Analysis of the Performance of Coarse-Grain Reconfigurable Architectures with Different Processing Element Configurations

Nikhil Bansal Sumit Gupta Nikil Dutt Alexandru Nicolau

Center for Embedded Computer Systems
School of Information and Computer Science
University of California at Irvine
{nbansal, sumitg, dutt, nicolau}@cecs.uci.edu

Abstract

Several coarse-grain reconfigurable architectures have been proposed in recent years that consist of a large number of processing elements connected together in a network. These architectures vary widely from each other – particularly, the processing elements (PEs) in these architectures range from simple functional units to entire RISC cores and micro-controllers. In this paper, we study the effect of different PE architectures on the quality of results (performance) of mapping applications to coarse-grain architectures. We present a scheduling and mapping heuristic that takes the interconnections between PEs and the delays on these interconnections into consideration while mapping applications to the PEs of mesh-like coarse-grain architectures. We performed experiments in which we varied the number of functional units in the PEs, the functionality of these units, the number of PEs in the coarse-grain architecture, and the delays on the interconnect that connects the PEs. Our results for a set of designs derived from DSP applications demonstrate that we achieve better performance for these designs by increasing the number of functional units in individual PEs as compared to architectures in which each PE has only one functional unit. Also, our results show that performance improvement for multi-functional unit configurations becomes even more pronounced as delays on the interconnect between PEs increase.

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.

A range of coarse-grain architectures have been proposed that vary in the number of PEs in the fabric, the interconnects between the PEs, the communication delays on these interconnects. The PEs themselves have varied architectures that range from simple functional units to PEs that are complete RISC cores and microcontrollers [10]. In fact the performance of applications mapped to a coarse-grain architecture is best when the granularity of the PEs matches that of the task set in the application.

In this paper, we focus on exploring the effects of varying the functionality and the number of units in PEs, number of PEs in the fabric, and interconnect communication delays between PEs on the performance of applications mapped to the coarse-grain architecture. We use a scheduling and mapping algorithm that does a combination of operation scheduling and operation to PE binding (or mapping) while taking data routing into consideration.

The rest of the paper is organized as follows. Section 2 outlines related work. In Section 3, we describe the different parameters that can be changed in processing element architectures. We then describe the factors that affect the scheduling and mapping of applications to such coarse-grain architectures. We present our list scheduling heuristic in Section 5 and in Section 6, we present our experimental setup and results. Section 7 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. [11] presented a framework that produces an optimal RAW microprocessor structure [3] under cost and area balance constraints for a given application. Nageldinger [12] proposed the *KressArray Xplorer* for

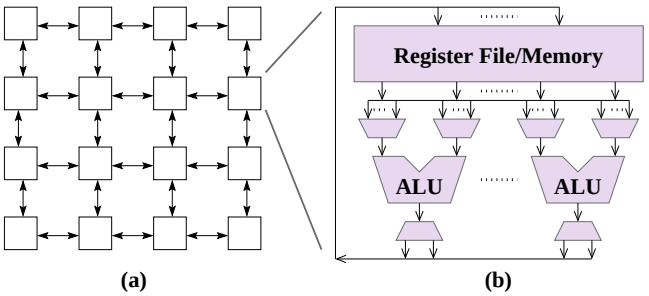


Figure 1. ***(a) A Grid of 4x4 Processing Elements (PEs). (b) Each PE may consist of one or more functional units (ALUs) with a register file or other local memory.***

KressArray architectures [1] to find the best trade-off between the complexity of the hardware and an optimization objective such as performance. Bossuet et al. [13] 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. [14] presented a compiler framework for mapping loops written in SA-C language to the MorphoSys architecture [5]. Mei et al. [15] proposed a modulo loop scheduling approach to map loops on a generic reconfigurable architecture. Lee et al. [16] 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 [17].

Rather than proposing an architecture or mapping to a single architecture, we are more interested in examining the effects of architectures with a different number of processing elements, different delays on the interconnect connecting the PEs, and different number and functionality of units within PEs.

3 Exploration of Processing Element Architectures

A range of architectures have been proposed in recent years for coarse-grain reconfigurable fabrics. These architectures typically consist of a large number of processing elements connected together in mesh-like grids [5, 18, 10], star connections [19], and so on. The processing elements in these architectures also vary widely, ranging from simple functional units (adders, subtracters, multipliers) [5, 20] to multiple functional units [18, 19] to entire RISC cores [10]. We explore the effects of these PE architecture variances on the quality of results (performance) of the applications that are mapped to the coarse-grain architecture.

We consider the target coarse-grain architecture shown in Figure 1. This architecture consists of PEs connected in a mesh-like network, i.e., they are arranged in rows and columns with interconnects among PEs of the same row and column. Each PE consists of one or more functional units with a local memory or register file. We allow the designer to vary different parameters to change the configuration of PE architecture. The parameters we are going to explore in this paper are listed in the following subsections.

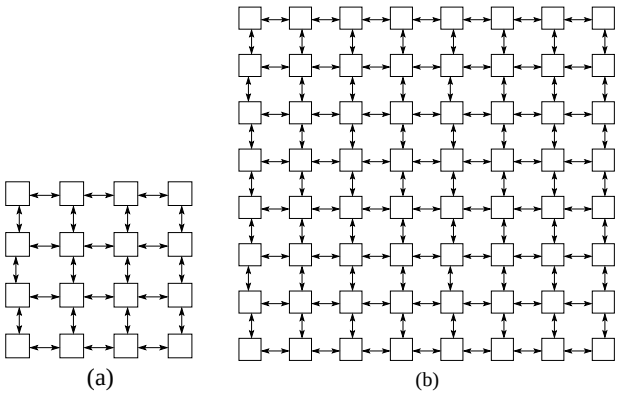


Figure 2. ***(a) 4x4 grid and (b) 8x8 grid with connections only between adjacent PEs. This corresponds to the interconnect topology*** $IT0$ ***described later in the results section (Section 6).***

3.1 Functionality and Number of Units in PEs

The functionality of processing elements in coarse-grain architectures varies greatly. For example, the QuickSilver architecture [19] has four different types of units with different functionality: one each for arithmetic, bit-manipulation, finite state machine, and scalar operations. RAW [3] has a separate floating point unit within each PE (besides the ALU). This motivates us to examine the effects of variation in the functionality and number of units in PEs on application performance.

In our framework, we give the designer the flexibility to specify (a) the number of functional units (FUs) within each PE, (b) operations each FU can perform, and also (c) execution latency of each FU. We assume that the functional units within a PE are fully connected, i.e., there are enough forwarding paths to route data from one unit to another in zero cycles. Thus, increasing the number of FUs in a PE takes advantage of a well-connected cluster of ALUs that can perform sets of dependent computations.

3.2 Number of PEs

The interconnection networks in coarse grain architectures are designed to be scalable and enable adding and removing PEs as and when required (e.g. [1, 18]). This motivates us to explore the effect of changing the number of PEs in the architecture on the performance results. We use a base architecture with PEs are connected in a mesh-like network and then vary the number of PEs in the rows and columns in the network. Figure 2 shows two sample architectures. The architecture of Figure 2(a) has grid of size 4x4 (i.e. with four PEs in each row and column respectively) and the grid on Figure 2(b) has size 8x8.

3.3 Different Communication Delays

Processing elements can communicate with each other either through direct connections or through other PEs. Thus, communication through one other PE is known as 1-hop communication and so on. PEs may also be clustered into tightly connected *grids* that may themselves be

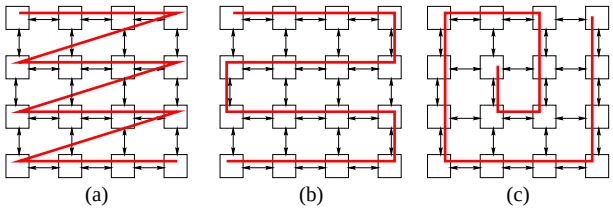


Figure 3. ***Different network traversal strategies: (a) Zig-zag, (b) Reverse-S, (c) Spiral.***

connected to other grids using shared system buses (for example the MorphoSys [5] and MATRIX [23] architectures).

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. In our experiments, we first model existing architectures such as MorphoSys [5] and MATRIX [23] that have zero communication delay between PEs and one cycle for shared bus communication. However, we believe that interconnect speeds will begin to trail computation delays as technology improves. Thus, we also present results for experiments with different communication models in the results section (Section 6).

3.4 Interconnects among PEs

Instead of putting multiple functional units in a PE, we can add more connections between PEs in the coarse-grain fabric. We studied this before in [21], but we present these results here again to compare against the multi-FU case.

Interconnect topology or the interconnection network between PEs restricts the way the data can be routed among the PEs. In fact, most existing and proposed coarse-grain reconfigurable architectures have diverse network topologies. For example, in DReAM [22], each PE is only connected to its immediate neighbors. On the other hand, in MorPhoSys [5], each PE is connected to all other PEs in the same row and same column provided they are in the same grid. The QuickSilver architecture [19] has PEs connected in clusters of star-like formations and these clusters are themselves interconnected in star networks.

In our scheduling algorithm, we enable the designer to vary the number of interconnects between PEs and also support buses between grids or clusters of PEs [5] (however, currently we model only mesh-based networks).

4 Factors Affecting the Scheduling and Mapping Algorithm

Mapping applications to coarse-grain architectures is a combination of scheduling operations in time steps, mapping these operations to PEs that they will execute on, and mapping the data routing to interconnections in the fabric. Hence, we have developed an interconnect scheduling algorithm that takes into account the PE configuration (functionality and number of units), availability of interconnects, and communication delays on the interconnects. We first discuss the factors that have the most effect on the quality of results produced by the scheduler.

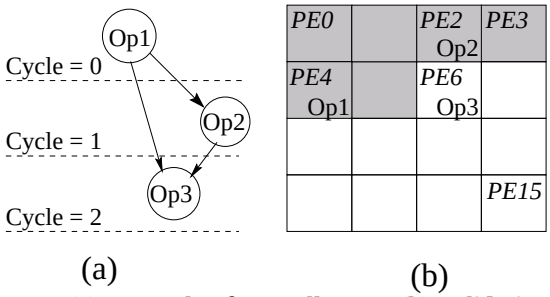


Figure 4. ***(a) Example of a small DFG. (b) Validating the ability to route data.***

4.1 Topology Traversal Strategies

Topology traversal is the order in which PEs are traversed in the network while mapping operations to PEs. We found in earlier work [21] that topology traversal has a profound effect on the quality of mapping results. We proposed the following three network topology traversal strategies:

- *Zig-zag*: As shown in Figure 3(a), each row is traversed from left to right and after completing the mapping of a row, mapping continues from the leftmost PE of the next row.
- *Reverse-S traversal*: As shown in Figure 3(b), this strategy traverses the network in a reverse-S manner. The advantage of this strategy over the zig-zag traversal is that it always traverses spatially adjacent PEs, thus enabling better data routing among 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 showed in [21] that the spiral traversal performs the best since it exploits both spatial locality between PEs and temporal locality for data transfer between operations. Thus, we use the spiral traversal strategy for all the experiments.

4.2 Ability to Route Data

The other important issue is that our scheduling and mapping algorithm has to verify the ability to route the data between PEs. This can be understood by the example shown in Figure 4.

Figure 4(a) shows a sample DFG and we have to map it on the grid shown in Figure 4(b). In this figure, a box represents a PE and each PE is connected to its immediate neighbor only (similar to the grid shown in Figure 2(a)). The shaded box in this figure denotes that there is already an operation mapped on the PE for the current cycle. If we map Op_1 and Op_2 to PE_4 and PE_2 as shown in Figure 4(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 it is possible to schedule operation Op_3 in cycle 2 and map it on PE_6. So, while scheduling an operation on a PE, our scheduling algorithm takes into account the communication delay to get the operations to the

```
/* 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.***

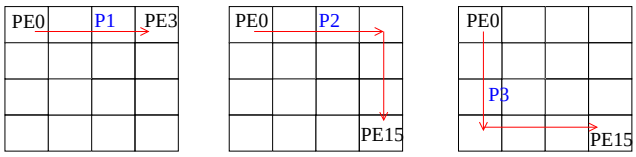
PE being mapped to and cycle in which the operation that produces these operands is scheduled in. This is discussed in detail in the next section.

5 Our Scheduling and Mapping Algorithm

To perform our coarse-grain architecture exploration experiments, we implemented our techniques in a list scheduling algorithm [24]. 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 [16].

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 specified by the user; as discussed earlier, we found the spiral traversal strategy gave the best performance results. A global



(a) (b) (c)

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

clock cycle *currCycle* is maintained during scheduling.

The *ScheduleBB* heuristic starts by collecting a list of available or ready operations, $\mathcal{A}$. *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 $\mathcal{A}_{currPE}$ for *currPE* (lines 3 and 4 in Figure 5(a)). Next, the heuristic chooses the operation (*candOp*) with the highest priority from $\mathcal{A}_{currPE}$. 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.

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 $\mathcal{A}_{CurrPE}$, 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.

While determining the paths from one PE to another, we only consider the most direct (and shortest) paths among PEs. Figure 4(a) 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 (P2 and P3) between PE_0 and PE_{15} as shown in Figure 6 (b) and (c).

Design	Ops.	Design Characteristics
Lowpass	62	A filter which accentuates low frequency in an image
Hydro	120	A 2-D explicit hydrodynamics fragment
Predictor	29	An integrate predictor
ATR	57	Application Target Recognition
FFT	33	A Fast-Fourier Transform implementation
PDE	43	A partial differential equation integration
Laplace	58	It performs edge enhancement of northerly directional edges in an image
GSR	35	Gauss-Seidel Relaxation method
Compress	26	An image compression scheme
LPC	21	A Linear Predictive Coding encoder
SOR	60	A Successive Over-Relaxation algorithm

Table 1. ***Characteristics of designs used in our experiments.***

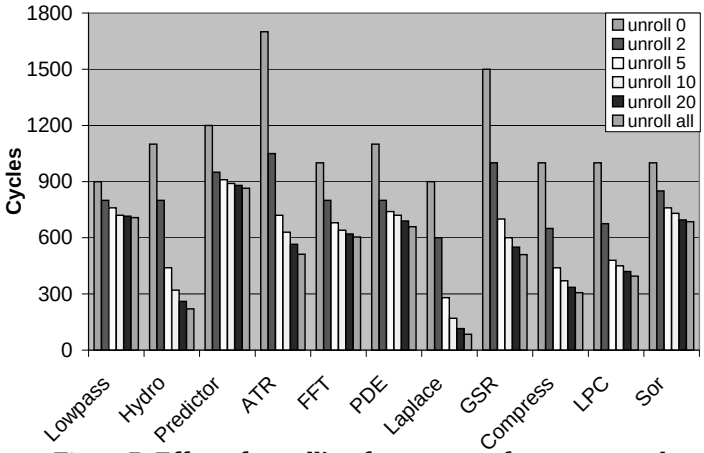


Figure 7. ***Effect of unrolling factor on performance results for grid of size 4x4***

6 Experimental Setup and Results

In order to perform the PE configuration analysis, we implemented our 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 performed by varying different configuration parameters.

We used a set of eleven designs drawn from the DSP domain for our experiments. The characteristics of these designs are shown in Table 1. The first column lists the name of the design, the second column lists the number of operations in the loop of the design before unrolling, and the third column gives a short description of the design.

6.1 Effect of Delay Model

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, 23]).
- *Delay Model DM*1: Direct connection delay is one cycle. 1-hop communication and shared buses take two cycles.

Note that, shared buses between grids can only be used for communicating between one pair of PEs in a cycle [5]. In subsequent sections, we use both of these models to compare the results.

6.2 Effect of Loop Unrolling

In order to expose the instruction level parallelism (ILP) across loop iterations, we unroll the loops in these designs. For designs with nested loops, we unroll the innermost loop. Figure 7 shows the effect of varying the unrolling factor on the performance results, i.e., these represent the cycles that the design takes to execute on the architecture.

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 significant improvement only for the Laplace and Hydro designs. 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 read-after-write data dependencies (Laplace and Hydro), unrolling keeps improving the performance. Since an unrolling factor of 10 improves instruction level parallelism substantially, for the rest of the experiments, we unroll the loops in the designs by 10.

6.3 Effect of varying number of PEs

Next, we varied the number of PEs in the architecture. In Tables 2 and 3, we compare the performance results for architectures with two different grid sizes: 4x4 and 8x8 for delay models *DM*0 and *DM*1 respectively. For both the architectures (shown earlier in Figure 2), each PE is connected only to its immediate neighbor.

The results in this table show that for a few cases there is some improvement in performance as we increase the number of PEs in the architecture, while in most other cases there is no improvement. The improvements are in designs that do not have many or any inter-loop iteration data dependencies. Thus, increasing the number of PEs enables the scheduler to execute several iterations in parallel. In contrast, most of the designs have high inter-iteration dependencies that limit the amount of parallelism among operations. As a result, there are usually not enough operations to fully utilize (or saturate) the reconfigurable fabric.

6.4 Effect of Number of Functional Units in PEs

In this section, we show the effect of changing the configuration of PE. We compare the performance results of two configurations:

- *Configuration Conf*1 *:* A grid of size 8x8 and each PE having 1 ALU.
- *Configuration Conf*2 *:* A grid of size 4x4 and each PE having 4 ALUs.

The number of PEs in both the configurations is 64. Tables 4 and 5 list the execution cycles for these two configurations

Design	Grid Size 4x4 Total PEs = 16	Grid Size 8x8 Total PEs = 64	Total Improvement
Lowpass	94	94	0.00 %
Hydro	88	35	60.23 %
Predictor	90	90	0.00 %
ATR	83	77	7.23 %
FFT	92	92	0.00 %
PDE	31	13	58.06 %
Laplace	42	15	64.29 %
GSR	65	65	0.00 %
Compress	40	40	0.00 %
LPC	49	49	0.00 %
SOR	85	85	0.00 %

Table 2. ***Performance comparison with different number of PEs with delay model*** $DM0$**.**

Design	Grid Size 4x4 Total PEs = 16	Grid Size 8x8 Total PEs = 64	Total Improvement
Lowpass	138	138	0.00 %
Hydro	93	61	34.41 %
Predictor	122	122	0.00 %
ATR	105	100	4.76 %
FFT	137	138	0.00 %
PDE	32	17	46.88 %
Laplace	43	18	58.14 %
GSR	68	68	0.00 %
Compress	42	42	0.00 %
LPC	70	70	0.00 %
SOR	87	87	0.00 %

Table 3. ***Performance comparison with different number of PEs with delay model*** $DM1$**.**

for delay models $DM0$ and $DM1$ respectively. The first column in these tables is the design name, the second and third columns list the number of cycles taken by the two configurations respectively, and the fourth column represents the percentage improvement of $Conf2$ over $Conf1$.

The results in these tables show that $Conf2$ outperforms $Conf1$. The reason for this improvement is that the communication delay among the ALUs within a PE is 0 cycle in $Conf2$. As a result, if an operation Op_i is mapped on a PE PE_j then the dependent operations can also be mapped on the same PE (i.e. PE_j) in the next cycle without incurring any penalty due to interconnect delays (we assume PEs with multiple functional units have a rich set of forwarding paths). However, in $Conf1$ (8x8 with one ALU in each PE), data has to be routed among PEs from producing operations to dependent operations.

We also note that some designs achieve higher improvement on $Conf2$ (as high as 40% in case of Hydro) and other designs achieve much lower improvements (as low as 4% in Compress). We found that the improvements are larger for designs in which there are a larger number of operations with multiple dependent operations. These designs benefit from the well connected functional units in $Conf2$.

Also, the results for different delay models $DM0$ and $DM1$ in Tables 4 and 5 show that performance improves more dramatically with delay model $DM1$ with a higher number of functional units ($Conf2$). These results indicate that as technology improves and the speed of the PEs outpaces the speed of the interconnects, putting more functionality into each PE will lead to better performance for coarse-grain architectures.

Design	1 ALU in a PE Grid Size 8x8	4 ALUs in a PE Grid Size 4x4	Total Improvement
Lowpass	94	72	23.40 %
Hydro	35	34	2.86 %
Predictor	90	79	12.22 %
ATR	77	67	12.99 %
FFT	92	91	1.09 %
PDE	13	12	7.69 %
Laplace	15	15	0.00 %
GSR	65	63	3.08 %
Compress	40	40	0.00 %
LPC	49	49	0.00 %
SOR	85	83	2.35 %

Table 4. ***Performance Comparison with different PE configurations with delay model*** $DM0$**. Total number of PEs in both configurations is 64.**

Design	1 ALU in a PE Grid Size 8x8	4 ALUs in a PE Grid Size 4x4	Total Improvement
Lowpass	138	94	31.88 %
Hydro	61	36	40.98 %
Predictor	122	98	19.67 %
ATR	100	68	32.00 %
FFT	138	90	34.78 %
PDE	17	14	17.65 %
Laplace	18	17	5.56 %
GSR	68	63	7.35 %
Compress	42	40	4.76 %
LPC	70	49	30.00 %
SOR	87	84	3.45 %

Table 5. ***Performance Comparison with different PE configurations with delay model*** $DM1$**. Total number of PEs in both configurations is 64.**

6.5 Effect of Interconnect Topologies

Instead of increasing the number of functional units, we can increase the number of connections among PEs. Thus, we performed experiments with three different interconnect topologies [21]:

- Interconnect Topology $IT0$: PEs are connected to their immediate neighbors in the same row and same column in the grid.
- Interconnect Topology $IT1$: 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.
- Interconnect Topology $IT2$: PEs are connected to 3 other PEs in the same row and column. This corresponds to the MorphoSys architecture [5].

Tables 6 and 7 compare the number of cycles each design takes to execute on the architectures with the three different interconnect topologies for delay models $DM0$ and $DM1$. The tables also list the percentage reduction in cycles with $IT2$ over $IT0$. These results are for one 8x8 grid and four grids of size 4x4 where each row and column in each grid is connected to its adjacent grid by shared buses (as in the MorphoSys architecture [5]).

The results in these tables demonstrate that performance improves significantly as the number of direct connections increase from 1 to 2 (for example $IT0$ to $IT1$ with grid of size 4x4). A higher number of direct connections increases the opportunity to map the dependent operations without incurring any communication penalty.

No. of Cycles for Different Interconnect Topologies								
Design	4 Grids of Size 4x4			Total	1 Grid of Size 8x8			Total
	$IT0$	$IT1$	$IT2$	Reduc.	$IT0$	$IT1$	$IT2$	Reduc.
Lowpass	100	91	83	17.0 %	94	89	80	14.1 %
Hydro	36	34	34	5.6 %	35	34	33	5.7 %
Predictor	91	83	83	8.8 %	90	82	80	11.1 %
ATR	78	72	72	7.7 %	77	71	69	10.4 %
FFT	94	83	83	11.7 %	92	83	83	9.8 %
PDE	14	12	12	14.3 %	13	12	11	15.4 %
Laplace	17	15	15	11.8 %	15	13	13	13.3 %
Compress	40	40	40	0.0 %	40	40	40	0.0 %
GSR	65	64	64	1.5 %	65	64	63	3.3 %
LPC	49	49	49	0.0 %	49	49	49	0.0 %
SOR	84	75	75	10.7 %	85	77	76	10.6 %

Table 6. ***Delay model*** $DM0$***: Performance results comparison for different interconnect topologies.***

No. of Cycles for Different Interconnect Topologies								
Design	4 Grids of Size 4x4			Total	1 Grid of Size 8x8			Total
	$IT0$	$IT1$	$IT2$	Reduc.	$IT0$	$IT1$	$IT2$	Reduc.
Lowpass	135	124	111	17.7 %	138	126	118	14.5 %
Hydro	61	57	57	6.6 %	61	57	56	8.2 %
Predictor	122	110	110	9.8 %	122	113	108	11.5 %
ATR	102	98	97	4.9 %	100	100	97	3.0 %
FFT	137	127	120	12.4 %	138	132	124	10.1 %
PDE	16	15	14	12.5 %	17	16	15	11.8 %
Laplace	19	18	16	15.8 %	18	18	16	11.1 %
GSR	67	65	65	3.0 %	68	67	65	4.4 %
Compress	41	41	41	0.0 %	42	42	42	0.0 %
LPC	70	70	70	0.0 %	70	70	70	0.0 %
SOR	86	82	76	11.6 %	87	84	77	11.5 %

Table 7. ***Delay model*** $DM1$***: Performance comparison for different interconnect topologies.***

In contrast, there is almost no performance improvement when number of direct connections increase from 2 to 3 ($IT1$ to $IT2$ in grid of size 4x4). This is because, when we change the configuration from $IT1$ to $IT2$ with 4x4 grid, 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 for some designs from configuration $IT1$ to $IT2$ with 8x8 grid. These experiments show that the interconnect topology $IT1$ is sufficient to exploit most of the ILP for the set of designs we considered in our experiments.

The results in Tables 6 and 7 also show that in most of the designs, the gains over different configurations become more prominent as the penalty on various connections is increased, i.e., when we move from delay model $DM0$ to $DM1$. This is similar to the results in Section 6.4 that show that as interconnect delays increase, better connected fabrics will outperform poorly connected networks.

6.6 Effect of changing the functionality of FUs within a PE

Next, we vary the the functionality of the functional units (FUs) within the PEs. In all our experiments so far, the base PE architecture consists of functional units (or ALUs) that are capable of performing every arithmetic and bit-manipulation operation (additions, multiplications, comparisons, logical ANDs/ORs et cetera). For simplicity, we take the execution time of the ALUs as one cycle for all operations(in reality, multiplies should take longer to execute).

Design	1 ALU with all operations	1 Arithmetic + 1 Multiplier	Improvement
Lowpass	135	117	13.33 %
Hydro	61	58	4.92 %
Predictor	122	121	0.82 %
ATR	102	92	9.80 %
FFT	137	129	5.84 %
PDE	16	16	0.00 %
Laplace	19	18	5.26 %
GSR	67	67	0.00 %
Compress	41	41	0.00 %
LPC	70	62	11.11 %
SOR	86	86	0.00 %

Table 8. ***Performance Comparison with one ALU that performs all the operations versus a separate multiplier and a ALU that performs all other operations. Results are for a 4 grids of size 4x4 with delay model*** $DM1$***.***

We change the PE functionality by splitting the ALU in two functional units. One of these functional units can perform multiplications (and divisions if required) while other one can perform the remaining operations. As before, we assume that there are forwarding paths within PEs to route data between the functional units in zero cycles. The overall functionality of the PEs is same in both the cases, i.e., each PE has only one functional unit for every kind of operation. We call the configuration in which each PE has one FU that performs all operations as configuration $FU1$ and the other configuration with two separate FUs as $FU2$.

Table 8 compares the performance results for the two configurations with delay model $DM1$ (we got similar results for $DM0$). The first column lists the design, the second and third columns list the number of cycles taken by the two configurations respectively and the fourth column lists the percentage improvement of the configuration with two FUs over the one with one FU.

The results in Table 8 show that we observe improvements in only half the designs with two split ALUs. These improvements come from the fact that these designs have multiplies and additions (or other operations) that can be executed in parallel due to the split ALUs. We plan to study this more carefully in future work to determine what kind of split in ALU functionality gives the best performance improvements. For example, QuickSilver does coarse level functionality splitting into different PEs for arithmetic, bit-manipulation, finite state machine, and scalar operations. Splitting the functionality of FU gives the scheduler some more flexibility in scheduling non-data dependent operations concurrently.

7 Conclusions and Future Work

We explored several aspects of PE configuration in mesh-based coarse-grain architectures: (a) number of PEs in the architecture, (b) number of functional units in each PE, (c) different number of interconnects between PEs, and (d) different functionality of functional units in the PEs. Moreover, we presented results for these aspects for two interconnect communication delay models. We first found that larger grids do not necessarily translate into better per-

formance. Often, there is not enough instruction-level parallelism (ILP) in the designs to fully utilize the PEs in the fabric. We also found that increasing either the number of functional units in each PE or the interconnections among PEs leads to much better quality of results. In both cases, there are more interconnections and therefore, fewer delays in routing the data between dependent operations. Finally, we found that splitting the functionality of PEs can lead to better results for some designs. This depends on whether the functionalities of the split ALUs in the PEs are aligned to the availability of concurrent operations in the design that can be mapped to these ALUs. Overall, we observed as delays between interconnects increase, the improvements increase for architectures with better connected networks and with PEs that have a larger number of functional units. In future work, we want to verify our results for a larger set of designs and explore loop transformations to increase the instruction level parallelism in the 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 *International Symposium on FPGAs*, 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] picoChip Designs Limited. `http://www.picochip.com/`.

[11] 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.

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

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

[14] 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.

[15] 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.

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

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

[18] Pact XPP Technologies. `http://www.pactcorp.com/`.

[19] QuickSilver Technology. `http://www.qstech.com/`.

[20] IPFlex Incorporated. `http://www.ipflex.com/`.

[21] N. Bansal, S. Gupta, N.D. Dutt, A. Nicolau, and R.K. Gupta. Network topology exploration of mesh-based coarse-grain reconfigurable architectures. Technical Report CECS-TR-03-27, Center for Embedded Computer Systems, Univ. of California, Irvine, 2003.

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

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

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