Network Topology Exploration of Mesh-Based Coarse-Grain Reconfigurable Architectures

Nikhil Bansal[†] Sumit Gupta[†] Nikil Dutt[†] Alex Nicolau[†] Rajesh Gupta[§]

CECS
Technical Report # 03-27
August 2003

Center for Embedded Computer Systems

[‡]Dept. of Information and Computer Science
University of California at Irvine
{nbansal, sumitg, dutt, nicolau}@cecs.uci.edu

[§]Dept. of Computer Science and Engineering
University of California at San Diego
gupta@cs.ucsd.edu

Abstract

Several coarse-grain reconfigurable architectures proposed recently consist of a large number of processing elements (PEs) connected in a mesh-like network topology. We study the effects of three aspects of network topology exploration on the performance of applications on these architectures: (a) changing the interconnection between PEs, (b) changing the way the network topology is traversed while mapping operations to the PEs, and (c) changing the communication delays on the interconnects between PEs. We propose network topology traversal strategies that first schedule PEs that are spatially close and that have more interconnections among them. We use an interconnect aware list scheduling heuristic as a vehicle to perform the network topology exploration experiments on a set of designs derived from DSP applications. Our experimental results show that a spiral traversal strategy, coupled with a two neighbor interconnect topology leads to good performance for the DSP benchmarks considered. Our prototype framework thus provides an exploration environment for system architects to explore and tune coarse-grain reconfigurable architectures for particular application domains.

Contents

List of Figures

List of Tables

1 Introduction

Reconfigurable fabrics have emerged as an important bridge in the gap between ASICs and microprocessors. They merge the performance of ASICs with the flexibility of microprocessors. Coarse-grain reconfigurable architectures trade-off some of the configuration flexibility of fine-grain FPGAs in return for smaller delay, area and configuration time. They provide massive parallelism, high computational capability and their behavior can be configured dynamically, thus making them a better alternative to ASICs and fine-grain FPGAs in many aspects. As a result, we have seen the emergence of a wide range of coarse-grain reconfigurable architectures over recent years [1, 2, 3, 4, 5, 6, 7, 8, 9].

We focus on a set of these architectures that consist of *processing elements* (PEs) or arithmetic logic units (ALUs) connected together by a mesh-like network [5, 7, 8]. This is a popular architecture model which is simple in construction and scalable due to the ability to add more PEs to the mesh (or more grids of PEs connected by buses). We focus on mapping the time consuming and data-intensive loops of a class of DSP applications to these coarse-grain architectures. The ample resources available in coarse-grain architectures can be used to exploit the parallelism in, and thus accelerate, the loops in these applications.

Mapping applications to such architectures is a complex task that is a combination of the traditional operation scheduling, operation to PE binding (or mapping), and routing problems. Indeed, we believe that the network topology (interconnections among PEs) and communication delays on these interconnects are critical concerns for good mapping of applications on these architectures. Also, different network and interconnect topologies offer a wide design space and it is not clear which topologies perform best for a given class of applications.

In this paper, we explore the effects of varying the network topologies, the topology traversal strategies, and the delay models for the interconnects on the quality of performance results for applications mapped to these mesh-based coarse-grain reconfigurable architectures. We employ an operation to PE mapping technique that exploits temporal locality between operations by mapping operations with data dependencies on spatially close PEs in order to minimize the data transfer delays. We present a list scheduling heuristic that simultaneously considers routing of data between operations and present scheduling results for a set of designs derived from DSP applications.

The rest of the paper is organized as follows. Section 2 outlines related work. Next, we describe three important aspects of network topologies. We present our list scheduling heuristic in Section 4 and in Section 5, we present our experimental setup and results. Section 6 concludes the paper.

2 Related Work

Recently, several coarse-grain reconfigurable architectures have been proposed (e.g., [1, 2, 3, 4, 5]) and some have addressed analysis and exploration for their architectures. Mortiz et al. [10] presented a framework that produces an optimal RAW microprocessor structure [3] under cost and area balance constraints for a given application. Nageldinger [11] proposed the *KressArray Xplorer* for KressArray architectures [1] to find the best

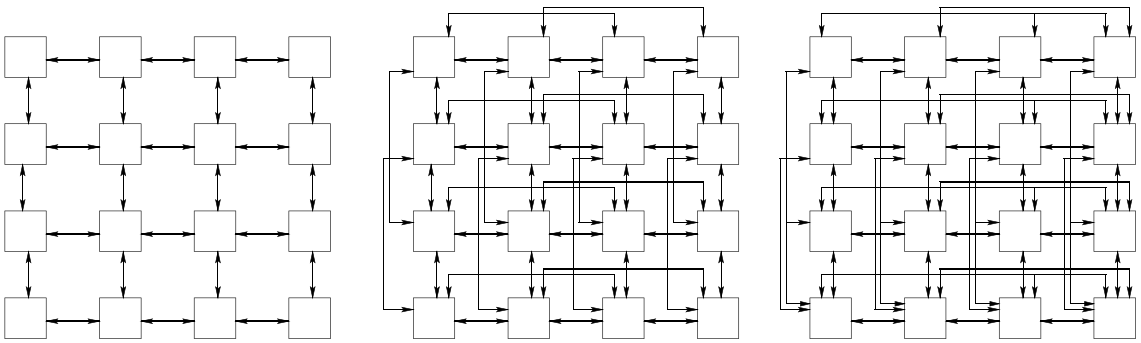

Figure 1. **Different Connection Topologies**

trade-off between the complexity of the hardware and an optimization objective such as performance. Bossuet et al. [12] proposed a framework that performs application profiling and performance estimation on a range of coarse-grain architectures.

In the context of application mapping, Venkataramani et al. [13] presented a compiler framework for mapping loops written in SA-C language to the MorphoSys architecture [5]. Mei et al. [14] proposed a modulo loop scheduling approach to map loops on a generic reconfigurable architecture. Lee et al. [15] addressed memory bandwidth and interconnection issues while mapping algorithms to a generic reconfigurable architecture called the Dynamically reconfigurable ALU array (DRAA). Note that, there is also a body of prior work on mapping applications to systolic arrays [16].

Our work differs from, and complements previous efforts by examining the effects of different network interconnection topologies, different topology traversal strategies, and different communication delays (again related to network topology) on system performance.

3 Network Topology Exploration

As stated earlier, we target coarse-grain reconfigurable architectures that consist of a large number of processing elements (PEs) connected together in a 2-D array or mesh. PEs can only be connected to PEs in the same row or column. An example of such an architecture is shown in Figure 1(a). In this figure, each square box represents a processing element (or ALU) and the double headed arrows denote data communication links between PEs. We call these links between PEs as *direct interconnects* and we denote such a tightly connected network of PEs as a *grid.* Multiple such grids of PEs may be connected together using system buses in a *matrix* of grids. This is similar to several coarse-grain architectures that have been proposed recently [5, 7, 8].

In our application mapping framework, we support a family of such coarse-grain architectures by allowing the designer to vary: (a) the number of PEs in each grid, (b) the number of grids in a matrix, (c) the network topology or interconnections between the PEs within a grid, (d) the communication penalties on the various interconnects. All the PEs in the architecture template are considered to be identical and comprise of exactly same functional units. However, we allow the designer to specify the configuration of a PE, in terms of the type and number of

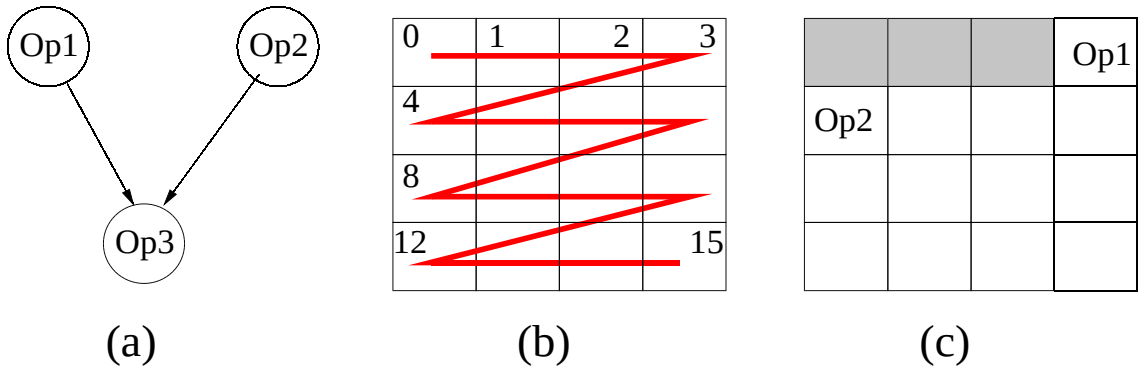

Figure 2. **(a) Sample DFG (b) Zig-zag topology traversal (c)** Op_3 **suffers a communication delay in this mapping**

functional units existing in a PE, the operations that can be executed on the units, and the execution delay of each unit.

3.1 Different Network Topologies

The architectures shown in Figure 1 illustrate the range of different network topologies we support within a grid. Figure 1(a) shows a grid in which PEs are connected to their immediate neighbors in the same row and same column. In the grid of Figure 1(b), all the PEs are connected to their immediate and *1-hop* neighbors, i.e., the neighbors that can be reached by traversing through one other PE. Similarly, in Figure 1(c), PEs are connected to all other PEs in the same row and same column.

Grids can in turn be connected to each other by system buses to form a matrix. For example, in the MorphoSys architecture [5], there are four grids each of size 4x4 (i.e. each grid has 16 PEs) forming a 2x2 matrix. System row buses connect PEs in the same row of different grids and likewise for shared column buses.

3.2 Different Topology Traversal Strategies

Topology traversal, the order in which PEs are traversed, is another important issue affecting the quality of mapping results. Consider that we want to map the sample DFG shown in Figure 2(a) to the architecture shown earlier in Figure 1(a). One traversal order of PEs is shown in Figure 2(b), wherein PEs are traversed in a *zig-zag* manner starting from the PE in the top left corner of the grid. Consider now that Figure 2(c) represents the current state of our mapping. Shaded boxes (processing elements PE_0 to PE_2) indicate that the PE already has an operation mapped on it in the current cycle. Note that, we number the PEs from PE_0 to PE_{15} from the top left corner down to the bottom right corner (corresponding to the zig-zag).

If we map Op_1 to PE_3 and Op_2 to PE_4 as shown in Figure 2(c), then we cannot schedule Op_3 to execute in the next cycle. This is because the results from operations Op_1 and Op_2 cannot reach any PE in the next cycle. Hence, Op_3 will suffer a communication delay of one cycle. However, if we map Op_2 to PE_7, then we can map Op_3 to either PE_3 or PE_7.

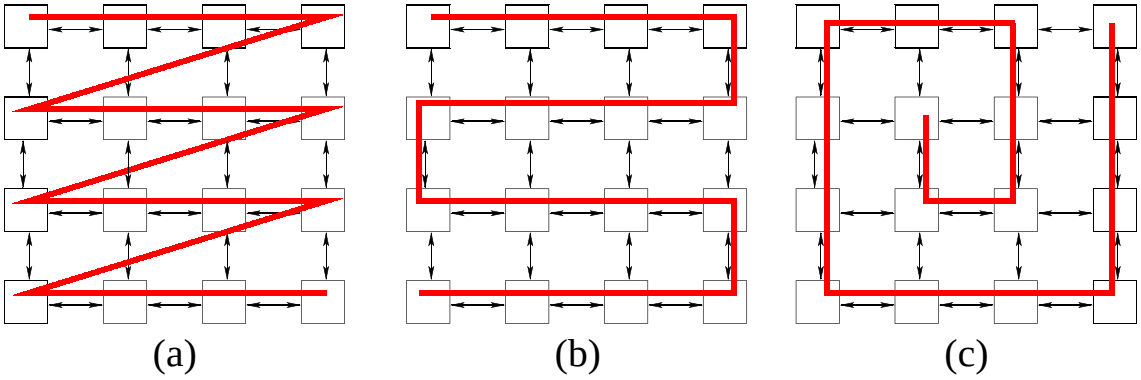

Figure 3. **Different network traversal strategies: (a) Zig-zag, (b) Reverse-S, (c) Spiral.**

One drawback of zig-zag traversal is that after completing the mapping of a row, the next PE traversed is not adjacent to the previous PE. To overcome this limitation, we propose two other topology traversal strategies:

- *Reverse-S traversal*: As shown in Figure 3(b), this strategy traverses the network in a reverse-S manner. This strategy always traverses spatially adjacent PEs.
- *Spiral traversal*: An improvement over the reverse-S strategy is to traverse the grid in a spiral manner starting with the PE(s) at the center of the grid, as shown in Figure 3(c). Thus, this strategy first maps operations to the central PEs that have more adjacent neighbors than the PEs at the edges.

We study the effect of these topology traversal strategies on system/application performance in Section 5.4.

3.3 Different Communication Delays

We enable the designer to specify (a) the communication delay on direct connections between PEs, (b) delay for 1-hop communication (i.e., communication through another PE), and (c) delays on shared system buses connecting PEs across grids. This models a range of coarse-grain architectures. For example, MorphoSys [5] and MATRIX [17] have zero communication delay between PEs and one cycle for shared bus communication. We believe that interconnect speeds will begin to trail computation delays as technology improves and thus, we experiment with different communication models in Section 5.5.

4 Mapping Applications on to Coarse-Grain Reconfigurable Architectures

Operation to PE mapping in coarse-grain architectures is a combination of traditional operation scheduling, resource binding, and data routing problems [18] as discussed next.

4.1 Interconnect aware *Op* to *PE* mapping

Consider the example DFG shown in Figure 4(a). Let us say that we have to map this application to the coarse-grain architecture shown in Figure 4(b). Consider also that there is no communication delay on direct interconnects between PEs. For simplicity, in all our examples we show that operations are mapped on different PEs. In practice, our scheduler maps two operations on the same PE provided their execution times do not overlap.

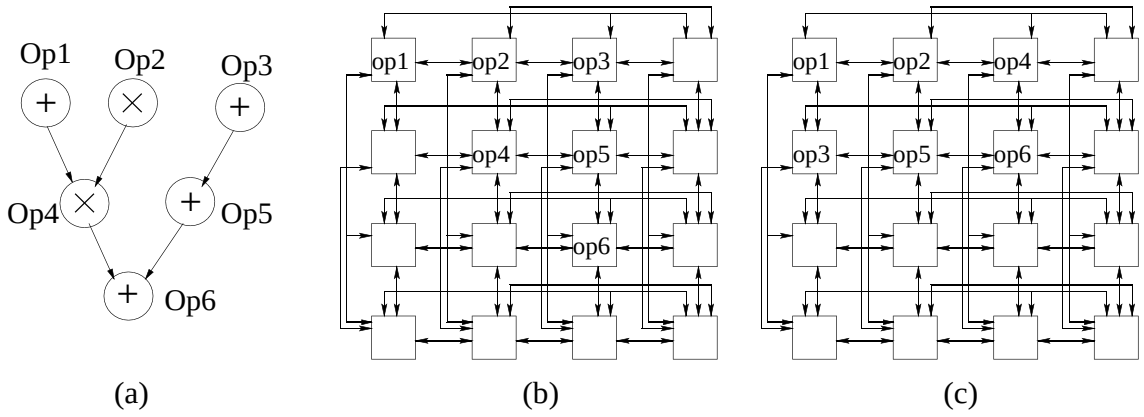

Figure 4. **(a) Sample Data Flow Graph (b) Mapping with penalty of 2 cycles (c) Mapping with no penalty**

Assume that a multiplication takes 2 cycles and an addition takes 1 cycle. Then this DFG, in the best case, should take 5 cycles to execute (the sequence of operations Op_2, Op_4, and Op_6). However, consider the mapping shown in Figure 4(b). The total execution time for this mapping is 7 cycles since the data from Op_1 takes 1 cycle to reach Op_4 and data from Op_4 takes 1 more cycle to reach Op_6. Thus, we have to take the data communication overheads between operations into consideration during operation to PE mapping. In Figure 4(c), we show one possible mapping in which there is no communication overhead and the design takes only 5 cycles to execute.

Hence, the *total run time* of an application includes the *execution time* of the operations and the *routing delay* for the corresponding operands. The goal of our scheduler is to minimize the total run time of the application and hence, to map operations such that routing delay is minimized.

4.2 Implementation in a List Scheduling Heuristic

To perform our network topology exploration experiments, we implemented our techniques in a list scheduling algorithm [18]. However, our strategies are independent of the scheduling heuristic and can be used in other heuristics such as force-directed scheduling as well.

Our list scheduling heuristic uses interconnect information (connectivity and delays) between PEs and attempts to map operations with data dependencies on spatially close PEs in order to exploit the inherent parallelism. The scheduler traverses the control-data flow graph of the application and schedules one basic block at a time. Currently, we do not support speculation, predication and do not take memory bus bandwidth into consideration [15].

The heuristic for scheduling a basic block, *ScheduleBB*, is listed in Figure 5(a). This heuristic takes as input *PEList*, the list of all the PEs in the architecture. The PEs in *PEList* are ordered based on the network traversal strategy as specified by the user. A global clock cycle *currCycle* is maintained during scheduling.

The *ScheduleBB* heuristic starts by collecting a list of available or ready operations, A. *Available operations* are operations whose data dependencies are satisfied and can be scheduled in the current cycle. The heuristic then schedules the PEs starting from the front of the *PEList*. We first make a copy of the available list as A_{currPE} for

```
/* Schedules operations in basic block currBB */
ScheduleBB(currBB, PEList, currCycle)

1: A ← List of all available operations in currBB
2: while (A ≠ ϕ) {
3:    foreach (currPE ∈ PEList) {
4:       A_CurrPE ← A
5:       while (A_CurrPE ≠ ϕ) {
6:          Pick candOp ∈ A_CurrPE with highest priority
7:          A_CurrPE ← A_CurrPE - candOp
8:          if ( IsRoutable(candOp, currPE, currCycle) ) {
9:             A ← A - candOp
10:            Schedule candOp on currPE in currCycle
11:            A_CurrPE ← ϕ   /* Exit while(A_CurrPE) loop */
12:         }   /* end if */
13:      }   /* end while */
14:   }   /* end foreach */
15:   currCycle ← currCycle + 1
16: }   /* end while */          (a)
```

```
/* Verifies routability of candOp on currPE*/
IsRoutable(candOp, currPE, currCycle)

1: foreach (predOp ∈ PREDs(Op_i)) {
2:    predPE ← PE on which predOp is mapped
3:    foreach (path ∈ PATHs(predPE, currPE)) {
4:       if (currCycle < EndTime(predOp) - 1 + Delay(path))
5:          or (PathNotAvailable(path, currCycle))
6:             return false
7:    }   /* end of foreach */
8: }   /* end of foreach */
9: return true                (b)
```

Figure 5. **(a) Algorithm to schedule a basic block (b) Algorithm for validating the interconnect delays.**

currPE (lines 3 and 4 in Figure 5(a)). Next, the heuristic chooses the operation (*candOp*) with the highest priority from A_{currPE}. The *priority* of an operation is calculated as one more than the maximum of the priorities of all the operations that use its result. Operations whose results are not read (i.e., primary outputs) have a priority of one. Thus, we give preference to operations on the longest data dependency (critical) paths.

The scheduler calls the *IsRoutable* function to verify the ability to route data to *candOp*. This function, outlined in Figure 5(b), checks if the data from all the predecessors of *candOp* (operations whose results *candOp* reads) are available at *currPE* in *currCycle*. Thus, the *IsRoutable* function checks all the paths from *predPE* (on which the predecessor operation is mapped) to *currPE* by calling the function *PATHs* (lines 2 and 3 in Figure 5(b)). These paths and the delays on them are determined statically before scheduling.

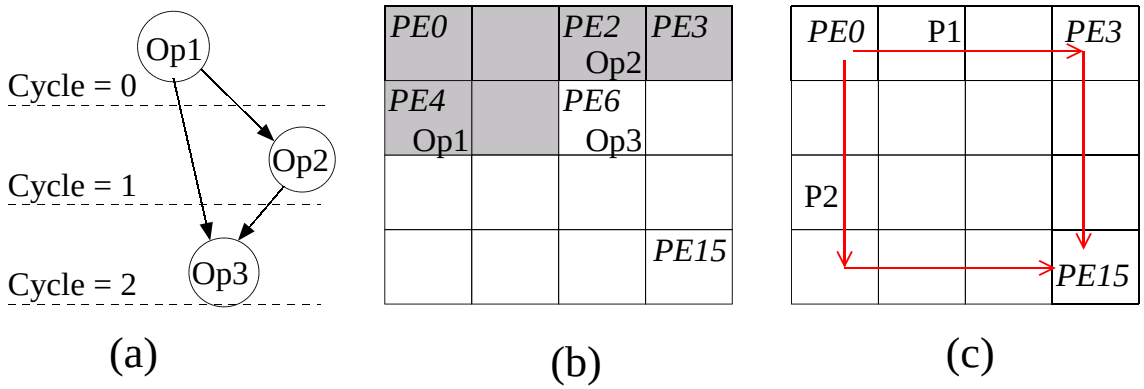

Figure 6. **(a) Example of a small DFG. (b) Validating the ability to route data. (c) Showing Paths between PEs**

There are two situations in which we cannot use a path from *predPE* to *currPE*: either the cycle in which the predecessor operation finishes execution ($EndTime(predOp)$ - 1) summed with the delay of the path ($Delay(path)$) is larger than the current cycle (*currCycle*), or if the path is not available, i.e., any connection on the path is used by another data communication in the current cycle. This is given in lines 4 to 6 of the algorithm in Figure 5(b).

If the *IsRoutable* function finds no path for data to reach *currPE* in *currCycle*, then the *ScheduleBB* considers the next operation in A_{CurrPE}, till all the operations are exhausted. If *IsRoutable* returns a true result, the *candOp* is mapped on *currPE* and scheduled to execute in *currCycle* (lines 7 to 11 in Figure 5(a)). Usage information for all paths required for data transfers is updated. In this way, the *ScheduleBB* heuristic schedules operations on each PE in *PEList* and then increments *currCycle* when *PEList* is exhausted. This process is continued until all the available operations in the current basic block have been scheduled.

The example in Figure 6 illustrates the logic behind the *IsRoutable* function. If we map Op_1 and Op_2 to PE_4 and PE_2 as shown in Figure 6(b), then to map Op_3 on PE_6, we have to route the result of Op_1 to PE_6 from PE_4. If the communication delay from PE_4 to PE_6 is one cycle and Op_1 executes in one cycle (it starts execution in cycle 0), then we can schedule operation Op_3 in cycle 2 and map it on PE_6.

While determining the paths from one PE to another, we only consider the most direct (and shortest) paths among PEs. Figure 6(c) shows the most direct path, P1, from PE_0 to PE_3 (since these are in the same row). In contrast, there are two paths (P1 and P2) between PE_0 and PE_{15}.

5 Experimental Setup and Results

In order to perform the network topology exploration, we implemented our techniques and the mapping and scheduling algorithm in a prototype compiler framework. This framework accepts an application code in C and applies basic compiler transformations such as copy propagation and dead code elimination. In this section, we present results for experiments by varying the different network topology parameters.

We modeled six different architecture configurations listed in Table 1. The name of each configuration (column 1) is represented as a 4 digit number (RCDG) where each digit signifies an architecture parameter: *R* and *C*

Config Name RCDN	Grid Size RxC	Direct Connects D	Num of Grids N	Resembling Architecture
4414	4x4	1	4	DReAM [8]
4424	4x4	2	4	
4434	4x4	3	4	MorphoSys [5]
8811	8x8	1	1	REMARC [7]
8821	8x8	2	1	
8831	8x8	3	1	

Table 1. **Characteristics of Different Architectures**

Design	Ops	Cycles	IPC	Utilization
FFT	286	76	4.26	6.67%
ATR	508	75	8.47	13.23%
Laplace	608	22	30.40	47.50%
Sor	630	93	7.59	11.86%
Lowpass	652	105	7.85	12.27%
PDE	463	81	5.71	8.93%
Predictor	618	102	6.06	9.47%
Hydro	1290	37	36.85	57.59%

Table 2. **Scheduling results for the eight designs on 4414 configuration with zig-zag traversal**

represent the number of **r**ows and **c**olumns in the grid, *D* is the number of **d**irect connections each PE has, and *G* is the number of **g**rids in the configuration. The last column lists the architectures that these configurations resemble. The number of PEs in every configuration is 64. Hence, configurations 4414, 4424, and 4434 consist of four grids of the architectures shown earlier in Figure 1(a), (b), and (c) respectively.

We performed the experiments with two different communication delay models:

- *Delay Model DM*0: Direct connection delay is zero. 1-hop communication through another PE and inter-grid shared buses take one cycle (corresponds to [5, 17]).
- *Delay Model DM*1: Direct connection delay is one cycle. 1-hop communication and shared buses take two cycles.

In both of these models, only one pair of PEs can use a shared bus at a time. We take the simplest configuration as the base case for all our experiments: 4414 configuration with zig-zag topology traversal strategy.

5.1 Characteristics of Benchmarks

We used a set of eight designs drawn from the DSP domain for our experiments. To give some insight into the characteristics of these designs, we present scheduling results for the designs on base case in Table 2. The columns in this table list the name of the design, the number of operations in the design, the number of cycles it takes to execute on the 4414 configuration, the instructions per cycle (IPC) it achieves, and the percentage utilization of the PEs in the matrix. The typical run time of our scheduler is about 15 user seconds on a 400 Mhz UltraSparc-II.

From the results in Table 2, we see that designs such as Laplace and Hydro achieve a relatively high IPC. This is because there is significant instruction level parallelism in these designs and few or no inter-iteration dependencies

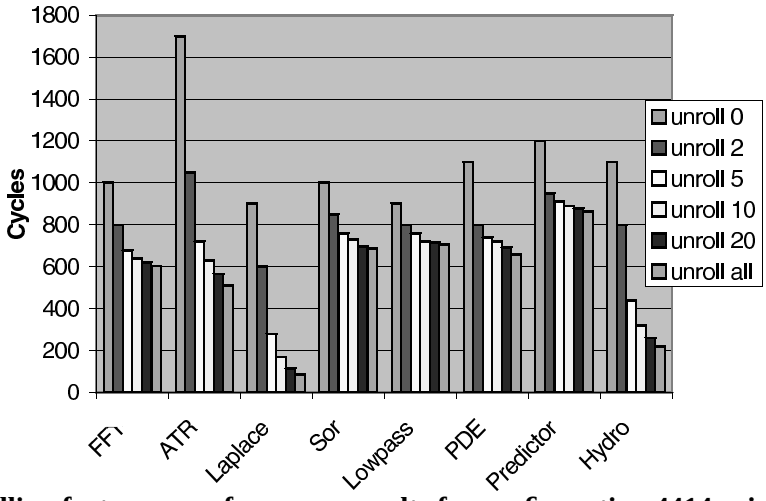

Figure 7. **Effect of unrolling factor on performance results for configuration 4414 using zig-zag traversal**

– so several iterations can execute in parallel. In contrast, other designs have high inter-iteration *read after write* data dependencies. As a result, these designs are not able to fully utilize the PEs available, thus, leading to poor performance.

5.2 Effect of Unrolling Factor

In order to expose the parallelism of the application, we unroll the loops that increase the number of operations to map. In case of nested loops, we unroll the innermost loop. Figure 7 shows the effect of varying the unrolling factor for the base configuration (RCDG = 4414). We got similar results for all the other configurations as well.

The results in Figure 7 demonstrate that the performance improves rapidly as the unrolling factor is increased from 0 to 10. This is because the opportunities to extract parallelism increase as the number of available operations increase due to unrolling. When we unroll the loop further, there is not much improvement for all the designs except Laplace and Hydro. This is because in all other designs, most of the new operations generated due to unrolling are dependent on some operations from previous iterations. The improvement up to unrolling factor of 10 is due to early scheduling of non-dependent operations. In the designs where there are less inter-iteration RAW data dependencies (Laplace and Hydro), unrolling keeps improving the performance. Since an unrolling factor of 10 gives substantial ILP, for the rest of the experiments, we unroll the loops in the designs by 10.

5.3 Effect of Varying Configurations

Table 3 compares the number of cycles each design takes to execute on the different architecture configurations using zig-zag topology traversal. This table also shows the percentage reduction in cycles on 4434 configuration over 4414 configuration.

The results in this table demonstrate that the performance improves significantly as the number of direct connections increase from 1 to 2 (for example from configuration 4414 to 4424). A higher number of direct connections increases the opportunity to map the dependent operations without incurring any communication penalty.

No. of Cycles for Different Configurations								
Design	4414	4424	4434	Total Reduc.	8811	8821	8831	Total Reduc.
FFT	76	67	67	11.8 %	74	67	67	9.4 %
ATR	75	69	69	8.0 %	74	68	66	10.8 %
Laplace	22	20	20	10.0 %	20	18	17	15.0 %
Sor	93	83	83	10.7 %	94	85	83	10.1 %
Lowpass	105	97	87	17.14 %	99	94	85	14.1 %
PDE	81	72	72	11.1 %	78	71	67	14.4 %
Predictor	102	93	93	8.8 %	100	91	89	11.0 %
Hydro	37	35	35	5.3 %	36	35	34	5.5 %

Table 3. **Delay model *DM*0: Performance comparison for different configurations with zig-zag traversal.**

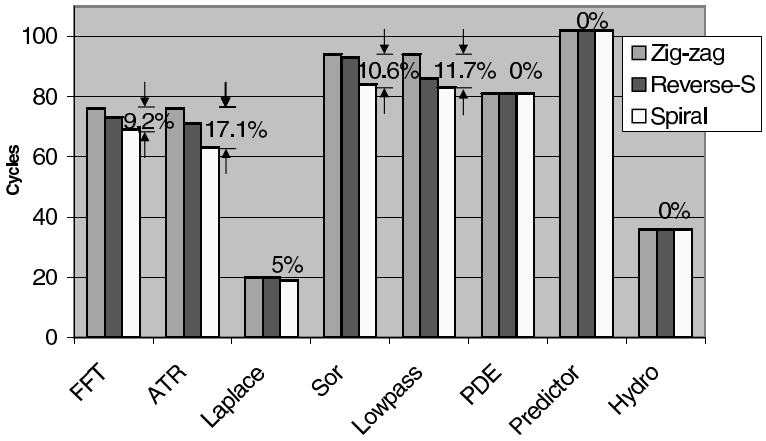

Figure 8. **Delay model *DM*0: Effect of Network Traversal Strategy on performance for configuration 4414**

In contrast, there is almost no performance improvement when number of direct connections increase from 2 to 3 (configuration 4424 to 4434). This is because, when we change the configuration from 4424 to 4434, the connectivity improves only for the PEs at the corner of each grid since remaining PEs are already connected to other PEs in the same row and column (because of smaller grid size). However, for the larger grids, connectivity improves for all the PEs. As a result performance does improve from configuration 8821 to 8831 for some designs. These experiments show that a network topology in which each PE has two connections to other PEs in the same row and column is sufficient to exploit the ILP for these designs.

5.4 Effect of Topology Traversal Strategy

In this section, we show the effect of the different topology traversal strategies discussed in Section 5.4 on the scheduling results (cycles). Figure 8 shows the scheduling results for three traversal strategies for the base configuration (RCDG = 4414). The bars in these graphs represent the results with the zig-zag traversal (first bar), reverse-S traversal (second bar), and spiral traversal (third bar).

From this figure, we can see that *reverse-S* traversal gives modest improvements in some cases. The largest improvements in performance (cycles) are achieved using *spiral traversal* (up to 17 % for ATR over zig-zag traversal). We found that this is because mapping operations first to the PEs at the center of the grid – that are more well-connected than PEs at the edges – enables more opportunities to schedule the dependent operations.

No. of Cycles for Different Configurations								
Design	4414	4424	4434	Total Reduc.	8811	8821	8831	Total Reduc.
FFT	133	123	117	12.0 %	130	124	117	10.0 %
ATR	102	97	96	5.8 %	99	99	96	3.0 %
Laplace	26	25	23	11.5 %	24	23	21	12.5 %
Sor	121	112	107	11.6 %	119	112	105	11.8 %
Lowpass	147	133	117	20.4 %	137	125	117	14.6 %
PDE	101	92	91	9.9 %	100	94	91	9.0 %
Predictor	143	129	129	9.8 %	138	128	122	11.6 %
Hydro	45	42	42	6.7 %	45	42	41	8.9 %

Table 4. **Delay model *DM*1: Performance comparison for different configurations with zig-zag traversal.**

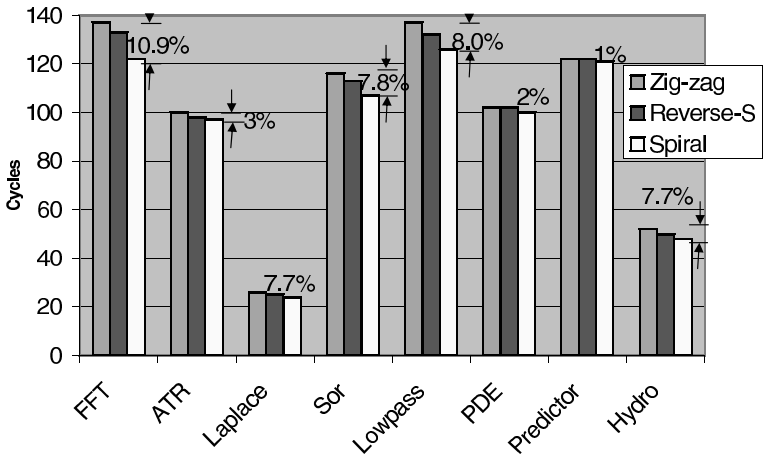

Figure 9. **Delay model *DM*1: Effect of Network Traversal Strategy on performance for configuration 4414**

5.5 Effect of Delay Model

In this section, we present the results corresponding to delay model *DM*1 and compare them with the results shown in previous sections. Recall that in this model, delay between adjacent PEs is one cycle and delay on buses is two cycles. Table 4 shows the performance of various architecture configurations with this delay model using zig-zag topology traversal and Figure 9 shows the effect of different topology traversal strategies.

The results in Table 4 show that in most of the benchmarks, the gains over different configurations become more prominent as the penalty on various connections is increased. This is because as the communication penalty is more in this model, any saving in communication delay (due to better connectivity) leads to relatively higher improvement. Figure 9 shows that with the *DM*1 delay model, almost every design gives some improvement with the spiral traversal over zig-zag traversal as opposed to model *DM*0 in which some benchmarks do not show any improvement.

6 Conclusion and Future Work

We explored three aspects of network topology in mesh-based coarse-grain reconfigurable architectures: (a) interconnects in the network, (b) topology traversal, and (c) communication delays. Our experimental results show that a topology in which each PE is connected to two other PEs in the same row and column is enough to exploit all the available instruction-level parallelism (ILP) in the set of DSP applications we explored. Also, we

achieve higher performance by employing the spiral network topology traversal strategy since it exploits spatial locality between PEs and first maps PEs that have more interconnections. In future work, we plan to explore speculative code motions and loop transformations to improve the ILP in these designs.

References

[1] R. W. Hartenstein and R. Kress. A datapath synthesis system for the reconfigurable datapath architecture. In *ASP-DAC*, 1995.

[2] C.Ebeling, D. C. Cronquist, P. Franklin, J. Secosky, and S. G. Berg. Mapping applications to the rapid configurable architectures. In *FCCM*, 1997.

[3] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. P. Amarasinghe. Space-time scheduling of instruction-level parallelism on a RAW machine. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[4] S. Cadambi and S. C. Goldstein. Fast and efficient place and route for pipeline reconfigurable architectures. In *ICCD*, 2000.

[5] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho. Morphosys: an integrated reconfigurable system for data parallel and computation-intensive applications. In *IEEE Transactions on Computers*, 2000.

[6] R. Hartenstein. A decade of reconfigurable computing: A visionary retrospective. In *Design, Automation and Test Conference in Europe*, 2001.

[7] T. Miyamori and K. Olukotun. Remarc: Reconfigurable multimedia array coprocessor. In *FPGA*, 1998.

[8] J. Becker, M. Glesner, A. Alsolaim, and J. Starzyk. Architecture and application of a dynamically reconfigurable hardware array for future mobile communication systems. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

[9] P. Schaumont, I. Verbauwhede, M. Sarrafzadeh, and K. Keutzer. A quick safari through the reconfigurable jungle. In *Design Automation Conference*, 2001.

[10] C. A. Mortiz, D. Yeung, and A. Agarwal. Exploring optimal cost-performance designs for raw microprocessors. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1998.

[11] U. Nageldinger. Coarse-grained reconfigurable architectures design space exploration. In *Ph.D. Thesis, University of Kaiserlautern, Germany*, 2001.

[12] L. Bossuet, G. Gogniat, and J. Philippe. Fast design space exploration method for reconfigurable architectures. In *Engineering Of Reconfigurable Systems and algorithms*, 2003.

[13] G. Venkataramani, W. Najjar, F. Kurdahi, N. Bagherzadeh, and W. Bohm. A compiler framework for mapping applications to a coarse-grained reconfigurable computer architecture. In *CASES*, 2001.

[14] B. Mei, S. Vernalde, D. Verkest, H. Man, and R. Lauwerneins. Exploiting loop-level parallelism on coarse-

grained reconfigurable architectures ucing modulo scheduling. In *Design, Automation and Test Conference in Europe*, 2003.

[15] J. Lee, K. Choi, and N. D. Dutt. Compilation approach for coarse-grained reconfigurable architectures. In *Design and Test*, 2003.

[16] P. Quinton and Y. Robert. *Systolic Algorithms and Architectures*. Prentice Hall, 1991.

[17] E. Mirsky and A. DeHon. Matrix: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *FCCM*, 1996.

[18] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.