IPSJ Transactions on System LSI Design Methodology Vol. 4 193–209 (Aug. 2011)

Regular Paper

Data Flow Graph Partitioning Algorithms and Their Evaluations for Optimal Spatio-temporal Computation on a Coarse Grain Reconfigurable Architecture

Ratna Krishnamoorthy,[†1] Saptarsi Das,[†3]
Keshavan Varadarajan,[†3] Mythri Alle,[†3]
Masahiro Fujita,[†2] S K Nandy[†3]
and Ranjani Narayan[†4]

Coarse Grain Reconfigurable Architectures (CGRA) support spatial and temporal computation to speedup execution and reduce reconfiguration time. Thus compilation involves partitioning instructions spatially and scheduling them temporally. The task of partitioning is governed by the opposing forces of being able to expose as much parallelism as possible and reducing communication time. We extend Edge-Betweenness Centrality scheme, originally used for detecting community structures in social and biological networks, for partitioning instructions of a dataflow graph. We also implement several other partitioning algorithms from literature and compare the execution time obtained by each of these partitioning algorithms on a CGRA called REDEFINE. Centrality based partitioning scheme outperforms several other schemes with 6–20% execution time speedup for various Cryptographic kernels. REDEFINE using centrality based partitioning performs 9× better than a General Purpose Processor, as opposed to 7.76× better without using centrality based partitioning. Similarly, centrality improves the execution time comparison of AES-128 Decryption from 11× to 13.2×.

1. Introduction

Reconfigurable Processors are composed of an interconnection of computation units, which help exploit a higher degree of spatial computation[⋆1] than what is available in General Purpose Processors (GPP). This hardware organization helps exploit parallelism better than in a GPP [6]. However, this necessitates changes in the compilation process, which needs to address both spatial and temporal aspects of computation. Compilation for temporal computation involves identifying a total order of instruction, such that it satisfies all program dependences. For spatial computation, the compiler needs to allocate different instructions to different computation units available on the reconfigurable fabric, such that the overall computation and communication time is minimized. As mentioned in Ref. 11), this involves use of several VLSI CAD algorithms, viz. circuit clustering and partitioning, which were previously used for hardware synthesis. In this paper, we propose a new partitioning algorithm based on Edge Betweenness Centrality [10] for partitioning instructions of an acyclic dataflow graph. Recently Edge Betweenness Centrality has been proposed for detecting community structure in social and biological networks [10], and to the best of our knowledge, this is the first time it is applied to partitioning of programs for CGRA and achieves 10% speed up over other partitioning schemes. Although the proposed partitioning algorithm can be applied to CGRAs in general, in this paper we present the algorithm in the context of a specific CGRA, REDEFINE [1]. Detailed information of certain aspects of the architecture is required for accurate evaluation of the partitioning scheme. Hence, a brief introduction to the architecture and partitioning scheme currently used in REDEFINE is provided in the subsequent sections.

1.1 REDEFINE

The architecture of REDEFINE [1),7)] is presented in **Fig. 1**. The core computation engine of REDEFINE is the reconfigurable hardware fabric, which is an interconnection of tiles, where each tile includes a computation element (CE) and a router. The CEs are connected to the router. The routers are interconnected in a honeycomb topology, which forms a Network-on-Chip (NoC). The use of the NoC, as opposed to a programmable interconnect helps reduce the amount of configuration information needed to transfer data from one CE to another. The

†1 Department of Electronics Engineering, The University of Tokyo
†2 VLSI Design and Education Center, The University of Tokyo
†3 CAD Lab, SERC, Indian Institute of Science, Bangalore, India
†4 Morphing Machines, Bangalore, India

⋆1 In temporal execution instructions are sequenced and executed one at time. In spatial computation the number of instructions to be executed is equal to the number of hardware units that are available and the results from one operation to another is conveyed through dedicated wires or an interconnection [6].

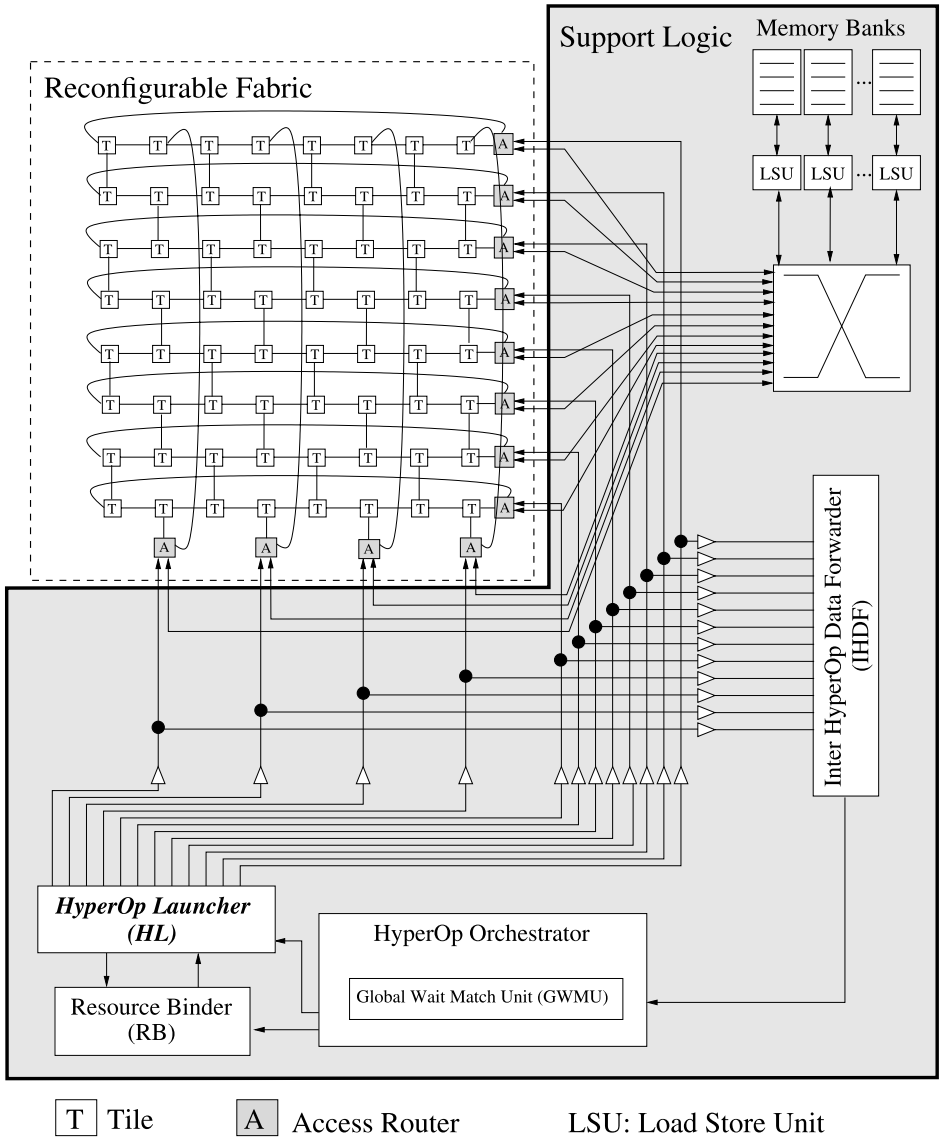

Fig. 1 Architecture of REDEFINE.

design of the router and the NoC is described in Ref. 8). Like other CGRAs, it employs a *Spatio-Temporal* execution paradigm, i.e., multiple CEs can be employed for spatial execution. Each CE can store several instructions, which are executed in dataflow order. Applications coded in C language are compiled into an executable by the REDEFINE compiler [2]. This transformation involves using the Static Single Assignment output of LLVM [5] to generate the Control Flow Graph. Several basic blocks are combined to form an application substructure, referred to as a HyperOp. A HyperOp is a vertex-induced subset of an application's dataflow graph such that they are acyclic, pairwise disjoint, and satisfy the convexity condition [2]. A HyperOp is partitioned into *p-HyperOps*, where each p-HyperOp is mapped to a CE.

Each CE can hold c instructions (we assume $c = 16$ in this paper) in the reservation station (reservation station is a part of the first pipeline stage). An instruction whose operands are available is selected for execution. A priority encoder logic performs arbitration when more than one instruction is ready to be executed simultaneously. The ready instruction along with the opcode and operands are transferred to the ALU. Apart from elementary operations the ALU supports custom function units (CFU) which are very specific to a certain problem domain. The ALU after performing the computation forwards the results to point of consumption through the NoC. In some cases the consuming instruction may be present within the same CE where the data was produced. In this case the data is sent on the bypass channel. The CE is pipelined and its latency is 3 clock cycles$^{\star 1}$. If the path through the router is taken to the destination then additional latency is incurred based on the traffic at the router and the hop distance of the destination.

Instructions of a HyperOp are distributed across various CEs. Instructions spread across various CEs of a HyperOp can execute in parallel (spatial execution) and instructions placed within the same CE execute sequentially (temporal execution). Hence this is referred to as "spatio-temporal" execution. For efficient spatio-temporal execution, the compiler must be able to *partition* the instructions into different p-HyperOps so as to minimize the total execution delay. *Increas-*

⋆1 Assuming Single cycle Function Units. There are few units which are not single cycle.

ing the number of p-HyperOps can help reduce execution time through increased exploitation of parallelism, but also increases the communication time which may affect execution time adversely.

REDEFINE can be specialized for a specific domain through the use of appropriate custom function units (CFUs) within each ALU. In this paper, we use cryptography kernels as our running example. The details of the CFUs for the REDEFINE Cryptography Fabric is given in Section 4.

HyperOp is a single schedulable entity. The scheduling of HyperOps and their launching is handled by the Support Logic (refer Fig. 1). The support Logic also facilitates transfer of data from one HyperOp (which is executing on the fabric) to another HyperOp (which is yet to be launched, awaiting arrival of all input operands). The launch and execution of a HyperOp proceeds as follows. HyperOps are ready to execute when all its input operands are available. The *HyperOp Orchestrator* schedules the ready HyperOp for launch. **Figure 2** shows the various steps from the time the HyperOp is scheduled upto the completion of its execution on the Fabric. Scheduling a HyperOp for launch involves sending the HyperOp identifier to the *Resource Binder* to determine the CEs on the fabric where the instructions of the HyperOp are to execute. HyperOp Orchestrator also transfers the input operands of the HyperOp to the *HyperOp Launcher*. The HyperOp Launcher transfers instructions and data operands (i.e., constants and inputs) to the CEs identified by the Resource Binder. Each CE follows a dataflow execution order i.e. an instruction whose input operands are available is ready to execute. Due to the dataflow execution of the CEs, the launching of the HyperOp and HyperOp execution on the Fabric overlap to a certain extent. This is refered to as *Self-Overlap*, indicated as *SO* in Fig. 2. *Perceived HyperOp launch Time* or *PHT* is the time between the start of the HyperOp launch and upto the time for the first instruction to start. The time taken to transfer all instruction and data operands of a HyperOp is called *Complete HyperOp launch Time* or *CHT*. *Fabric Execution Time* or *FET* is the time taken to execute all instructions of a HyperOp. The instructions of the HyperOp may produce data to be consumed within the same CE or data which is the input operand of another HyperOp. Input data for a HyperOp that is yet to be scheduled, is stored within the HyperOp Orchestrator. The CE forwards data that is meant for another HyperOp to the *Inter-HyperOp Data Forwarder* (IHDF) (refer Fig. 1). The IHDF determines the location within the HyperOp Orchestrator where the data needs to be written.

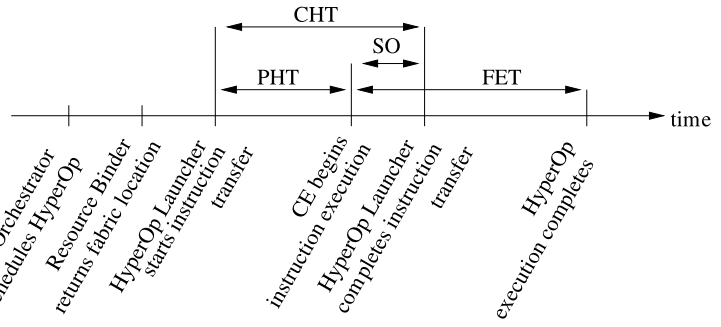

Fig. 2 Timeline showing the various steps involved in scheduling, launching and execution of a HyperOp. The time spent in each of these tasks are labeled. FET: Fabric Execution Time; PHT: Perceived HyperOp launch Time; CHT: Complete HyperOp Launch time; SO: Self-Overlap.

Of the Support Logic modules, the finer details of the HyperOp Launch process is needed for designing an efficient partitioning algorithm. As mentioned previously, when a HyperOp is ready for execution, the HyperOp Launcher transfers instructions belonging to the HyperOp to the CEs on the fabric. It is connected to the reconfigurable fabric through access routers present along the periphery (marked as A in Fig. 1). The HyperOp Launcher reads the instructions from the five$^{\star 1}$ instruction memory banks and chooses the closest router to the destination to transfer the instructions and constants. The HyperOp Launcher has five submodules which allow independent handling of instruction stream from each instruction memory bank. Each HyperOp Launcher sub-module transfers instructions, constants and input operands of a single p-HyperOp. In order to support a peak transfer rate of 5 instructions every clock cycle, the 5 instructions must belong to different p-HyperOps. *This design choice favours the use of more p-HyperOps.* For any given number of instructions in a HyperOp, it is beneficial

⋆1 It is 5 in the current implementation.

to use more p-HyperOps in order to reduce the transfer latency. *However, increasing the number of p-HyperOps increase the number of control packets such as those which mark the beginning and end of the p-HyperOp.*

1.2 Partitioning the HyperOp

While partitioning the HyperOp into p-HyperOps (each p-HyperOp is assigned to a CE), several factors need to be considered:

- Parallel instructions are best mapped to different CEs, since this helps exploit parallelism and hence reduce the overall time taken to execute a HyperOp.
- Dependent instructions are best mapped close together, potentially within the same CE. Even when the hop count is one (i.e., instruction is placed in the neighbouring CE) the time to transfer the data may be more than one clock cycle.

While these obvious factors to partitioning the HyperOp's dataflow graph appear to be non-conflicting, they do work as opposing forces. Two parallel instructions consuming data from the same parent instruction need to be placed in different p-HyperOps for exploiting more parallelism, but communication latency is minimized if they are placed in the same p-HyperOp and executed sequentially.

Yet another factor that needs to be considered is whether balanced partitioning or unbalanced partitioning should be preferred. Balanced partitions are those where the number of nodes (or instructions) included in each partition are approximately equal to each other i.e., the deviation, of the number of nodes included in a partition, from the average number of nodes across all partitions is minimal. Creation of balanced partitions will reduce the Complete HyperOp Launch Time (CHT) since all memory banks are busy to the same extent. However, this reduces the Self-Overlap (SO). On the other hand having unbalanced partitions increases the amount of Self-Overlap but also increases the Complete HyperOp launch Time (and Perceived HyperOp launch Time decreases). The choice of balanced versus unbalanced partitions seems unclear and requires further experimentation. In summary:

- *Use of more p-HyperOps:* Increased memory parallelism reduces reconfiguration time; More Instruction Level Parallelism (ILP) can be exploited. However, this can increase the communication cost since more p-HyperOps implies larger area over which they are spread on fabric. More p-HyperOps increases control packet overhead during HyperOp launch.
- *Use of balanced partitions:* The CHT decreases (and PHT becomes equal to or nearly equal to CHT and SO tends towards zero).
- *Use of unbalanced partitions:* PHT is less than the CHT and the SO increases and can help improve overall execution time.

We present a summary of related literature in Section 2. We present an adaptation of various algorithms including edge centrality based partitioning scheme for dataflow graphs (Section 3). The evaluation framework and the performance of each of these partitioning algorithms is presented in Section 4.

2. Related Work

Graph Partitioning is probably one of the most studied algorithms in computer science, due to its extensive applications from social networks to hardware netlists. In the area of compilation, graph partitioning problem is relevant in the context of reconfigurable architectures and multiprocessor systems. However, the solution to the graph partitioning problem in each of these contexts is expected to be different due to different objective functions. The most important factor which influences this is the granularity of the computation units. In reconfigurable architectures such as REDEFINE, the granularity of each unit is a few 10 s of instructions (16 instructions in all our experiments). In the context of multiprocessor systems viz. TRIPS [19], Wavescalar [20] and RAW [14),21] the granularity of instruction is much higher and can be as many as 128 instructions (in TRIPS; RAW and Wavescalar employ instruction caches and can accomodate a large number of instructions). All these multiprocessor systems employ a Network-on-Chip, which is similar to REDEFINE. However, the computation element in these systems can accomodate a much larger number of instructions.

TRIPS [19] uses a simple computation element with register file, ALU and router at each node of the interconnect. There are a set of globally accessible registers (one per column). The nodes are interconnected through a multi-layer mesh interconnect. Each computation element can accomodate 128 instructions. The instruction storage in each computation element can be used in various modes. In a specific mode the instruction store's registers with the same index can be used as a configuration frame and all instructions within a frame can pertain to

one application sub-structure. Several application sub-structures can be loaded into the various configuration frames to reduce the configuration load overhead. This mode helps in exploiting ILP. In another an application substructure is loaded into a single computation element and several computation elements can interact with each other. This mode helps in exploiting Data Level Parallelism (DLP) or Thread Level Parallelism (TLP). The TRIPS processor allows instructions belonging to one application substructure to exchange data through the network-on-chip and data interactions between application sub-structures are facilitated through register files. An application specified in a high level language is compiled into hyperblocks [15]. The hyperblock in TRIPS, referred to as the TRIPS block, has restrictions on the number of instructions and number of load-store instructions. It also does not include basic blocks based on profile information. The orchestration of hyperblocks is based on control flow.

Partitioning the graph for exploiting ILP involves different objective function as opposed to partitioning the graph for exploiting DLP or TLP. When trying to exploit ILP (to reduce the execution time), one needs to weigh the cost of exploiting ILP, which involves NoC communication, to the actual performance improvement that can be obtained. Closely interacting nodes cannot be placed such that their communication happens over the NoC. This negatively impacts performance. *The aim of this study is to determine which communication is classified as closely interacting and those communications which can be accomodated on the NoC without affecting the execution time.*

Partitioning problem has been studied extensively for reconfigurable architectures like time multiplexed FPGA [22], dynamically reconfigurable FPGAs (DRFPGAs) [4], etc. One of the early and most influential work in this field was done by Kernighan and Lin in 1970 [12]. In their seminal paper [12] they proposed a balanced partitioning heuristic. The details of the algorithm are excluded since the algorithm is well understood and several variants of this algorithm exist in literature. This algorithm is of specific interest in our case, since it minimizes the communication between two partitions, while keeping the partitions balanced. This condition directly helps reduce the communication cost and helps contain all dependent instructions within the same partition. We explain its adaptation to REDEFINE and results in the subsequent sections.

Reference 9) is one of the first partitioning works in the domain of reconfigurable computing. A program or an application is represented by a data flow graph (DFG) which is partitioned into a set of segments such that size of each segment is less than the size of the reconfigurable unit, where each unit is an FPGA board. Two partitioning algortihms have been proposed in this paper, namely Level based partitioning and Clustering based partitioning have been proposed. In level based partitioning the nodes of the DFG are assigned ASAP levels. This algorthim exposes parallelism by considering all nodes at same level for parallel execution. The DFG is horizontally cut keeping the size of the reconfigurable unit in mind. In clustering based partitioning, the nodes are clustered with the common parent while traversing the DFG in a breadth first traversal fashion By doing this it tries to reduce the number of terminal edges between the partitions and hence reducing communication overhead. We implement two variants of the clustering based partition scheme, where nodes are assigned according to its parent affinity. Clustering based partitioning algorithms tend to create unbalanced partitions due to its affinity.

List scheduling method is another popular algorithm used for performing partitioning. The method has the advantage of having a linear run-time in the number of nodes of the graph being partitioned. Several heuristics based on List scheduling have been proposed. In the temporal partitioning scheme Pandey, et al. [16] proposed the use of an enhanced version of the force directed list scheduling (FDLS) [17),18]. In FDLS, the probability of each node being placed at a specific time step is computed based on distribution graph that depicts the number of concurrent operations. Forces are proportional to the concurrency in the system. The most desired solution is the one which achieves the least increase in the execution time for the given set of resources. Pandey et al. apply FDLS iteratively on the DFG by varing the resource set available in the form of CEs on the fabric and the communication logic. Since we do not restrict the number of CEs over which a HyperOp can span, we employ a variant of the Force Directed Scheduling technique, in one of the schemes, instead of FDLS (refer Fig. 5). A variant of the FDS algorithm was proposed in Ref. 4). In this paper, the authors extend the FDS scheme by including the cost of the communication modules that needs to be placed between two partitions, over and above the gate cost accounted for by

the original FDS algorithm. The cost of communication is given half the weight of the gate costs. As indicated in Bobda's thesis [3], list scheduling tends to suffer due to the level based assignment. This has the potential to affect the total execution time since it does not take into consideration the communication between the partitions. The same applies to FDS, as it follows level based assignment. However, the algorithm can be a good candidate for graphs with less density.

The use of network Flow based partitioning schemes in the context of netlist partitioning was made possible due to the paper by Yang, et al. [23] who proposed the Flow based bipartitioning scheme which generates balanced partitions. In this scheme a min-net cut is accepted only if the resulting partitions generated after the edge removal is *r-balanced*. However, the construction of the flow network requires the addition of several nodes and edges and the resulting partitions need to be reconverted back to the original form prior to generating the executable. A scheme similar to network flow is the centrality based partitioning, which is used extensively in the context of social networks. There are several measures of centrality which are defined. We are specifically interested in the edge-betweenness centrality based partitioning technique. This was proposed by Girvan, et al. [10] in the context Social and Biological networks. Edge betweenness centrality is a measure that determines how "between" an edge. The adaptation of these algorithms in our context is elucidated in the subsequent section.

3. Description of the Algorithms

In this section, we describe three types of algorithms namely parent affinity based scheme, Kernighan-Lin heuristic and edge centrality based partitioning scheme. We explore three variants of parent affinity scheme. Apart from these algorithms, we also describe the partitioning scheme currently employed in REDEFINE. The evaluation of each of these algorithms is presented in Section 4.

3.1 Parent Affinity based Schemes

Purna, et al. [9] proposed two different algorithms, namely: Level based partitioning and Clustering based partitioning algorithms. This algorithm exposes parallelism by assigning all nodes at same ASAP level to different p-HyperOps. The terms level and ASAP level are used interchangeably in the rest of the paper. Assignment of nodes at the same level to different partitions eliminates the artificial sequentialization of instructions due to assignment of nodes in the same level to the same Computation Element (CE). In order to reduce the impact on communication, the nodes at the subsequent ASAP level are assigned in the same p-HyperOp as the parent node. A scheme similar to parent affinity is also used in Ref. 14). In the following subsection, we explain the implementation of parent affinity scheme which is common to three partitioning schemes, which are presented thereafter. It is to be noted that the algorithms reported in Ref. 9) are for splitting a netlist across multiple FPGA boards and hence it cannot be directly applied in our context. We have developed three variants which uses the same underlying mechanism used by Purna, et al. [9]. However, the algorithms themselves are very different from those reported in Ref. 9).

3.1.1 Parent Affinity

In our implementation each parent of a node is assigned a weight[⋆1] in the following manner: $weight(p) = \frac{1}{level(n)-level(p)}$, where p is the parent node and n is the child node whose p-HyperOp is to be determined. This measure for weight assigns greater importance to parents which are closer in level. The difference in levels has an impact on when the instruction is executed. A node at a lower level has a higher probability of being scheduled for execution after a node a higher level. This implies that the result gets a shorter time to travel to the destination. Thus, the consuming instruction is best placed closer to the last input producing source. This is approximated using the level numbers. However, if the p-HyperOp containing the parent with the highest weight cannot accomodate more instructions, then the p-HyperOp containing the parent with the second highest parent weight is used. The parent affinity computation is detailed in **Fig. 3**.

3.1.2 Interleaved Parent Affinity

The parent affinity mechanism assigns the nodes to partitions where its parent nodes are present. Any two data dependent instructions assigned to the same CE, can execute at best 3 cycles apart. This is because the CE has a 3 stage pipeline. If only sequentially dependent instructions are assigned to the CE, then utilization of the Function Units within the CE drops to 33%. This can be

⋆1 This weight assignment is not mentioned in Ref. 9) but is our adaptation of this algorithm.

```
1: for all parents p of node do
2:     h ← pHyperOp number of p
3:     weight(h) ← 1/(level(node)−level(p))
4: end for
5: maxpHyperOp ← max(weight)
6: while no. of instructions in maxpHyperOp exceeds limits do
7:     maxpHyperOp ← next highest weight pHyperOp
8: end while
9: if all parent pHyperOps are full then
10:    return newpHyperOp
11: end if
12: return maxpHyperOp
```

Fig. 3 pHyperOp Selection based on parent affinity.

maximally utilized by assigning more independent, i.e., parallel instructions to the same CE. In the Interleaved Parent Affinity (ILPA) partitioning method, the nodes with the lowest ASAP level number are interleaved into $\lceil \frac{n}{m} \rceil$ partitions, where n is the number of nodes with the lowest ASAP level and m is the number of pipeline stages in the CE. The nodes at the remaining level are assigned based on the parent affinity mechanism, explained in Section 3.1.1. A variant of this scheme without Parent Affinity was experimented previously in Ref. 13). Since it did not show promising results, it is not described in this paper. The algorithm for ILPA partitioning is shown in **Fig. 4**.

3.1.3 Non-Interleaved Parent Affinity-Compute

Unlike the ILPA scheme, in the Non-Interleaved Parent Affinity-Compute (NIPA-C) partitioning technique the nodes with least ASAP level are assigned to p partitions. The value of p is determined by running the Force Directed Scheduling algorithm [17] on the graph and computing the average across all time steps. The NIPA-C partitioning method tries to exploit as much parallelism as possible (without consideration to resource utilization). The nodes at subsequent ASAP levels are assigned to partitions based on parent affinity mechanism (Fig. 3). The NIPA-C algorithm is shown in **Fig. 5**.

3.1.4 Non-Interleaved Parent Affinity-Memory

Both the ILPA and the NIPA-C algorithms are based on the available ILP and do not take into acount the available instruction memory bandwidth. As mentioned previously, the HyperOp Launcher is connected to five instruction memory

```
1: maxTopLevelPHyperOps ← ⌈number_toplevelnodes / pipelineDepth⌉
2: currentpHopNum ← 0
3: counter ← 0
4: for node n in topologically sorted list do
5:     predecessor ← predecessors(n)
6:     if predecessor = [] then
7:         while currentpHopNum is full do
8:             currentpHopNum ← currentpHopNum + 1
9:             if currentpHopNum > maxTopLevelpHyperOps then
10:                currentpHopNum ← 0
11:            end if
12:        end while
13:        assign n to currentpHopNum
14:        counter ← counter + 1
15:        if counter = pipelineDepth then
16:            currentpHopNum ← currentpHopNum + 1
17:            if currentpHopNum > maxTopLevelpHyperOps then
18:                currentpHopNum ← 0
19:            end if
20:        end if
21:    end if
22: end for
23: for node n in topologically sorted list do
24:    predecessor ← predecessors(n)
25:    if predecessor ≠ [] then
26:        parentAffinity(G, n)
27:    end if
28: end for
```

Fig. 4 Algorithm describing the ILPA scheme.

banks. The time needed to launch 1 p-HyperOp or 5 p-HyperOps remains the same since a "Launch-Level Parallism" of 5 p-HyperOps can be obtained. In order to minimize the Complete HyperOp launch Time we explore a scheme where the number of p-HyperOps to be used is snapped to the closest higher multiple of i, where i is the number of instruction memory banks. This is used to determine the number of p-HyperOps to which nodes with the least ASAP level are assigned. All other nodes are assigned partitions (p-HyperOps) based on the parent affinity mechanism, shown in Fig. 3. The function that computes the maximum number of p-HyperOps for the NIPA-M scheme is shown in **Fig. 6**.

3.2 Kernighan-Lin Algorithm

The Kernighan-Lin (KL) [12] algorithm was proposed as a heuristic for parti-

```
1: maxTopLevelpHyperOps ← FDS_avgResources(G)
2: currentpHopNum ← 0
3: for node n in topologically sorted list do
4:    predecessor ← predecessors(n)
5:    if predecessor == [] then
6:       while currentpHopNum is full do
7:          currentpHopNum ← currentpHopNum + 1
8:          if currentpHopNum > maxTopLevelpHyperOps then
9:             currentpHopNum ← 0
10:         end if
11:      end while
12:      assign n to currentpHopNum
13:      currentpHopNum ← currentpHopNum + 1
14:      if currentpHopNum > maxTopLevelpHyperOps then
15:         currentpHopNum ← 0
16:      end if
17:   end if
18: end for
19: for node n in topologically sorted list do
20:   predecessor ← predecessors(n)
21:   if predecessor ≠ [] then
22:      parentAffinity(G, n)
23:   end if
24: end for
```

Fig. 5 Algorithm for Non-Interleaved Program Affinity-Compute.

```
1: n ← number_vertices(G)
2: maxpHops ← ⌈ n / InstructionsPerpHyperOp ⌉
3: maxtransfers ← ⌈ maxpHops / memoryBanks ⌉
4: maxtoplevel ← maxtransfers * memoryBanks
5: return maxtoplevel
```

Fig. 6 Calculate no. of top levels snapping to closest multiple of number of memory banks for NIPA-M scheme.

tioning the netlist into equal sized blocks with minimal communication so that they may be mapped to PCBs. The hueristic is a bi-partitioning algorithm which starts with an initial equal sized partition and then proceeds by exchanging "most externally communicating" vertices. The algorithm stops when no more inter-partition communication reduction can be obtained with any vertex exchanges. For partitioning the HyperOp, we first assign all nodes to one p-HyperOp. If the number of instructions within a p-HyperOp exceeds the maximum allowed limit,

```
1: pHyperOps[0] ← nodesofG
2: while needToDivide(pHyperOps) do
3:    for each pHyperOp p in pHyperOps do
4:       n ← equiDivide(pHyperOps[p])
5:       kernighan_lin(G, pHyperOps[p], pHyperOps[n])
6:    end for
7: end while
```

Fig. 7 Kernighan-Lin based partitioning algorithm. The function *kernighan_lin* exchanges nodes between pHyperOps n and p so as to minimize the communication between them.

then the function *needToDivide* returns *true*. Since Kernighan-Lin algorithm produces balanced partitions, if one partition exceeds the size, then all partitions will exceed the size. For each p-HyperOp in the list we invoke the function *equiDivide*, which partitions the p-HyperOp into two equal halves. This is followed by a call to the *kernighan_lin* algorithm which makes repeated exchanges of nodes with high inter-partition interaction until no positive gain achieving nodes exist for exchange. As mentioned previously, parent affinity based techniques tends to create unbalanced partitions. KL algorithm on the other hand creates equal sized partitions and also tries to reduce the inter-pHyperOp communication. The reduction in inter-pHyperOp communication helps in reducing the on-Fabric communication cost. Creation of balanced partitions helps reduce Complete HyperOp launch Time and also decreases the Self-Overlap. The KL partitioning scheme is shown in **Fig. 7**.

3.3 Edge Betweenness Centrality

The edge betweenness centrality based technique is similar in nature to network flow based scheme, but more amenable to graphs without edge weights viz. dataflow graphs and social networks. This technique was first proposed by Girvan, et al. [10]. Edge betweenness centrality is a measure that determines how "between" an edge is; it measures the number of shortest paths, between every pair of vertices, which passes through that edge. If more than one shortest path exists between the considered pair of vertices then equal weight is assigned to all paths such that they all add up to 1. The edge betweenness is higher if the edge connects two clusters since several vertices's shortest paths pass through that edge. The algorithm proposed in Ref. 10) involves computing the edge betweenness centrality followed by removal of the edge with the highest betweenness.

This is followed by recomputation of the edge betweenness for the graph and removal of the edge and so on. This scheme helps identify edges which are "most between" two clusters. Elimination of this edge from the graph, partitions the graph into two components which are tightly coupled.

Partitioning nodes into p-HyperOps involves assigning communication between certain vertices (represented by edges in the HyperOp's data flow graph) to the Network-on-Chip, while other communication happens over the bypass channel, i.e., communication between nodes within the same partition does not leave the CE and communication between nodes in different partitions are facilitated over the NoC. The choice of the edges assigned to the NoC are as critical as the reduction in communication achieved through efficient partitioning. In a general graph containing a clique as a subgraph (or subgraphs with higher graph density⋆1 than the average graph density of the whole graph), edge betweenness centrality based partitioning algorithm would break those edges which connect the clique (or higher density regions) with the rest of the graph. In a dataflow graph too, centrality breaks those edges around subgraphs with higher graph density from the rest of the graph. Since subgraph of higher graph density have more interactions, it is therefore logical to assign these communications to a bypass channel (which has a lower cost of communication) instead of making them NoC communications (which have a higher cost).

This technique is explained with a simple example shown in **Fig. 8**. The figure shows a portion of the dataflow graph of a HyperOp for a simple expression $a * b + c$, where a and b are loaded from memory and c is obtained from a previous computation. The edge between two loads (viz. `ld1` and `ld2`) indicate a sequencing predicate (needed to ensure sequential consistency). The result of this expression is sent out of the HyperOp using the `o/p2` operation and the sequencing predicate from the HyperOp is forwarded to the next load within the HyperOp `ld3`. The graph below `ld3` is not shown. While computing the centrality, we treat the graph as an undirected graph. The shortest path from every source to `ld3` is shown in Fig. 8. As is evident, one of the edges with highest

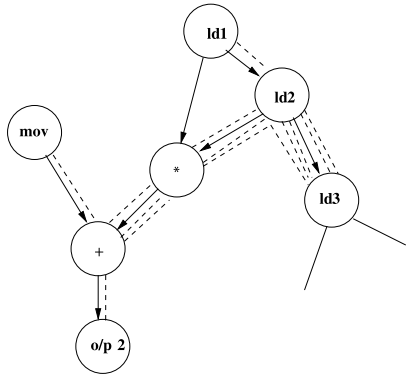

Fig. 8 The figure shows a portion of the dataflow graph of a HyperOp. The solid lines are the edges of the dataflow graph and the dotted line traces the path from every node to the node *ld3*.

centrality is the one between `ld2` and `ld3`. This edge (between `ld2` and `ld3`) is assigned to the NoC. Such an assignment is advantageous because the cost of communication on the NoC is much higher than the cost of communication over the bypass channel. Assigning edges which interconnect two clusters to the NoC makes computation of nodes within a cluster faster.

The edge-betweenness centrality based partitioning algorithm works by assigning edges with high edge-betweenness to the NoC and the rest of the communication is local to the CE. The algorithm subsumes the parent affinity scheme, but groups only those successors connected by low centrality edges into the same partition. This adaptation of the original algorithm [10] (shown in **Fig. 9**) has the following changes over the original algorithm [10]:

- Edge Centrality on Subgraph: Several times removal of an edge causes the graph to be disconnected⋆2. Instead of recomputing the Edge betweenness centrality on the whole graph, it is computed on a connected component. A connected component is chosen if it has more than the maximum number of nodes (i.e., $c = 16$, the maximum number of instructions that a CE can

⋆1 Graph Density measures the ratio of edges to the total number of possible edges for a given set of vertices and is given by the relation $\frac{e}{n*(n-1)}$ for a directed graph.

⋆2 If more than one path connects two the clusters, then removal of the edge does not cause the graph to be disconnected Further, this edge may be the one which appears on the shortest path but may not be the only path.

```
 1: if number_of_nodes(G) > maxInstruction then
 2:     compute the edge betweenness centrality for G
 3:     e ←edge with the highest centrality
 4:     flag ← true
 5:     while there exists atleast one connected component with more nodes than allowed do
 6:         Remove edge e from G
 7:         H ← convert G to an undirected graph
 8:         graphList ← list of connected components of H
 9:         for g in graphList do
10:             if number_of_nodes(g) > maxInstruction then
11:                 compute the edge betweenness centrality for g
12:                 e ←edge with the highest centrality
13:                 break from the loop
14:             end if
15:         end for
16:     end while
17: end if
```

Fig. 9 Partitioning based on Edge betweenness Centrality.

hold).

- The algorithm terminates when no connected components has more than the maximum number of nodes.

Parallelism is exploited in this scheme only at the level of clusters. Centrality may create unbalanced partitions since it does not take into account the ensuing size of partitions after edge removal.

3.4 Time Complexity

The partitioning algorithm invokes edge-betweenness centrality repeatedly. The time complexity of the edge-centrality algorithm is known to be $O(n^2 \log(n))$ [10]. This algorithm is invoked repeatedly after the removal of each edge. The time complexity of the partitioning algorithm can therefore be written as $O(r \cdot n^2 \log(n))$, where r corresponds to the number of edges that are removed. We speculate that $r \ll e$, where e is the number of edges. This assumption on r is validated and results of which is presented is presented in Section 4.

3.5 Existing p-HyperOp Generation Scheme

REDEFINE compiler implements the following greedy algorithm to generate p-HyperOps [1]. The HyperOp's dataflow graph (which is a subset of the application's dataflow graph) is broken into individual threads of sequential instructions. The inter-thread interaction is measured and two threads with the highest interaction are interleaved together. Each of these interleaved components forms a p-HyperOp. If the interleaved component has more instructions than what is allowed in a p-HyperOp, it is horizontally split into two components and so on.

4. Experiments and Observations

4.1 Applications and Kernels

As mentioned previously, to evaluate the performance of the various partitioning schemes, we have chosen a set of applications and kernels spanning the cryptography domain.

- *Applications*: These include applications such as a deterministic finite automaton, SHA-1, AES-128 bit encryption and decryption, a pseudo random number generator and CRC-16.
- *Elliptic Curve Cryptography kernels*: Elliptic Curve Cryptography (ECC) kernels form the set of primitive operations needed for implementing next generation Cryptography Algorithms. The key sizes in these cases are very large and computations are based on algebraic properties of Galois Fields. The kernels include Field Addition, Field Multiplication, Field Squaring and Reduction operations based on binary fields using polynomial basis. The first set of these kernels are software implementations. Another variant of the same kernels accelerated with Custom Function Units (CFU) are also used to test the partitioning algorithms. The use of CFUs render the dataflow graph completely different. The kernels employing CFUs have a different structure, since in many cases loops or large instruction blocks are replaced by a CFU. The macro interactions in the original kernels manifest themselves are micro interactions when CFUs are used. It must be noted that only those ECC kernels marked with CFU make use of the CFU. The other variant of the kernel does not employ the CFU.

4.2 Simulation Environment

To evaluate the performance of the partitioning scheme, we have performed a full simulation of the generated p-HyperOps on the REDEFINE SystemC based simulator. The simulator is cycle-accurate and models an 8×8 fabric. Each tile of the fabric consists of a CE and a router. The CE includes a reservation station that can store 16 instructions, its operands and predicates. The CEs

issue one instruction every clock cycle to the function unit. Each CE supports all integer arithmetic operations (other than division), logical operations, shift and rotate operations, memory access instructions, data transfer instruction (from one CE to another), REDEFINE-specific instructions (viz. instructions to send data to the Support Logic). Apart from these each CE is enhanced with Custom Function Units (CFU) which are specific for the domain under consideration. For cryptography, we include the following CFUs in the CE:

- Field Multiplier: This performs a Galois Field multiplication on 2 32 bit operands with the 3rd input being an irreducible polynomial over $GF(2^n)$. It also supports vectored-execution mode for 8 bit and 16 bit operaands. This CFU is used in several applications and kernels. These include CRC 16, AES-128 Encryption, AES-128 Decryption, Field Multiplication, Point Addition and Point Doubling. This operation takes 3 cycles to complete execution.
- Field Squarer: This CFU performs a squaring on a 32 bit number. While this can be achieved using a multiplier as well. However, the multiplier returns a reduced result. Since an unreduced result is desired a separate CFU was implemented. Squaring in binary fields using polynomial basis has interesting properties which make this CFU very cheap and the result can be obtained within 1 clock cycle. This CFU is used by the Field Squarer, Point Addition and Point Doubling codes.
- Barrett Reduction: This CFU performs a field reduction operation with respect to an irreducible polynomial. This CFU is used by the Barrett Reduction, Point Addition and Point Doubling kernels.

As mentioned previously, the routers are connected in a honeycomb topology. Each router implements four virtual channels and ensures in-order delivery, i.e., 2 packets sent from the same source to the same destination arrive at the destination in the same order as sent from the source. This is achieved primarily due to the use of deterministic routing and the use of aging policy in routers, which gives a higher priority to older packets.

The process of evaluation involves compiling applications listed in **Table 1** written in C language into HyperOps. The compiler outputs the dataflow graphs of all the HyperOps. Different partitioning schemes are used to generate the p-HyperOps. These are then converted into binaries which are executed on the simulator. The partitioning schemes are evaluated with respect to the overall execution time of each application.

Table 1 Table showing the overall execution time achieved by various partitioning algorithms for various applications.

Application	Application Type	Current	NIPA-C	NIPA-M	ILPA	Centrality	KL
DFA		30193	26784	27351	27072	***24050***	27483
Random Number Generation	Cryptography	8714	8985	8953	9129	***8181***	8472
SHA-1	Applications	33714	37387	33343	35020	***29599***	34614
CRC 16		***26517***	28563	29327	28568	27007	27015
AES Encrypt		5632	5540	5533	5718	***4807***	5435
AES Decrypt		5831	6020	6001	5930	***4848***	5693
Field Adder		125	119	124	123	***114***	120
Field Multiplication		145931	154854	154810	148322	***145663***	146213
Field Squarer	ECC kernels	18619	18763	18700	18787	16810	***15692***
Barrett Reduction		2794	2592	2548	2553	***2533***	2656
Field Multiplication		1675	1808	1736	1769	***1447***	1556
Field Squarer		248	247	249	247	***233***	***233***
Barrett Reduction	ECC kernels	166	166	174	168	***154***	162
Point Addition	with CFU	7513	7821	7665	7950	***6415***	6994
Point Doubling		4419	4566	4535	4661	***3829***	4192

4.3 Results

Comparison of the execution time obtained for the applications are listed in Table 1. The plot of the average execution time recorded for various partitioning schemes normalized with respect to the existing partitioning scheme (described in Section 3.5) is shown in **Fig. 10**. Centrality based scheme shows 6–20% reduction in execution time in comparison to the current scheme. In two cases, the centrality scheme performs worse than the other schemes. In the case of CRC-16, the existing scheme performs 1.8% better than centrality and the Kernighan-Lin scheme performs 9.7% better than centrality for Field Squarer (without CFU).

The centrality based scheme gains primarily on account of use of higher number of p-HyperOps and appropriate choice of edges to be broken. As mentioned previously, increasing the number p-HyperOps has a favourable effect on the PHT and the appropriate choice of edges has a favourable impact on the FET. This is illustrated by the drop in normalized PHT and a drop in normalized FET in Fig. 10. As noted previously, this is because centrality tends to create a unbalanced partitions, i.e., few small partitions and a few large partitions. While the instructions of the larger partitions are still being loaded, the smaller partitions start execution. This stretches observed fabric execution time but

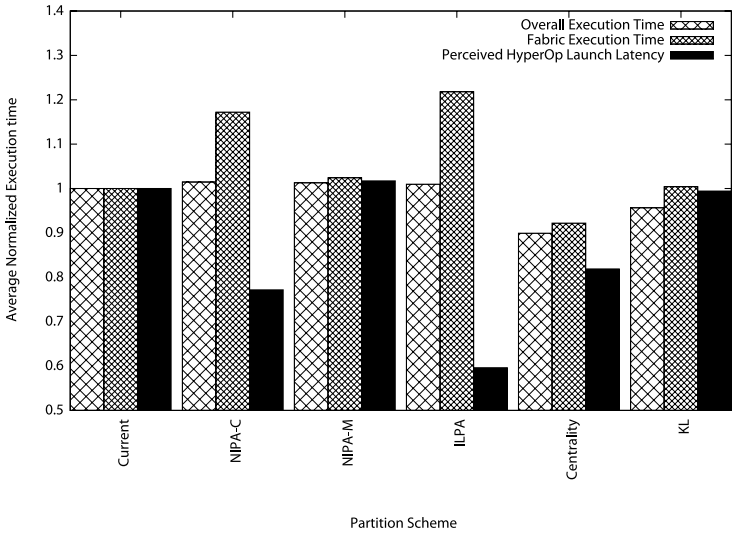

Fig. 10 The plot showing the average normalized execution time (w.r.t existing scheme) for various partitioning schemes. The plot also shows the average normalized PHT and average normalized FET.

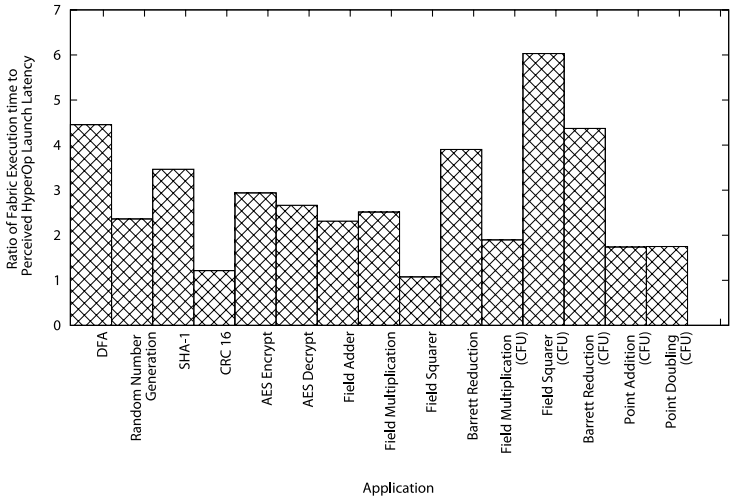

Fig. 11 The plot shows the ratio of FET to PHT for various application when the Centrality based partitioning scheme is applied.

reduces overall execution time due to the overlap. It should be noted that parallel execution of smaller partitions are also faciliated by the nature of Centrality based partitioning in creating dense subgraphs, which tends to execute within the same CE and external communication is minimized. Centrality records the lowest fabric execution time (refer Fig. 10).

On the other hand, KL based partitioning scheme creates nearly equal sized partitions, so the extent of self overlap is reduced. Equal sized partitions help in reducing the CHT. As seen from Fig. 10, ILPA scheme achieves the lowest PHT, however also records the highest average FET. Parent Affinity based scheme, as mentioned in Section 3.1.1, tends to create highly unbalanced partitions that explains the lowest HyperOp launch latency recorded. In the parent affinity based scheme the parent that is closer in level to the said node is preferred. This can lead to long chain of nodes or threads. Creation of long threads invariably leads to other partitions which have shorter threads. Further in ILPA, unlike NIPA, since more top level nodes are assigned to the same p-HyperOp three threads are interleaved into the same p-HyperOp. This is the reason for NIPA-C and ILPA recording the least average normalized PHT.

It is observed that applications whose ratio of PHT to FET is close to one, i.e., the PHT is nearly equal to the FET (**Fig. 11**), show poor performance when centrality based partitioning scheme is used (viz. Field Squarer – without CFU and CRC-16). This implies that the CHT is either nearly equal to or greater than the FET. In such a case, it is not advantageous to decrease the PHT at the cost of the CHT. Even though decreasing PHT (hence increasing SO and hence increasing CHT) has a favourable impact on the FET in cases where the ratio is close to 1, the favourable impact on the FET may not be able to offset the increase in CHT. Also, Amdahl's law would favour the reduction of the CHT in the said case, instead of the FET. In order to effect such a change, balanced partitions need to be created. Such a balanced centrality based partitioning approach would be pursued in future.

In conclusion, centrality partitioning algorithm performs well for most applications. If it is observed that the ratio of the Fabric execution time to the

Table 2 The table gives the running time of the partitioning and REDEFINE compilation for various applications.

Application	Running Time of Partitioning (in seconds)	Running Time of Compilation (in seconds)
DFA	0.216	0.109
Random Number Generation	1.425	0.199
SHA-1	1.096	0.519
CRC 16	0.165	0.079
AES Encrypt	1.148	4.667
AES Decrypt	1.143	5.46
Field Adder	0.19	0.054
Field Multiplication	5.56	8.784
Field Squarer	0.328	0.432
Barrett Reduction	3.877	1.476
Field Multiplication (CFU)	0.743	5.837
Field Squarer (CFU)	0.271	0.114
Barrett Reduction (CFU)	0.425	0.079
Point Addition	1.798	8.726
Point Doubling	3.352	10.845

HyperOp Launch latency is close to 1, then it may be worthwhile to experiment with the partitions producing balanced partitions viz. Kernighan -Lin Partitioning scheme.

4.4 Algorithm Running Time and Time Complexity

Table 2 gives the details of the time it takes to partition the the graph. Also, shown alongside is the time it takes to compile the application (other than the time spent in partitioning). The time spent in the code was determined using the `time` command in Unix.

As can be observed, from the table it can be seen that the running time of the algorithm is quite low and takes at most a few seconds. Each of these applications comprise several HyperOps. Each of these HyperOps which are partitioned contain several nodes with a maximum of 124 nodes and an average of 43 nodes per HyperOp. The maximum number of nodes that are permitted within a HyperOp$^{\star 1}$ is 128. **Table 3** shows the average density of these HyperOps for each of the applications listed in Table 2. The average ratio of the number of edges cut by centrality to the number of edges in the HyperOp and the average ratio of the number of edges cut by centrality to the number of vertices in the HyperOp are also presented alongside.

Table 3 Table showing the Average Graph Density for various application, the Average numbers of edges cut by centrality to the number of edges in the graph and the average number of edges cut to the number of vertices in the graph.

Application	Average Graph Density	Average Edge Cut to Edge Count	Average Edge Cut to Vertex Count
DFA	0.1034	0.0328	0.0366
Random Number Generator	0.0708	0.0786	0.1014
SHA1	0.0407	0.0897	0.1058
CRC 16	0.0689	0.0490	0.0595
AES Encrypt	0.0196	0.0912	0.1120
AES Decrypt	0.0282	0.0892	0.1043
Field Adder	0.0236	0.1176	0.1277
Field Multiplication	0.0309	0.1059	0.1336
Field Squarer	0.0569	0.0586	0.0638
Barrett Reduction	0.0633	0.0704	0.0841
Field Multiplication (CFU)	0.0379	0.0382	0.0365
Field Squarer (CFU)	0.0202	0.1266	0.1587
Barrett Reduction (CFU)	0.0219	0.1875	0.2459
Point Addition	0.0375	0.0451	0.0446
Point Doubling	0.0471	0.0452	0.0479
Maximum	0.1034	0.1875	0.2459
Minimum	0.0196	0.0328	0.0364
Average	0.0447	0.0817	0.0975
Standard Deviation	0.0231	0.0399	0.0540

There are several things which can be concluded from Table 3.

- The number of edges which centrality removes is limited by the number of edges in the graph. Given n vertices in a directed graph, $n(n-1)$ edges can be drawn. Since all HyperOps are acyclic, the number of possible edges reduces to $\frac{n(n-1)}{2}$. Column 2 in Table 3 shows the average graph density.

⋆1 A HyperOp can have a fixed number of nodes, since instructions of a HyperOp has to be accomodated on the fabric at the same time.

It is evident that the density is very low (average of 4% and a maximum of 10%), i.e., the number of edges present in the dataflow graph is far lesser than the maximum possible number of edges.

- We had speculated that r (the number of edges considered for removal) is far lesser than the number of edges, e, in the graph. Column 3 of Table 3 indicates that the average ratio $\frac{r}{e}$ is about 8% (with a maximum of 18%). This shows that our speculation was accurate.
- Column 4 of Table 3 shows the ratio $\frac{r}{n}$. This measure averages 9.8%. Thus $r = kn$, where $k \approx 0.1$. The complexity of the partitioning algorithm $O(r \cdot n^2 \log n)$ can be rewritten as $O(kn \cdot n^2 \log n)$ or $O(k \cdot n^3 \log n)$ where $k \approx 0.1$.

It must be noted that some of the observations made above, may not apply in the context of all applications. However, we believe that the observations will hold in general, for most applications.

4.5 Putting Results into Perspective

In order to understand these cycle counts in relation to other processors, we compare the performance of AES encryption and decryption performance with the performance on a General Purpose Processor. The AES-128 kernels have been recently subjected to several successful cache timing attacks. In order to prevent such an attack from occurring there is a need to eliminate any memory based lookup operation. In order to achieve that we obtain the result of the Sub-bytes stage of AES through computation of the field inverse for the given 8 bit number instead of performing a look up. This mode of execution does not lend itself to a cache timing attack. Such implementations, which are immune to this attack were taken and executed on both REDEFINE and GPP platforms. The results are shown in **Table 4**. The simulation was performed on a desktop with Core 2 Duo processor (45 nm) running at 3 GHz. The clock cycles were determined with appropriate assembly level instrumentation. Care was taken not to include the input output delays in the measured time. On the GPP, AES-128 Encryption executed in 327,901 cycles amounting to 109.3 μ seconds and AES-128 decryption completed in 481,351 amounting to 160.45 μ seconds. For REDEFINE the cycle count was obtained on the SystemC simulator, as indicated previously. The frequency for REDEFINE is set to 400 MHz. This is limited by the frequency of the ultra-high speed Faraday memory used on REDEFINE.

Table 4 Comparison of the performance of AES encryption and decryption on REDEFINE with a GPP.

Application	Cycle Count on REDEFINE	Time (in μ secs) on REDEFINE	Improvement Over GPP
AES-128 Encryption (without CFU)	11070	27.675	3.95×
AES-128 Encryption (with CFU)	5632	14.08	7.76×
AES-128 Encryption (with CFU + Centrality)	4807	12.02	9.09×
AES-128 Decryption (without CFU)	12677	31.69	5.06×
AES-128 Encryption (with CFU)	5831	14.58	11×
AES-128 Decryption (with CFU + Centrality)	4848	12.12	13.23×

As can be observed from Table 4, AES-128 Encryption runs 7.76× with a CFU and 9.09× faster with centrality based partitioning technique. In the case of AES-128 Decryption when centrality based partitioning was used the perfomance was 13.23× faster than GPP, as opposed to 11× faster with CFU.

5. Conclusion

Higher degree of spatial computation exploited in CGRAs necessitate spatial partitioning of instructions. Partitioning involves maintaining: (1) a balance between decreasing instruction execution time through ILP exploitation and reducing the communication time, by placing dependent instructions closer. (2) a balance between reducing reconfiguration overhead through better use of available bandwidth while not incurring higher control overheads and (3) choice between balanced partitions and unbalanced partitions. We presented an extension of edge betweenness centrality scheme for partitioning the dataflow graph, which achieved the lowest execution time through reduction in both instruction execution time and reconfiguration overhead. The execution time is improved in the range of 6–20%, which is significant in the context of the highly optimized REDEFINE compiler. Our results indicate that this scheme performs almost always better than several other schemes viz. Kernighan-Lin and parent-affinity based

schemes. However, we observed for certain applications whose execution time and reconfiguration time are almost equal do not perform well with centrality based scheme. This is due to the use of unbalanced partitions. Unbalanced partitions reduce the perceived reconfiguration time by overlapping it with instruction execution. However, it also tends to stretch the total reconfiguration time. We compared the execution time for AES encryption and decryption kernels on REDEFINE with a general purpose processor and observe that use of centrality based partitioning algorithm helps improve the execution time substantially.

References

1) Alle, M., Varadarajan, K., Fell, A., Reddy C., R., Joseph, N., Das, S., Biswas, P., Chetia, J., Rao, A., Nandy, S.K. and Narayan, R.: REDEFINE: Runtime reconfigurable polymorphic ASIC, *ACM Trans. Embed. Comput. Syst.*, Vol.9, No.2, pp.1–48 (2009).
2) Alle, M., Varadarajan, K., Fell, A., Nandy, S.K. and Narayan, R.: Compiling Techniques for Coarse Grained Runtime Reconfigurable Architectures, *Proc. 5th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, Berlin, Heidelberg, pp.204–215, Springer-Verlag (2009).
3) Bobda, C.: *Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications*, Springer Publishing Company, Incorporated (2007).
4) Chang, D. and Marek-Sadowska, M.: Partitioning Sequential Circuits on Dynamically Reconfigurable FPGAs, *IEEE Trans. Comput.*, Vol.48, No.6, pp.565–578 (1999).
5) Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *CGO '04: Proc. International Symposium on Code Generation and Optimization*, Washington, DC, USA, p.75, IEEE Computer Society (2004).
6) DeHon, A. and Wawrzynek, J.: Reconfigurable computing: what, why, and implications for design automation, *DAC '99: Proc. 36th Annual ACM/IEEE Design Automation Conference*, New York, NY, USA, pp.610–615, ACM (1999).
7) Fell, A., Alle, M., Varadarajan, K., Biswas, P., Das, S., Chetia, J., Nandy, S.K. and Narayan, R.: Streaming FFT on REDEFINE-v2: An application-architecture design space exploration, *Proc. 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, *CASES '09*, New York, NY, USA, pp.127–136, ACM (2009).
8) Fell, A., Chetia, J., Biswas, P., Narayan, R. and Nandy, S.K.: Generic Routing Rules and a Scalable Access Enhancement for the Network-on-Chip RECONNECT, *IEEE International SOC Conference* (*SoCC*) (2009).
9) Gajjala Purna, K.M. and Bhatia, D.: Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers, *IEEE Trans. Comput.*, Vol.48, No.6, pp.579–590 (1999).
10) Girvan, M. and Newman, M.E.J.: Community structure in social and biological networks, *Proc. National Academy of Sciences of the United States of America*, Vol.99, No.12, pp.7821–7826 (2002).
11) Hartenstein, R.: A decade of reconfigurable computing: A visionary retrospective, *Proc. Design, Automation and Test in Europe* (*DATE*), pp.642–649 (2000).
12) Kernighan, B.W. and Lin, S.: An Efficient Heuristic Procedure for Partitioning Graphs, *The Bell System Technical Journal*, Vol.49, No.1, pp.291–307 (1970).
13) Krishnamoorthy, R., Varadarajan, K., Garga, G., Alle, M., Nandy, S.K., Narayan, R. and Fujita, M.: Towards Minimizing Execution Delays on Dynamically Reconfigurable Processors: A case study on REDEFINE, *Proc. 2010 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, *CASES '10*, New York, NY, USA, ACM (2010).
14) Lee, W., Barua, R., Frank, M., Srikrishna, D., Babb, J., Sarkar, V. and Amarasinghe, S.: Space-time scheduling of instruction-level parallelism on a raw machine, *ACM SIGOPS Operating Systems Review*, Vol.32, No.5, pp.46–57 (1998).
15) Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E. and Bringmann, R.A.: Effective Compiler Support for Predicated Execution Using the Hyperblock, *MICRO 25: Proc. 25th Annual International Symposium on Microarchitecture*, Portland, Oregon, pp.45–54, IEEE Computer Society TC-MICRO and ACM SIGMICRO (1992).
16) Pandey, A. and Vemuri, R.: Combined Temporal Partitioning and Scheduling for Reconfigurable Architectures, *SPIE Conference on Configurable Computing: Technology and Applications* (1999).
17) Paulin, P.G. and Knight, J.P.: Force-directed scheduling in automatic data path synthesis, *DAC '87: Proc. 24th ACM/IEEE Design Automation Conference*, New York, NY, USA, pp.195–202, ACM (1987).
18) Paulin, P.G. and Knight, J.P.: Scheduling and binding algorithms for high-level synthesis, *DAC '89: Proc. 26th ACM/IEEE Design Automation Conference*, New York, NY, USA, pp.1–6, ACM (1989).
19) Sankaralingam, K., Nagarajan, R., Liu, H., Kim, C., Huh, J., Burger, D., Keckler, S.W. and Moore, C.R.: Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture, *Proc. 30th Ann. Intl Symp. on Computer Architecture* (*30th ISCA 2003*), *FCRC'03, ACM Computer Architecture News* (*CAN*), San Diego, CA, pp.422–432, ACM SIGARCH/IEEE CS (2003). Published as *Proc. 30th Ann. Intl Symp. on Computer Architecture* (*30th ISCA 2003*), *FCRC'03*, *ACM Computer Architecture News* (*CAN*), Vol.31, No.2, UT Austin.
20) Swanson, S., Schwerin, A., Mercaldi, M., Petersen, A., Putnam, A., Michelson, K., Oskin, M. and Eggers, S.J.: The WaveScalar architecture, *ACM Trans. Comput. Syst.*, Vol.25, No.2, pp.1–54 (2007).
21) Taylor, M.B., Lee, J.-w., Seneski, M. and Frank, M.: The RAW Microprocessor:

A Computational Fabric for Software Circuits and General-Purpose programs Microprocessor, *Ieee Micro*, pp.25–35 (2002).
22) Trimberger, S.: Scheduling designs into a time-multiplexed FPGA, *FPGA '98: Proc. 1998 ACM/SIGDA 6th International Symposium on Field Programmable Gate Arrays*, New York, NY, USA, pp.153–160, ACM (1998).
23) Yang, H. and Wong, D.F.: Efficient network flow based min-cut balanced partitioning, *ICCAD '94: Proc. 1994 IEEE/ACM International Conference on Computer-aided design*, Los Alamitos, CA, USA, pp.50–55, IEEE Computer Society Press (1994).

(Received November 29, 2010)
(Revised March 5, 2011)
(Accepted April 24, 2011)
(Released August 10, 2011)

(Recommended by Associate Editor: *Yuichi Nakamura*)

Ratna Krishnamoorthy received her B.E. degree from Swami Ramanand Teerth Marathwada University, Nanded, India. She received her M.E. degree in VLSI Design from Anna University, Chennai, India. She is currently a Ph.D. student in Department of Electronics Engineering at the University of Tokyo. Her research interests include Reconfigurable Computing, Algorithms and High Level Synthesis.

Saptarsi Das received his B.E. degree from Jadavpur University, Kolkata, India in 2007. He is pursuing his M.Sc (Engg.) at Centre for Electronic Design and Technology, Indian Institute of Science, Bangalore. His primary research interests include Computer Architecture, Field Arithmetic and Cryptography.

Keshavan Varadarajan received his B.E. degree from VTU, Belgaum, India in 2002. He is currently a Ph.D. student at the Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore. His research interests include Computer Architecture and Compilers.

Mythri Alle received her B.E. degree from JNTU, Hyderabad, India in 2002 and her M.E. degree from the Computer Science and Automation Department, Indian Institute of Science, Bangalore in 2004. She is currently a Ph.D. student at the Supercomputer Education and Research Centre, Indian Institute of Science. Her research interests include Compilers and Media Processing.

Masahiro Fujita received his B.S. degree in electrical engineering in 1980, and M.S. and Ph.D. degrees in information engineering from the University of Tokyo, in 1982 and 1985, respectively. From 1985 to 1993, he was a Research Scientist at Fujitsu Laboratories, Japan and in 1994 became the Director of the Advanced Computer Aided Design Research Group, Fujitsu Laboratories of America. He is currently a Professor at the Department of Electrical Engineering, University of Tokyo. His research interests include computer-aided design of digital systems. He has received the Sakai Award from IPSJ in 1984.

S K Nandy received his Bachelors degree in Physics (Honors) from the Indian Institute of Technology, Kharagpur in 1977. He received his Bachelors degree in Electrical Communications Engineering in 1980, and M.Sc (Engg.) and Ph.D. degree in Computer Science and Automation in 1986 and 1989 respectively from the Indian Institute of Science, Bangalore. His current research interests include VLSI Systems and Runtime Reconfigurable System-on-Chip platforms. He is a Professor at the Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore.

Ranjani Narayan received her Bachelor's degree in Science in 1977. She received her Bachelor's in Electrical Engineering in 1980 and received Ph.D. degree in Electrical Engineering in 1989 from Indian Institute of Science, Bangalore. She has over 5 years experience at Indian Institute of Science and 9 years at Hewlett Packard where she worked on varied topics ranging from Operating Systems to Hardware Monitoring and Diagnostics. Since 2006, she is the Chief Technology Officer of Morphing Machines, Bangalore. Morphing Machines is incubated at the Indian Institute of Science engaged in the design and development of reconfigurable silicon cores. Her research interests include Reconfigurable Computing and dataflow architectures.