the same shifted values). Hence, there is a need to check if the required inputs of the merged cluster exceed *K* (line 22 of Figure 4). We have employed the algorithm in [28] for checking the input constraint violation of the merged clusters.

2) Merging Clusters With Multiplexer

If the cluster pair cannot be merged with the method described in the previous section, a multiplexer will be introduced in an attempt to merge the cluster pair. Figure 9 shows the various scenarios for cluster pair C_x and C_y to be merged by introducing a multiplexer. When both the cluster pair do not consist of an *ari* component (e.g. Figure 9(a)), the multiplexer is inserted at the output of C_x and C_y. However, if at least one of the cluster pair has an *ari* component (e.g. Figure 9(b) and (c)), the multiplexer must be inserted at the input of the arithmetic operation. This is due to the fact that all logical operations (we assume that the multiplexer is a logic operator) must execute before the arithmetic operation in a cluster (see Section 3.1).

The input pins of C_x and C_y are also merged whenever possible and an additional input pin is required for the multiplexer select. The actual number of required input pins of the merged cluster candidate must be recalculated using the algorithm in [28] to verify if the input pin constraint is still met.

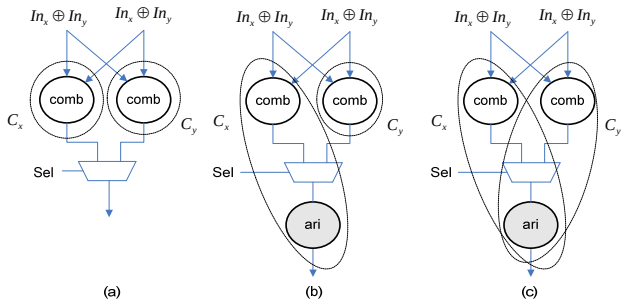


Fig. 9. Incorporating multiplexer into a merged cluster

3) Choosing Unique Sets of Merged Clusters

The cluster merging method will result in a basic cluster appearing in a number of merged clusters. In order to ensure that each basic cluster can only be merged once, there is a need to select a unique set of merged clusters from the merged cluster set.

A compatibility graph is constructed to select a unique set of merged clusters from $Cs_i, \ldots, Cs_0$ (line 20 in Figure 3). The set Cs_i for $i \neq 0$ is a set of merged clusters where each merged cluster can be formed by merging a number i of basic clusters. The compatibility graph approach is similar to that proposed in [27] for selecting a set of resources in two data-paths for merging. However, our approach differs from that in [27] as we consider the selection of basic clusters for merging in all the custom instruction data-paths (instead of two data-paths at a time as in [27]). This global selection strategy can lead to better quality results.

Definition 4: A compatibility graph is an undirected graph $G_u(V_u, E_u)$ where:

- A vertex $v \in V_u$ is a merged cluster which consists of a number i of basic clusters that can be merged to form v. A vertex v is associated with a weight $w(v) = i^2 \times freq$, where *freq* is the number of occurrences of the basic clusters in v that is found across all the custom instructions. The weight has been chosen in order to maximize cluster sharing among the most frequently occurring clusters in the custom instruction data-paths. The factor i^2 is used as the value of i is typically much lower than *freq*. Note that the value of i varies according to the number of basic clusters that are used to form the merged cluster (see line 14 of Algorithm 1).
- There is an arc $e = (u, v) \in E_u$ if the merged clusters represented by u and v are compatible.

Definition 5: Vertex u and vertex v are not compatible if a basic cluster associated with u also exists in v.

In the worst case, the number of vertices $|V_u|$ in the compatibility graph, is $\sum_{r=1}^{i} \frac{n_{bc}!}{r!(n_{bc}-r)!}$, where n_{bc} is the number of basic clusters in Cs_0. i is the number of basic clusters that are used to form a merged cluster in the corresponding iteration in the *while* loop of Algorithm 1 (lines 4-19). This equation assumes that in iteration i, all combination of i basic clusters can be merged. Hence, in each iteration of the *while* loop, the number of possible merged clusters can be calculated using the combination function $\frac{n_{bc}!}{i!(n_{bc}-i)!}$. Note that this assumption will not be the case in practice. The maximum number of compatibility graph vertices in our experiments is less than 200. In addition, the maximum value of i in our experiments is five, as it is unlikely to find more than five clusters that can be merged into a logic group. The time complexity to construct the compatibility graph is $O(|V_u|^2)$ as there is a need to check for the compatibility of each vertex with all other vertices.

In order to identify a unique set of merged clusters that would maximize the FPGA resource utilization, the maximum weight clique is heuristically computed from the compatibility graph. The following definition of maximum weight clique is obtained from [27].

Definition 6: The maximum weight clique of a graph $G_c(V_c, E_c)$ is a set of vertices $C \subseteq V_c$ where for all vertices $u, v \in C$, the arc $(u, v) \in E_c$ and $\sum_{\forall v \in C} w(v)$ is maximum.

As mentioned in [27], the execution time to compute the maximum weight clique can be polynomially bounded by $|V_u|$.

3.3 Resource Sharing of Clusters in the Combined Data-path

The basic clusters in the custom instruction data-paths are replaced with the selected merged clusters so as to de-

termine the resulting data paths. This involves combining custom instruction data-paths based on the binding of basic clusters that are associated with the unique merged cluster set. This may necessitate introducing multiplexers in order to maintain the correctness of data-path of the associated custom instruction.

In order to further minimize the area costs without incurring additional area-time overhead, resource sharing is performed on the basic clusters and merged clusters that do not reside on the same data-path. A cluster can only be considered for resource sharing only once to avoid increasing the critical path delay. In addition, if resource sharing between the clusters leads to more inferior area-time results, the solution prior to resource sharing is adopted. In this work, we have employed the resource sharing method that was presented in [27] for merging a pair of clusters. It is noteworthy that in most of the experiments considered, the proposed approach leads to very little opportunity for resource sharing.

4 Experimental Results

In this section, we compare the results of the proposed approach with results from 1) a commercial FPGA implementation tool [32] that is targeted for area optimization with resource sharing option selected [20], and 2) one of the best known methods for resource sharing [27].

We have used eight applications from the MiBench embedded benchmark [33] and MediaBench benchmark [34] suites. Only integer operations are allowed in custom instructions and the maximum number of inputs/outputs for the custom instructions is 5/2. Previous work has shown that inputs/outputs more than this range results in little performance gain [35]. It is noteworthy that although larger custom instructions can be obtained by preprocessing the IR with certain advanced optimization passes, such approaches are only viable if the resulting instructions can be supported by the target processor. It is worth mentioning that the proposed method in this paper is not restricted to the number of input/output constraints of custom instructions. The outputs of custom instruction data-paths of all three approaches are multiplexed to meet the two-output port constraint of the RFU. In particular, single output custom instructions are multiplexed to the primary output port of the RFU, and dual-output custom instructions are multiplexed to the primary and secondary output ports of the RFU.

Table 1 reports the total number of operations in the resulting data-paths that are generated using the various methods. The *Original* approach is based on custom instruction data-paths that have been obtained from [28] without further optimization. The *Resource Sharing* approach is based on the method presented in [27] to obtain an optimized data-path that maximizes the resource sharing of the original data-paths. The approach in [27] performs data-path merging on two data-paths at a time until all the data-paths have been considered. For the *Proposed Method*, the original custom instruction data-paths are subjected to the methods presented in this paper. Note that the data-paths in the three approaches will be subjected to further optimization during implementation with the FPGA tool.

In Table 1, the optimized data-paths that are generated using *Proposed Method* are based on K = 4. The average number of operations per instruction in each application for *Original* ranges from 3 to 7. When compared to results of *Original*, the *Resource Sharing* approach leads to an average reduction of over 29% in the number of operations. In contrast, *Proposed Method* has an average percentage reduction in the number of operations of less than 7%. These results confirm that unlike the resource sharing approach, the proposed method does not aim at maximizing resource sharing between the data-paths.

TABLE 1
Number of Operations

Applications	Number of Custom Instructions	Original	Resource Sharing	Proposed Method
Adpcm Dec	6	16	12	14
Adpcm Enc	7	19	16	22
Bitcount	4	27	25	25
Blowfish	8	25	16	23
Pegwit	18	65	40	60
Sha	15	51	26	36
Rijndael Dec	13	35	21	29
Rijndael Enc	13	38	29	44

The optimized data-paths for each of these methods have been designed in VHDL, implemented using Xilinx ISE (version 9.1.01i) [32] and targeted on three state-of-the-art FPGA architectures, i.e. Spartan-3 (xc3s5000fg1156-4) [8], Virtex-4 (xc4vlx200ff1513-10) [9] and Virtex-5 (xc5vlx50ff1153-1) devices [10].

The Spartan-3 and Virtex-4 devices incorporate logic elements with 4-input LUTs. The Virtex-5 devices incorporate logic elements with 6-input LUTs that can be used to implement any 6-input function or two dual-output 5-input functions [36]. For the Spartan-3 and Virtex-4 solutions, the proposed method generates optimized data-paths for K = 4 such that all the clusters produced cannot have more than 4 inputs. For the Virtex-5 solution, two sets of results for K = 5 and K = 6 are first generated. The set which leads to the best area-time results is chosen. The data-paths produced using the various methods are implemented with the FPGA tool under the same design constraints and optimization options. In particular, we have enabled the implementation options for area optimization and resource sharing.

TABLE 2
AREA (NUMBER OF SLICE LUTS)

Applications	Spartan-3			Virtex-4			Virtex-5		
	Original	Resource Sharing	Proposed Method	Original	Resource Sharing	Proposed Method	Original	Resource Sharing	Proposed Method
Adpcm Dec	157	113	96 (15.0%)	156	113	96 (15.0%)	219	151	129 (14.6%)
Adpcm Enc	171	212	137 (35.4%)	171	211	139 (34.1%)	268	253	180 (28.9%)
Bitcount	160	176	117 (33.5%)	160	176	116 (34.1%)	324	254	252 (0.8%)
Blowfish	261	197	158 (19.8%)	261	194	158 (18.6%)	419	247	225 (8.9%)
Pegwit	591	466	391 (16.1%)	590	466	391 (16.1%)	904	540	493 (8.7%)
Sha	430	348	288 (17.2%)	428	348	287 (17.5%)	628	384	350 (8.9%)
Rijndael Dec	356	226	247 (-9.3%)	355	226	247 (-9.3%)	483	290	294 (-1.4%)
Rijndael Enc	1476	847	711 (16.1%)	1476	844	709 (16.0%)	3492	1745	1110 (36.4%)

4.1 Area Measures

Table 2 shows the optimized area for the various approaches. In order to compare the area utilization using a common measure (i.e. number of slice LUTs), we have disabled the option to map multiplication operations onto embedded multipliers and DSP blocks in the FPGA. The percentage values (in brackets) are the percentage area reduction of *Proposed Method* over *Resource Sharing*. It is worth mentioning that the original data-paths have also undergone area optimizations (with resource sharing as one of the optimization strategy) that are provided by the commercial tool.

It can be observed that *Resource Sharing* and *Proposed Method* both lead to notable area reduction when compared to *Original*. The average area reduction of *Resource Sharing* over *Original* is 17.2%, 17.3% and 33.6% on Spartan-3, Virtex-4 and Virtex-5 respectively. For certain applications (e.g. Adpcm Enc and Bitcount) on Spartan-3 and Virtex-4, *Resource Sharing* results in lower area efficiency than *Original*. This is due to the fact that custom instructions in these two applications have little opportunity for resource sharing and hence, the area of the multiplexers introduced through the limited sharing of resources outweighs the area savings. In comparison, *Proposed Method* is capable of achieving higher area efficiency over *Original* in all cases. The average area reduction of *Proposed Method* over *Original* is 34.3%, 34.2% and 42.4% on the Spartan-3, Virtex-4 and Virtex-5 respectively. These results demonstrate that unlike resource sharing methods, the proposed method is still favorable for merging data-paths which do not have a high degree of similarity in the operations. It can be observed that the area reduction in both methods for Spartan-3 and Virtex-4 is comparable due to the similar characteristics of the logic elements in both architectures. The higher area reduction for Virtex-5 is due to the larger LUTs in the architecture that enables more operations to be mapped onto a logic element. In the proposed method, the higher number of input pins allowable in a logic group for the Virtex-5 architecture also enables more clusters to be merged.

Proposed Method results in higher area efficiency than *Resource Sharing* in almost all cases. The only case where *Resource Sharing* leads to notably higher area efficiency than *Proposed Method* is for application Rijndael Dec on Spartan-3 and Virtex-4. Note that the percentage area difference for this case is lesser than the percentage area reduction of *Proposed Method* in all the other applications on the Spartan-3 and Virtex-4 device. The custom instructions in Rijndael Dec have many similar operations and hence the resource sharing method is more favorable in terms of area minimization. However as shown in the next sub-section, the area optimization of *Resource Sharing* is achieved at the cost of incurring large critical path delays. When compared to one of the best known resource sharing based method, the proposed method can achieve an average area reduction of 18%, 17.8% and 13.2% on the Spartan-3, Virtex-4 and Virtex-5 respectively. These results demonstrate that an approach that maximizes resource sharing may not be the best method for FPGA area optimization.

4.2 Critical Path Delay

While resource sharing based approaches can lead to area savings, they may incur undesirable delay in the data-paths due to the extensive introduction of multiplexers whenever operations are shared across data-paths. In contrast, the proposed method judiciously introduces multiplexers at a coarser-grain, i.e. for interconnect sharing when the clusters in different data-paths are merged. In addition, the proposed method attempts to maximize the logic utilization of the FPGA logic elements, which can lead to lesser critical path delay in the data-paths. Table 3 shows the critical path delay of the optimized data-paths. The percentage values (in brackets) are the percentage delay reduction of *Proposed Method* over *Resource Sharing*. The bracket below each critical path value shows the logic and route delay that constitute the critical path.

It can be observed that *Resource Sharing* leads to highest critical path delay in almost all cases. In Virtex-5, *Proposed Method* has a critical path delay that is marginally higher than *Resource Sharing* for only one application, i.e. Rijndael Enc (difference of less than 1ns). When compared to the implementation results of *Original*, *Resource Sharing* has an average increase in critical path delay of 20.0%, 28.8% and 18.9% on the Spartan-3, Virtex-4 and Virtex-5 respectively. In comparison, *Proposed Method* has an average increase in critical path delay over *Original* of only 4.3%, 2.2% and 11.3% on the Spartan-3, Virtex-4 and Virtex-5 respectively. In certain applications on the three FPGA devices, it can be observed that *Proposed Method* can lead to lower critical path delay than the original

TABLE 3
CRITICAL PATH (NS)

Applications	Spartan-3				Virtex-4				Virtex-5			
	Original	Resource Sharing	Proposed Method		Original	Resource Sharing	Proposed Method		Original	Resource Sharing	Proposed Method	
Adpcm Dec	28.1 (10.2, 17.9)	34.9 (12.0, 23.0)	26.9 (10.3, 16.7)	(22.9%)	21.3 (5.2, 16.1)	20.1 (7.9, 12.2)	15.5 (5.9, 9.7)	(22.7%)	12.2 (5.3, 6.9)	14.1 (5.0, 9.2)	12.2 (4.7, 7.5)	(13.5%)
Adpcm Enc	28.1 (9.5, 18.6)	38.2 (11.7, 26.6)	26.7 (11.6, 15.2)	(30.1%)	18.9 (5.3, 13.6)	26.3 (8.1, 18.2)	18.1 (7.3, 10.8)	(31.1%)	13.7 (5.0, 8.8)	14.7 (5.4, 9.3)	14.3 (5.4, 8.9)	(2.8%)
Bitcount	44.9 (18.9, 26.0)	44.8 (19.3, 25.5)	49.5 (19.1, 30.5)	(-10.6%)	29.5 (11.8, 17.6)	35.0 (11.9, 23.1)	27.7 (11.3, 16.4)	(20.6%)	21.8 (9.0, 12.8)	25.3 (8.8, 16.5)	22.5 (8.6, 13.9)	(11.3%)
Blowfish	30.9 (12.7, 18.2)	40.7 (15.2, 25.5)	33.9 (14.7, 19.1)	(16.7%)	20.4 (7.9, 12.5)	26.9 (9.6, 17.3)	21.6 (9.2, 12.5)	(19.5%)	14.5 (5.8, 8.7)	18.3 (6.5, 11.8)	18.1 (7.1, 11.0)	(1.4%)
Pegwit	43.8 (18.7, 25.0)	50.0 (23.6, 26.4)	49.5 (21.6, 27.9)	(1.0%)	27.4 (12.8, 14.6)	36.3 (13.8, 22.5)	34.3 (12.7, 21.7)	(5.6%)	23.8 (8.8, 14.9)	29.4 (9.1, 20.3)	29.1 (9.1, 19.9)	(1.2%)
Sha	33.8 (13.2, 20.6)	44.5 (17.4, 27.1)	35.9 (16.0, 19.9)	(19.4%)	21.3 (8.8, 12.5)	31.6 (10.3, 21.3)	24.2 (10.1, 14.0)	(23.5%)	16.8 (6.0, 10.8)	22.8 (6.7, 16.1)	21.1 (6.9, 14.3)	(7.3%)
Rijndael Dec	35.1 (14.0, 21.0)	42.8 (17.3, 25.6)	37.0 (17.1, 19.9)	(13.7%)	20.8 (8.8, 12.0)	28.9 (11.2, 17.7)	23.9 (10.6, 13.3)	(17.4%)	15.9 (5.5, 10.3)	19.3 (7.7, 11.6)	16.2 (6.3, 9.9)	(15.7%)
Rijndael Enc	50.0 (23.1, 26.9)	49.8 (22.5, 27.2)	49.1 (22.5, 26.5)	(1.3%)	33.8 (12.6, 21.2)	42.9 (13.0, 29.8)	32.1 (13.8, 18.2)	(25.2%)	28.0 (9.4, 18.6)	29.3 (7.9, 21.4)	30.1 (9.8, 20.4)	(-2.8%)

T_{cp} (T_l, T_r)

T_{cp} = critical path delay
T_l = logic delay
T_r = route delay
$T_{cp} = T_l + T_r$

data-paths. This is contributed by two reasons. Firstly, in certain applications, the combination of data-paths due to cluster merging in the proposed method has resulted in less complex output multiplexer. Secondly, in cases where the *Proposed Method* has lower critical path delay than the original data-paths, it can be observed that the proposed method has led to a notable reduction in the routing delay. This is due to the efficient packing of operations within the logic blocks in the proposed method that has enabled the implementation tool to perform tighter placement which, in turn, leads to more effective routing.

When compared to *Resource Sharing*, *Proposed Method* has an average critical path delay reduction of 11.8%, 20.7% and 6.3% on the Spartan-3, Virtex-4 and Virtex-5 respectively. It can be observed that when compared to the original data-paths, *Proposed Method* generally leads to a lower increment in the routing delay than *Resource Sharing*. This is due to the more compact designs generated by the proposed method. In particular, when compared to *Original*, *Resource Sharing* has an average increase in routing delay of 21.1%, 36.7% and 26.9% on the Spartan-3, Virtex-4 and Virtex-5 respectively. In contrast, *Proposed Method* leads to an average increase in routing delay of less than 1% on the Spartan-3 and Virtex-4, and only 14.5% on Virtex-5. This confirms that the proposed method is not only capable of achieving a higher degree of area optimization when compared to resource sharing based approaches, but it can also lead to lesser critical path delay.

4.3 Area-Time Measures

The authors in [37] have observed that FPGA LUT size is the most useful architectural parameter for making area-delay trade-offs. Their analysis is consistent with results in this paper, which shows that the implementations on the Virtex-5 architecture generally leads to higher area but lower delay when compared to implementations on Spartan-3 and Virtex-4 architectures. In addition, the work in [37] concludes that for minimum area-delay product, a FPGA LUT size of 4 (e.g. Virtex-4 devices) provides the best results. In order to compare the overall benefits of the proposed method, we use the metric area-delay product which is obtained by multiplying the area (in terms of number of slice LUTs) with the critical path delay (in terms of nanoseconds). Table 4 shows the area-delay product of the optimized data-paths. The percentage values (in brackets) are the percentage area-delay product reduction of *Proposed Method* over *Resource Sharing*.

Table 4 confirms that *Proposed Method* has lower area-delay product than *Original* and *Resource Sharing* in all cases. It can also be observed that in certain applications, the implementation of original data-paths using the optimization provided by the commercial tool can lead to lower area-delay product than the resource sharing ap-

TABLE 4
AREA-DELAY PRODUCT

Applications	Spartan-3				Virtex-4				Virtex-5			
	Original	Resource Sharing	Proposed Method		Original	Resource Sharing	Proposed Method		Original	Resource Sharing	Proposed Method	
Adpcm Dec	4406.8	3945.4	2582.9	(34.5%)	3323.6	2271.8	1491.4	(34.4%)	2667.9	2134.1	1577.0	(26.1%)
Adpcm Enc	4800.1	8107.7	3663.4	(54.8%)	3229.3	5544.2	2515.3	(54.6%)	3678.3	3719.9	2573.1	(30.8%)
Bitcount	7187.4	7887.6	5797.1	(26.5%)	4712.2	6152.6	3218.4	(47.7%)	7071.6	6427.0	5657.9	(12.0%)
Blowfish	8055.0	8010.2	5350.4	(33.2%)	5327.5	5209.9	3417.4	(34.4%)	6062.5	4524.8	4063.3	(10.2%)
Pegwit	25863.3	23296.3	19357.6	(16.9%)	16151.8	16933.0	13416.4	(20.8%)	21480.8	15879.8	14325.1	(9.8%)
Sha	14526.3	15499.2	10336.0	(33.3%)	9097.6	10984.6	6934.2	(36.9%)	10559.8	8750.2	7396.2	(15.5%)
Rijndael Dec	12489.5	9679.8	9130.1	(5.7%)	7382.6	6532.3	5895.4	(9.8%)	7660.9	5584.5	4771.0	(14.6%)
Rijndael Enc	73726.2	42139.9	34897.3	(17.2%)	49879.9	36166.2	22727.7	(37.2%)	97793.5	51116.3	33431.0	(34.6%)

proach. In particular when compared to *Original*, *Proposed Method* has an average area-delay product reduction of 31.4%, 32.5% and 36.3% on the Spartan-3, Virtex-4 and Virtex-5 respectively. When compared to *Resource Sharing*, *Proposed Method* has an average area-delay product reduction of 27.8%, 34.5% and 19.2% on the Spartan-3, Virtex-4 and Virtex-5 respectively. These results reinforce the benefits of the proposed method for area-time optimization on FPGA.

4.4 Execution Time

The execution time of the proposed method is longer than that required by the resource sharing approach to generate the optimized data-paths. However, both methods (resource sharing and proposed) can be executed in the order of milliseconds on a HP Workstation with two 2.66GHz processors and 2GB RAM. This is an insignificant overhead when compared to the time taken for the commercial FPGA tool to implement the optimized data-paths, which is typically in the order of seconds/minutes.

4.5 Performance Gain Over Base Processor

We have performed cycle-accurate simulations for several frequently executed functions in the benchmark applications using the SimpleScalar toolset [38], in order to evaluate the performance gain of the three approaches (*Original*, *Resource Sharing* and *Proposed Method*) over the base processor. Since the methodology for custom instruction generation in [28] relies on the Trimaran compiler [39] infrastructure, the Trimaran's IR is first converted to ARM assembly code. The custom instructions are then manually inserted into the assembly code. This process requires code motion and register allocation in order to maintain the correctness of the program. Finally, the ARM assembler is used to generate the binaries for simulation. The processor configuration in SimpleScalar is set as follows: pipeline depth = 5; cache size = 64KByte; line size = 8 byte; multiplication cycles = 3 and division cycles = 20. The remaining configurations are similar to the synthesizable OpenRiSC soft-core processor configuration in [40]. All division operations are handled in software. Multi-cycle custom instructions implementation is enabled and the critical path values for the three approaches are obtained from Table 3 (Virtex-4 device).

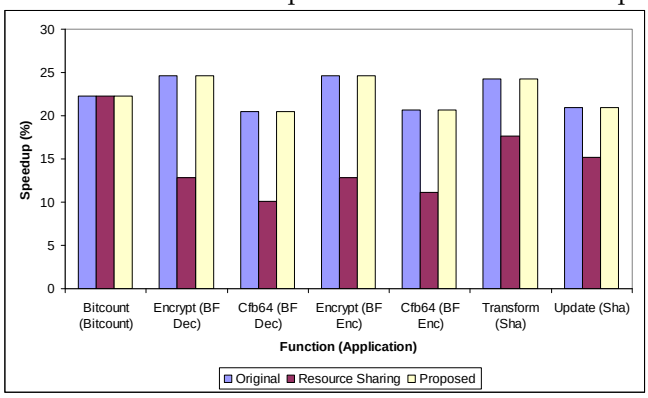


Fig. 10. Performance gain of custom processor over base processor

Figure 10 shows the percentage performance gain of the three approaches over the base processor implementation. It can be observed that *Proposed Method* and *Original* outperforms *Resource Sharing* in most of the functions considered (except Bitcount) due to the lower critical path delays of their custom instructions. It is noteworthy that the proposed method has an area reduction of more than 34% when compared to *Resource Sharing* for Bitcount (see Table 2). As shown in Figure 10, the proposed method can still achieve an average performance gain of over 22% over the base processor, while *Resource Sharing* can achieve an average performance gain of less than 15% over the base processor. Although the performance gain of *Original* and *Proposed Method* is comparable as their critical path delays are very close, it can be observed from Table 2 that *Proposed Method* has an average area reduction of more than 34% compared to *Original*.

5 Conclusion

In this paper, we have proposed a novel technique to realize area-time efficient custom instructions on commercial FPGA architectures. It leverages on our existing technique to partition the custom instructions into a set of basic clusters such that the basic clusters can be efficiently mapped onto the LUT and carry-look-ahead structure of the FPGA logic blocks. We have proposed conditions to aid the merging of basic clusters without introducing multiplexers. We have employed a heuristic based on the degree of cluster merging and the frequency of occurrences of the basic clusters to accelerate the selection of a unique set of merged clusters. Resource sharing is then performed on clusters only if the resulting data-paths do not lead to an increase in the area-delay product. When compared to the area optimization capabilities of the commercial tool and to one of the most efficient methods reported in the literature for resource sharing, the proposed method can achieve significantly lower area-delay product for all cases considered in this study due to its architecture-aware cluster merging strategy to maximize utilization of FPGA logic blocks.

References

[1] P. Garcia, K. Compton, M. Schulte, E. Blem and W. Fu, "An Overview of Reconfigurable Hardware in Embedded Systems", *EURASIP Journal on Embedded Systems*, Vol. 2006, pp. 1–19

[2] P. Lysaght and P.A. Subrahmanyam, "Guest Editors' Introduction: Advances in Configurable Computing", *IEEE Design & Test of Computers*, Vol. 22, No. 2, March-April 2005, pp. 85-89

[3] Altera: NIOS II Processors. Available: http://www.altera.com/products-/ip/processors/nios2/ni2-index.html

[4] J. Gould, "Designing Flexible, High Performance Embedded Systems", Xcell Journal, No. 58, 2006, pp. 66-69

[5] Stretch Processors. Available: http://www.stretchinc.com/

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

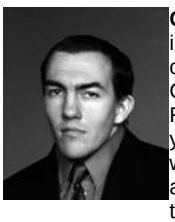
[7] W. Marx and V. Aggarwal, "FPGAs Are Everywhere – In Design, Test & Control", RTC Magazine, April 2008. Available: http://rtcmagazine.com/articles/view/100953

[8] Xilinx User Guide, "Spartan-3 Generation FPGA User Guide", UG331, Version 1.4, June 2008

[9] Xilinx User Guide, "Virtex-4 User Guide", UG270, Version 2.5, June 2008
[10] Xilinx User Guide, "Virtex-5 User Guide", UG190, Version 3.0, February 2007
[11] R. Kastner, A. Kaplan, S.O. Memik and E. Bozorgzadeh, "Instruction Generation for Hybrid Reconfigurable Systems", *ACM Transactions on Design Automation of Embedded Systems*, Vol. 7, No. 4, October 2002, pp. 605-627
[12] N.T. Clark, H. Zhong and S.A. Mahlke, "Automated Custom Instruction Generation for Domain-Specific Processor Acceleration", *IEEE Transactions on Computers*, Vol. 54, No. 10, October 2005, pp. 1258-1270
[13] F. Sun, S. Ravi, A. Raghunathan and N.K. Jha, "Custom-Instruction Synthesis for Extensible-Processor Platforms", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 23, No. 2, February 2004, pp. 216-228
[14] K. Atasu, L. Pozzi and P. Ienne, "Automatic Application-Specific Instruction-Set Extensions Under Microarchitectural Constraints", *Proceedings of the 40th IEEE/ACM Design Automation Conference*, June 2003, pp. 256-261
[15] P. Yu and T. Mitra, "Scalable Custom Instructions Identification for Instruction-Set Extensible Processors", *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, September 2004, pp. 69-78
[16] J. Cong, Y. Fan, G. Han and Z. Zhang, "Application-Specific Instruction Generation for Configurable Processor Architectures", *Proceedings of the ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, February 2004, pp. 183-189
[17] D.D. Gajski, N.D. Dutt, C.H. Allen Wu and Y.L. Steve Lin, "High-Level Synthesis: Introduction to Chip and System Design", Kluwer Academic Publishers, 1992
[18] D. Chen and J. Cong, "DAOmap: A Depth-Optimal Area Optimization Mapping Algorithm for FPGA Designs", IEEE International Conference on Computer-Aided Design, November 2004, pp. 752-759
[19] J. Lin, D. Chen, and J. Cong, "Optimal Simultaneous Mapping and Clustering for FPGA Delay Optimization", *Proceedings of Design Automation Conference*, July 2006
[20] Xilinx, "XST User Guide", UG627, Version 11.2, June 2009
[21] N. Shirazi, W. Luk and Y.K. Peter Cheung, "Automating Production of Run-Time Reconfigurable Designs", *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998, pp. 147-156
[22] Z. Huang and S. Malik, "Managing Dynamic Reconfiguration Overhead in Systems-on-a-Chip Design using Reconfigurable Data-paths and Optimized Interconnection Networks", *Proceedings of Design Automation and Test in Europe*, 2001, pp. 735-740
[23] P. Brisk, A. Kaplan and M. Sarrafzadeh, "Area-Efficient Instruction Set Synthesis for Reconfigurable System-on-Chip Designs", *Proceedings of Design Automation Conference*, June 2004, pp. 395-400
[24] P. Ienne and R. Leupers, "Customizable Embedded Processors: Design Technologies and Applications", Morgan Kaufmann Publishers, 2006
[25] K. Seto and M. Fujita, "Custom Instruction Generation with High-Level Synthesis", *Symposium on Application Specific Processors*, 2008, pp.14-19
[26] W. Geurts, F. Catthoor, S. Vernalde and H.D. Man, "Accelerator Data-Path Synthesis for High-Throughput Signal Processing Applications", Kluwer Academic Publishers, 1997
[27] N. Moreano, E. Borin, C. de Souza, and G. Araujo, "Efficient Datapath Merging for Partially Reconfigurable Architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 24, No. 7, pp. 969-980, July 2005
[28] S.K Lam and T. Srikanthan, "Rapid Design of Area-Efficient Custom Instructions for Reconfigurable Embedded Processing", Journal of Systems Architecture, Vol. 55, No. 1, January 2009, pp. 1-14
[29] A.K.Verma, P. Brisk and P. Ienne, "Data-Flow Transformations to Maximize the Use of Carry-Save Representation in Arithmetic Circuits", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 27, No. 10, October 2008, pp. 1761-1774
[30] M. Ramsay, "Dataflow Dominance: A Definition and Characterization", University of Wisconsin-Madison, December 2003
[31] G. Dittmann, "Organizing Libraries of DFG patterns", *IEEE Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2004
[32] Xilinx ISE Foundation. Available: http://www.xilinx.com/ise/-logic_design_prod/foundation.htm
[33] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge and R.B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite", *IEEE International Workshop on Workload Characterization*, December 2001, pp. 3-14
[34] C. Lee, M. Potkonjak and W.H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", *Proceedings of the 13th Annual IEEE/ACM International Symposium on Microarchitecture*, December 1997, pp. 330-335
[35] P. Yu and T. Mitra, "Characterizing Embedded Applications for Instruction-Set Extensible Processors", *Proceedings of the 41st IEEE/ACM on Design Automation Conference*, June 2004, pp. 723-728
[36] T. Ahmed, P.D. Kundarewich, J.H. Anderson, B.L. Taylor and R. Aggarwal, "Architecture-Specific Packing for Virtex-5 FPGAs", *Proceedings of the ACM/SIGDA Symposium on Field programmable Gate Arrays*, February 2008, pp. 5-13
[37] I. Kuon and J. Rose, "Area and Delay Trade-Offs in the Circuit and Architecture Design of FPGAs", *Proceedings of the ACM/SIGDA Symposium on Field programmable Gate Arrays*, February 2008, pp. 149-158
[38] SimpleScalar Tool Set. Available: http://www.simplescalar.com/v4test.html
[39] Trimaran: An Infrastructure for Research in Instruction-Level Parallelism. Available: http://www.trimaran.org
[40] D. Mattson and M. Christensson, "Evaluation of Synthesizable CPU Cores", Master's Thesis, Chalmers University of Technology, 2004

Siew-Kei Lam (M'03) received the B.A.Sc. (Hons.) degree and the M.Eng. degree in computer engineering from Nanyang Technological University (NTU), Singapore. Since 1994, has been with NTU, where he is currently a Research Associate with the Centre for High Performance Embedded Systems and has worked on a number of challenging projects that involved the porting of complex algorithms in VLSI. He is also familiar with rapid prototyping and applicationspecific integrated-circuit design flow methodologies. His research interests include embedded system design algorithms and methodologies, algorithms-to-architectural translations, and high-speed arithmetic units.

Thambipillai Srikanthan (SM'92) joined Nanyang Technological University (NTU), Singapore in 1991. At present, he holds a full professor and joint appointments as Director of a 100 strong Centre for High Performance Embedded Systems (CHiPES) and Director of the Intelligent Devices and Systems (IDeAS) cluster. He founded CHiPES in 1998 and elevated it to a university level research centre in February 2000. He has published more than 250 technical papers. His research interests include design methodologies for complex embedded systems, architectural translations of compute intensive algorithms, computer arithmetic, and high-speed techniques for image processing and dynamic routing.



Christopher T. Clarke received the B.Eng. degree in engineering electronics and the Ph.D. degree in computer science from the University of Warwick, Coventry, U.K., in 1989 and 1994, respectively. From 1994 to 1997, he was a Lecturer with Nanyang Technological University, Singapore, where he was the Cofounder of the Centre for High Performance Embedded Systems. Since then, he has spent time in industry, both as an in-house Engineering Manager and independent Consultant for U.K. silicon design houses, system integrators, and multinationals such as Philips Semiconductors. Since March 2003, he has been with the Microelectronics and Optoelectronics Research Group, Department of Electronic and Electrical Engineering, University of Bath, Bath, U.K. He has been involved with many European union funded research projects including PEPS, CIRCE, SENS, and IMANE. Dr. Clarke is a member of the Centre for Advanced Sensor Technologies.