Interconnect-Aware Mapping of Applications to Coarse-Grain Reconfigurable Architectures[⋆]

Nikhil Bansal[1], Sumit Gupta[2], Nikil Dutt[1], Alex Nicolau[1], and Rajesh Gupta[3]

[1] University of California, Irvine,
[2] Tallwood Venture Capital, Palo Alto,
[3] University of California, San Diego

Abstract. Coarse-grain reconfigurable architectures consist of a large number of processing elements (PEs) connected together in a network. For mapping applications to such coarse-grain architectures, we present an algorithm that takes into account the number and delay of interconnects. This algorithm maps operations to PEs and data transfers to interconnects in the fabric. We explore three different cost functions that largely affect the performance of the scheduler: (a) priority of the operations, (b) affinity of operations to PEs based on past mapping decisions, and (c) connectivity between the PEs. Our results show that a priority-based operation cost function coupled with a connectivity-based PE cost function gives results that are close to the lower bounds for a range of designs.

1 Introduction

Coarse-grain reconfigurable architectures have been proposed as co-processors for accelerating compute intensive portions (generally loops) of applications in embedded system platforms. These architectures are attractive to system designers because they provide the high performance of ASICs with the ease of reconfiguration of fine-grain FPGAs. As a result, we have seen the emergence of a wide range of coarse-grain reconfigurable architectures over the last few years [1,2,4,5,6]. (to name a few). Mapping applications to coarse-grain architectures is a combination of assigning time cycles for operations to execute in (scheduling), mapping these operation executions to specific PEs (mapping), and routing the operands or input data by mapping and scheduling data communications to specific interconnects in the fabric (routing).

In this paper, we present an algorithm that performs these tasks by taking into account the spatial locality or connections between the PEs and the temporal locality between the data used by the operations. We examine different metrics that affect this algorithm – specifically (a) the *priority* of operations based on the length of the dependency chain or critical path through the code, (b) the *affinity* of operations to PEs, i.e., operations are more likely to be mapped to a PE if their predecessor operations are also mapped to that PE or one of its connected neighbors, and (c) the *connectivity* of the PEs, i.e., we first map operations to PEs that are connected to the maximum number of other PEs, thereby, exploiting their spatial locality. The main contribution of this paper is in examining the

[⋆] This work was partially supported by NSF grants CCR-0203813, ACI-0204028, and Hitachi Corporation.

```
/* Schedules & maps operations in basic block currBB */
ScheduleMapBB(currBB, currCycle, PEList)

$\mathcal{A}_{avail} \leftarrow$ List of all available operations of currBB
        in currCycle
while ($\mathcal{A}_{avail} \neq \phi$ )
    Calculate $C_{op}$ $\forall$ ops in $\mathcal{A}_{avail}$ and $C_{PE}$ $\forall$ PEs in PEList
    $\mathcal{A}_{ordered} \leftarrow$ Order $\mathcal{A}_{avail}$ by cost function $C_{op}$
    PEList $\leftarrow$ PEList ordered by cost function $C_{PE}$
    foreach (candPE $\in$ PEList with lowest cost $C_{PE}$)
        $\mathcal{A}_{candPE} \leftarrow \mathcal{A}_{ordered}$
        while ($\mathcal{A}_{candPE} \neq \phi$ AND candPE not scheduled)
            Pick candOp $\in \mathcal{A}_{candPE}$ with lowest cost $C_{op}$
            $\mathcal{A}_{candPE} \leftarrow \mathcal{A}_{candPE}$ - candOp
            if ( IsRoutable(candOp, candPE, currCycle) )
                $\mathcal{A}_{ordered} \leftarrow \mathcal{A}_{ordered}$ - candOp
                $\mathcal{A}_{avail} \leftarrow \mathcal{A}_{avail}$ - candOp
                Schedule candOp on candPE in currCycle
                Update route usage information in currCycle
    currCycle $\leftarrow$ currCycle + 1
    $\mathcal{A}_{avail}$ += List of all available operations of currBB
            in currCycle
```

Fig. 1. Algorithm to schedule a basic block

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

foreach (predOp $\in$ PREDs(candOp))
    predPE $\leftarrow$ PE on which predOp is mapped
    foreach (route $\in$ GetRoutes(predPE, candPE))
        if (currCycle < EndTime(predOp) + Delay(route))
            or (RouteNotAvailable(route, currCycle))
            return false
return true
```

Fig. 2. Algorithm for verifying routability of data

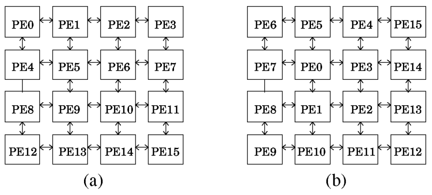


Fig. 3. **(a)** PE indexing in base algorithm and **(b)** PE indexing in connectivity based algorithm

different metrics that influence the mapping results and demonstrating which ones do better than others.

The rest of the paper is organized as follows. Section 2 outlines the related work. We present the base mapping algorithm in Section 3 and then explore three different cost functions which affect the performance of the algorithm. In Section 4 we present our experimental setup and results. Section 5 concludes the paper.

2 Related Work

Several efforts have focused on algorithms for mapping applications to coarse-grain architectures. Huang et al. [12] proposed a methodology to map loops on the architecture and then merge all the data paths corresponding to different loops. Venkataramani et al. [15] presented an algorithm for mapping loops written in SA-C language to the MorphoSys architecture [5] which uses a similar notion of *affinity*, but no detailed results are available.

RaPid [2] uses a C-like language to program loops which requires a considerable knowledge about the underlying architecture. Mei et al. [17] proposed a modulo loop scheduling approach to map loops on a generic reconfigurable architecture. A list scheduling based approach enriched with a priority based heuristic was used for PipeRench architecture [4] in which priority was defined on the basis of distance from the nearest *non-routing node*. Our algorithm bears some resemblance with this work, however, we differ in terms of the heuristics used and the target architecture which consists of a mesh based array of PEs in our case.

In this paper, we examine different cost functions that have an impact on the mapping of applications to a generic mesh-based coarse-grain architecture using a list scheduling based mapping approach. Based on our experimental results, we provide new insights into the metrics that affect this mapping.

3 Base Mapping Algorithm

Our base algorithm traverses the control-data flow graph of the application and schedules and maps one basic block at a time. $ScheduleMapBB$, the heuristic for scheduling the operations in a basic block and mapping them to PEs, is listed in Figure 1. The heuristic takes as input basic block to be considered ($currBB$), a global clock cycle ($currCycle$), and the list of all the PEs ($PEList$) in the architecture.

The $ScheduleMapBB$ heuristic starts by collecting a list of available or ready operations, $\mathcal{A}_{avail}$, in the current cycle. *Available operations* are operations whose data dependencies are satisfied and can be scheduled in the current cycle. $\mathcal{A}_{avail}$ and $PEList$ are then ordered by the cost functions C_{op} and C_{PE} to store back in $\mathcal{A}_{ordered}$ and $PEList$ respectively. We examine the effect of varying the C_{op} and C_{PE} cost functions in the following sections. The heuristic then maps operations to PEs starting with the PE $candPE$ having the minimum cost C_{PE}. We first make a copy of the available operation list as $\mathcal{A}_{candPE}$ for $candPE$ (lines 6 and 7 in Figure 1) since operations that cannot be mapped to $candPE$ will be removed from the $candPE$'s available list. Next, the heuristic chooses the operation ($candOp$) with the lowest cost, C_{op}, from $\mathcal{A}_{candPE}$.

Once an operation and a PE is selected, $ScheduleMapBB$ calls the $IsRoutable$ function to verify if there is a route available for the data required by $candOp$ to reach $candPE$ in $currCycle$. This function is presented in detail in the next section. If $IsRoutable$ does not find any route, then $ScheduleMapBB$ considers the next operation in $\mathcal{A}_{candPE}$ till it maps an operation to $candPE$ or no more operations are left. If $IsRoutable$ returns a true result, $candOp$ is mapped on $candPE$ and scheduled to execute in $currCycle$. We also store the usage information of different connections for each cycle. This information is used by $IsRoutable$ function to check the availability of different routes. Once we map an operation on a PE, usage information of all the connections used for this operation is updated. (lines 11 to 16 in Figure 1).

In this way, the $ScheduleMapBB$ heuristic schedules and maps operations on each PE in $PEList$ and then increments $currCycle$ when $PEList$ is exhausted. Note that, in each cycle we restart the mapping of PEs in the same fashion. This process is continued until all the available operations in the current basic block have been scheduled.

3.1 Routing Algorithm

The $IsRoutable$ function, outlined in Figure 2, verifies the ability to route data from the predecessors of $candOp$, to $candPE$ in $currCycle$. Thus, the $IsRoutable$ function checks all the routes from each $predPE$ (on which a predecessor operation is mapped) to $candPE$ by calling the function $GetRoutes$ (lines 2 and 3 in Figure 2). These routes and delays on them are determined statically before scheduling so that there is no additional run-time overhead of finding routes in terms of complexity of the algorithm. A route from $predPE$ to $candPE$ cannot be used if: either the cycle in which the predecessor operation finishes execution ($EndTime(predOp)$) summed with the delay of the route ($Delay(route)$) is larger than the current cycle ($currCycle$), or if the route is not available, i.e., some connection on the route is used by another data communication in the same cycle (lines 4 to 6 in Figure 2).

The routability verification algorithm is a constant time algorithm as routes are determined statically at the start of the scheduling. The worst case complexity of the scheduling and mapping algorithm is $O(m * n^2)$ where m is the number of PEs in the architecture and n is the number of operations in the basic block. The actual run-times of our algorithm for an architecture having 16 PEs is in the range of 10 user seconds for the designs considered (see Section 4).

Apart from verifying the routability of the operands, two other aspects that affect the performance of the scheduler are the cost functions C_{op} and C_{PE}. Over the next few sections, we present analysis of different cost functions.

3.2 Base PE and Operation Cost Functions

PEs are selected from the $PEList$ based on the cost function C_{PE}. C_{PE} for the base algorithm is defined as equal to the index of that PE; we assign indices to the PEs from PE_0 to PE_{15} (as shown in Figure 3(a)). Operations are selected from the list of available operations based on the cost function C_{op}. For the base algorithm, cost is randomly assigned to all the operations, i.e., operations are randomly selected. Next, we analyze other C_{PE} and C_{op} functions.

3.3 Priority-Based C_{op}

Since we are trying to optimize performance of the applications mapped to the coarse-grain architecture, it is intuitive to give preference to the operations that lie on the critical path through the code. Hence, we assign a priority to each operation in the input description based on the length of the chain of operations that depend on it. 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 (primary outputs) have a priority of one. The operation cost function (C_{op}) is taken as the negative of its priority. In other words, higher the priority, lower the cost.

3.4 Affinity Based C_{op} and C_{PE}

If operation Op_i is mapped on PE PE_m, then communication delay can be minimized by mapping operation Op_j that reads the result of Op_i on a PE PE_n that is either directly connected or connected through the fewest intermediate links to PE_m. This leads us to the notion of *affinity* between operations and PEs. We define affinity, $Aff(Op_i, PE_m)$, of an operation Op_i to a processing element PE_m as the sum of the number of *parents* of Op_i that were mapped on any PE adjacent to PE_m. Processing element PE_m is considered *adjacent* to PE PE_n if they have point-to-point connection between them. Note that, a PE is adjacent to itself. Note also that affinity of operations have to be calculated at the beginning of each cycle during the scheduling and mapping process.

Thus, affinity captures the data dependency information along with past operation to PE mapping decisions. We can, thus, use affinity to map operations to PEs with which they have the highest affinity and in the process minimize communication delays. That is, we calculate the operation cost function C_{op} as the negative of its

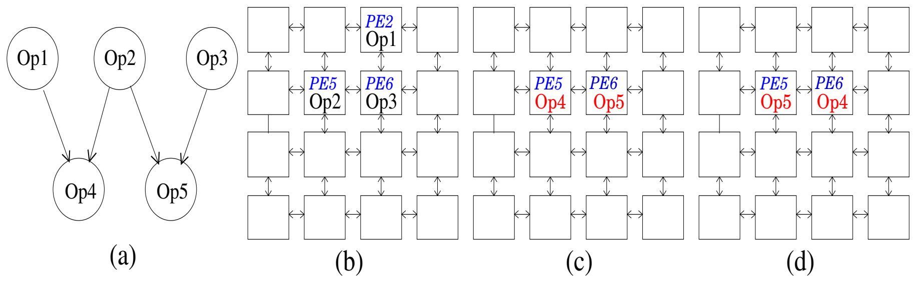


Fig. 4. (a) Example DFG, **(b)** Mapping of parent ops., **(c)** Mapping with a delay, **(d)** Mapping without delay

affinity. If two operations have same cost then we select the operations on the basis of their priorities. Note that, a similar notion of affinity has been discussed earlier by Venkataramani et al. [15]. We also associate an affinity with each PE. Affinity of a processing element PE_m is the sum of affinities of all the operations to PE_m, that is:

$$Aff_{PE}(PE_m) = \sum Aff(Op_i, PE_m) \; \forall \; Op_i \in \mathcal{A}_{avail}$$

We then take the cost of a PE, C_{PE}, as equal to its affinity. We schedule operations on PEs starting with the PE having *lowest affinity*. The reason for this can be understood by the example DFG shown in Figure 4(a). The target architecture we consider is shown in Figure 4(b). If we map Op_1 on PE_2, Op_2 on PE_5, and Op_3 on PE_6 as shown in Figure 4(b) then both Op_4 and Op_5 have affinity of two for PE_6. However, only Op_4 has affinity of one for PE_2 and only Op_5 has affinity of two for PE_5. By definition, the affinities of different PEs are: PE_2 has 2, PE_5 has 3, and PE_6 has 4. Now if we choose to schedule PE_6 first (i.e. PE with the highest affinity), then we *may* choose to map operation Op_5 on it, instead of Op_4 since they both have an affinity of 2 to PE_6. The resultant mapping is shown in Figure 4(c). However, this means that Op_4 will have to be mapped to PE_5 (or any other PE). This in turn means that the result of Op_1 will suffer a communication delay of one cycle to reach Op_4. However, according to the proposed algorithm, we first choose PE_2 only to find that no operation is routable on this PE. Then we choose PE_5 and find that Op_5 is routable on it, so we map it on PE_5. Now there is only one choice – that of mapping Op_4 on PE_6. The resultant mapping is shown in Figure 4(d) which has no communication delay.

Thus, we first map PEs (having non-zero affinity) in increasing order of affinity and then the rest of the PEs based on the PE indices, (starting from top left corner to bottom right corner).

3.5 Connectivity Based C_{PE}

We noticed that in mesh architectures like the one shown in Figure 3(a), the PEs at the corner of the grid (PE_0, PE_3, PE_{12}, PE_{15}) have only 3 directly connected neighbors. In contrast, PEs at the center of the grid have 5 neighbors. Mapping operations to PEs in increasing order of their indices means that the operations with highest priority and/or affinity are first mapped on the sparsely connected PEs on the edge (first row) of the grid. Thus, the data produced by these operations can be routed to a smaller number of PEs than if the operations were mapped to the PEs at the center of the grid. This observation led us to develop a PE ordering in which operations are first mapped to PEs that are *better connected*. Thus, in our *connectivity-based* cost function, we give higher preference to the PEs with more number of connections, i.e., the PEs at the center of the grid. We assign the indices to the PEs starting from the PE at the center, as shown in Figure 3(b). The PE cost function C_{PE} is equal to the index of the PE, but the indexing of PEs is changed which, in turn, changes the cost function.

4 Experimental Setup and Results

In order to evaluate the applicability of the algorithms proposed in this paper, we implemented them 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. We used a set of seven designs drawn from the DSP domain for our experiments. All these designs consist of straight-line code with a loop (or nested loops).

For all the experiments in this paper, we consider an architecture with 16 PEs connected in a 4x4 array. We found little change in the relative numbers with larger arrays [11]. Each PE has one functional unit, which is capable of executing any operation in one cycle. The interconnect delay on direct connections is taken as 0 cycle, unless otherwise specified. The typical run time of our algorithm is 10 user seconds on a 400 MHz UltraSparc-II machine.

In all the experiments presented in this paper, we make some assumptions: (a) there is enough memory bandwidth to fetch data without any delay, (b) there are enough registers to store all the intermediate and final results, and (c) the architecture supports cycle-by-cycle reconfiguration. We plan to address these assumptions in future work.

4.1 Comparison Algorithm

In an attempt to demonstrate the efficacy of the proposed algorithm, we created an *Integer Linear Programming* (ILP) formulation of the mapping problem. To solve these ILP formulations, we used publicly available solvers; LP_SOLVE and CAP (Contig Assembly Program). Despite their reported efficiency, neither of these solvers were able to solve ILP formulations (within a few days), corresponding to a realistic application. Still, to provide some comparison baseline, we devised a heuristic that uses a zero-delay routing model. Specifically, all the PEs are assumed to be connected to each other with a full crossbar interconnect. We use a priority based list scheduling heuristic to map operations on this architectural model. This heuristic gives a lower bound on the mapping results as there is no delay induced by routing. We manually looked at the results of this heuristic and found little or no opportunity to improve them. We compare all our results with the lower bounds generated by this heuristic.

Design	Operations	No. of Cycles	IPC
Lowpass	652	106	6.15
SOR	630	115	5.48
Hydro	1290	107	12.06
ATR	508	100	5.08
FFT	286	103	2.78
Predictor	618	104	5.94
PDE	463	99	4.68

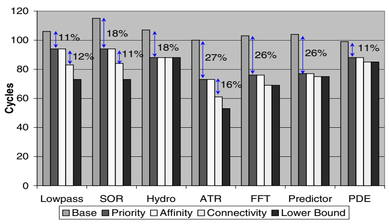


Fig. 5. (a) Mapping results for 7 DSP designs using base algorithm, **(b)** Comparison of mapping results using different cost functions for 0-delay interconnect model

4.2 Results for Base Algorithm

In order to expose the parallelism of the application, we unroll the loops that increases the number of operations to map. In case of nested loops, we unroll the innermost loop. For all the experiments in this paper we present the results with an unrolling factor of 10 since we have shown earlier that unrolling factor of 10 is sufficient to explore the inherent parallelism of the designs [10]. Table 5(a) shows the mapping results for the designs using base algorithm. Third column in this table represents the number of cycles needed to execute the design using base algorithm.

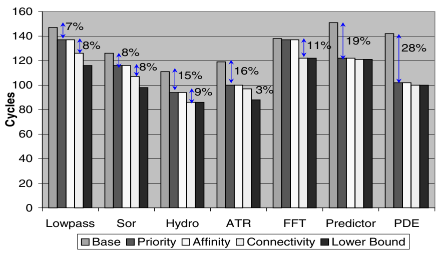
4.3 Comparison of Priority, Affinity, and Connectivity-Based Cost Functions

The results shown in Figure 5(b) demonstrate that a simple cost function based on the priority of the operations gives significant improvement (up to 27%) over base algorithm. But surprisingly a more sophisticated cost function based on affinity does not give any improvement over priority based algorithm. The reason is that the routing function $IsRoutable$ (explained in Section 3.1) implicitly considers the data dependency information when it finds the shortest routes in terms of communication time; this obviates the need for complex affinity-based cost functions.

In contrast, connectivity based algorithm gives a further improvement of up to 16% (in case of ATR) over the priority based algorithm. This is because of our earlier claim of exploiting the better connectivity of PEs at the center of the grid. In fact, in most cases, the connectivity based algorithm gives results that are close to the lower bounds.

4.4 Results for Varying Interconnect Delays

In order to support our claim about the applicability of the algorithm for different interconnect delays, we performed experiments with a delay of 1 cycle (instead of 0) on point-to-point connections. Figure 6(a) shows the performance results corresponding to different cost functions with this interconnect model. The results in this figure are similar to the results corresponding to the zero delay interconnect model. This demonstrates the effectiveness of our mapping algorithm and usefulness of the priority and connectivity based mapping strategies.



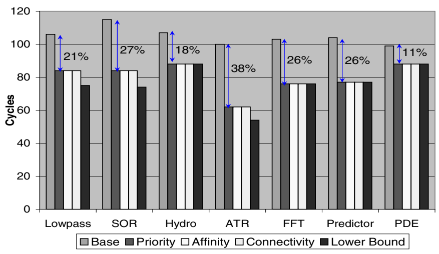


Fig. 6. **(a)** Comparison of mapping results using different cost functions for 1-cycle delay interconnect model, **(b)** Comparison of performance for different cost functions for the torus architecture.

4.5 Results for Torus Architectures

We introduced the notion of the connectivity-based PE cost function to use the information about the differing number of connections between PEs during mapping. However, there are some aggressive architectures in which all the PEs have same number of connections. For example, in the architectures having *torus* shaped interconnects [1], PEs in the first row (column) are also connected to the PEs in the last row (column) using wrap-around connections.

Figure 6(b) shows the performance results corresponding to this architecture model (delay on direct connection is zero cycle). These results show that with the torus architecture, as expected, the connectivity-based cost function does not give any improvement over the priority-based function. In fact the priority-based algorithm now gives results that are close to the lower bounds.

5 Conclusion

We explored three different cost functions which affect the performance of mapping applications on to coarse-grain reconfigurable architectures: (a) a priority-based function in which operations on the longest dependency chain are given preference, (b) an affinity-based function in which an operation gets preference for a PE, if any of its predecessors was mapped to that PE or its adjacent PEs, and (c) a connectivity-based function in which preference is given to PEs that have more connections to other PEs. Although the affinity-based strategy seems intuitive and useful, our experimental results show that the priority-based operation cost function coupled with connectivity-based PE cost function is sufficient enough to give results that are close to the lower bounds for most of the designs considered.

References

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

[2] C. Ebeling et al. Mapping applications to the rapid configurable architectures. In *FCCM*, 1997.

[3] W. Lee et al. Space-time scheduling of instruction-level parallelism on a RAW machine. In *ASPLOS*, 1998.

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

[5] H. Singh et al. 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 *DATE*, 2001.

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

[8] J. Becker, M. Glesner, A. Alsolaim, J. Starzyk. Architecture and application of a dynamically reconfigurable hardware array for future mobile communication systems. *FCCM*, 2000.

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

[10] Omitted for blind review.

[11] Omitted for blind review.

[12] Z. Huang, S. Malik. Exploiting operational level parallelism through dynamically reconfigurable datapath. *DAC*, 2002.

[13] T.J. Callaham and J. Wawrzynek. Adapting software pipelining for reconfigurable computing. In *CASES*, 2000.

[14] K. Bondalapati and V. K. Prasanna. Loop pipelining and optimization for run-time reconfiguration. In *RAW*, 2000.

[15] G. Venkataramani et al. A compiler framework for mapping applications to a coarse-grained reconfigurable computer architecture. In *CASES*, 2001.

[16] J. Lee, K. Choi, N. Dutt. Compilation approach for coarse-grained reconfigurable architectures. *IEEE D&T*, 2003.

[17] B. Mei et al. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures ucing modulo scheduling. *DATE*, 2003.

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

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

[20] R. A. Bittner, P. M. Athanas, M. D. Musgrove. Colt: An experiment in wormhole run-time reconfiguration. *SPIE*, 1996.