11

Greedy Algorithms for Mapping onto a Coarse-grained Reconfigurable Fabric[1]

Colin J. Ihrig, Mustafa Baz, Justin Stander, Raymond R. Hoare, Bryan A. Norman, Oleg Prokopyev, Brady Hunsaker and Alex K. Jones
University of Pittsburgh, United States

1. Introduction

This book chapter describes several greedy heuristics for mapping large data-flow graphs (DFGs) onto a stripe-based coarse-grained reconfigurable fabric. These DFGs represent the behavior of an application kernel in a high-level synthesis flow to convert computer software into custom computer hardware. The first heuristic is a limited lookahead greedy approach that provides excellent run times and a reasonable quality of result. The second heuristic expands on the first heuristic by introducing a random element into the flow, generating multiple solution instances and selecting the best of the set. Finally, the third heuristic formulates the mapping problem of a limited set of rows using a mixed-integer linear program (MILP) and creates a sliding heuristic to map the entire application. In this chapter we will discuss these heuristics, their run times, and solution quality tradeoffs.

The greedy mapping heuristic follows a top-down approach to provide a feasible mapping for any given application kernel. Starting with the top row, it completely places each individual row using a limited look-ahead of two rows. After each row is mapped, the mapper will not modify the mapping of any portion of that row. This mapping approach is deterministic as it uses a priority scheme to determine which elements to place first based on factors such as the number of nodes to which it connects and second based on the desirability of a particular location in the row. While the limited information available to the mapper does not often allow it to produce optimal or minimum-size mappings, its runtime is typically a few seconds or less. We use a fabric interconnect model (FIM) file in the mapping flow to define a set of restrictions on what interconnect lines are available, the capabilities of particular functional units (e.g. dedicated vertical routes versus computational capabilities) in the system, etc.

The greedy heuristic is deterministic in the priority system which it uses to place nodes. The second mapping heuristic we explore is based on this greedy algorithm and introduces randomness into the heuristic to make decisions along the priority list. In the first implementation the node selection order is selected randomly. In the second version, weights are assigned to nodes based on the deterministic placement order. Since the heuristic runs so quickly, we can run the heuristic 10's or possibly 100's of times and select the best result. This method is also parameterizable with the FIM.

[1] This work partially supported by The Technology Collaborative.

Source: Advances in Greedy Algorithms, Book edited by: Witold Bednorz,
ISBN 978-953-7619-27-5, pp. 586, November 2008, I-Tech, Vienna, Austria

Finally, we present a sliding window algorithm where groups of rows are placed using an MILP. This heuristic starts with an arbitrary placement where operations are placed in the earliest row possible and the operations are left justified. Starting from the top, a window of rows is selected and the IP algorithm adjusts column locations where the optimization criteria is to only use allowed routes specified by the architecture. If the program cannot find a feasible mapping, it tries to push violated edges (i.e. edges that do not conform to what is allowed in the architecture) down in the window so that subsequent windows may be able to find a solution. If no feasible solution can be found in the current window, then a row of pass-gates is added to increase the flexibility, and the MILP is run again. However, introducing a row of pass-gates delays the critical path and is undesirable from a power and performance perspective. This technique is also parameterizable within the FIM.
In this chapter, these three heuristics will be explained in detail and numerous performance evaluations (including feasibility) will be conducted for different architectural configurations. Section 2 provides a background on the reconfigurable fabric concept and the process of mapping as well as related work. Section 3 introduces the Fabric Interconnect Model, an XML representation of the fabric. In Section 4 the greedy heuristic is described in detail. In particular, the algorithms for row and column placement are discussed. Section 5 extends the greedy heuristic by introducing an element of randomness into the algorithm. Several methods of randomizing the greedy heuristic are explored, including completely random decisions and weighted decisions. In Section 6 the sliding partial MILP heuristic is introduced. In addition, several techniques for improving the execution time of the MILP are discussed. These techniques are based on decomposing the problem into smaller, simpler linear programs. Finally, Section 7 compares the different mapping techniques and provides some conclusions.

2. Background and literature review

A general trend seen during application profiling is that 90% of application execution time in software is spent in approximately 10% of the code. The idea of our reconfigurable device is to accelerate high incidence code segments (e.g. loops) that require large portions of the application runtime, called kernels, while assigning the control-intensive portion of the code to a core processor.

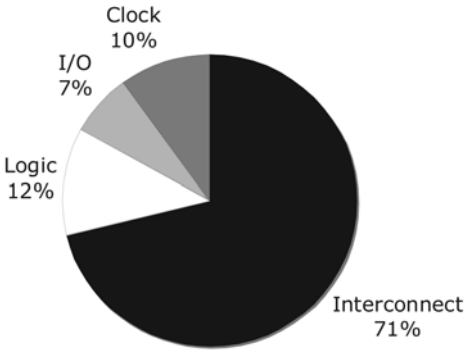


Fig. 1. Power consumption features of a Xilinx Virtex-2 3000 FPGA (Sheng et al., 2002).

A tremendous amount of effort has been devoted to the area of hardware acceleration of these kernels using Field Programmable Gate Arrays (FPGAs). This is a particularly popular method of accelerating computationally intensive Digital Signal Processing (DSP) applications. Unfortunately, while FPGAs provide a flexible reconfigurable target, they have poor power characteristics when compared to custom chips called Application Specific Integrated Circuits (ASICs). At the other end of the spectrum, ASICs are superior in terms of performance and power consumption, but are not flexible and are expensive to design.

The dynamic power consumption in FPGAs has been shown to be dominated by interconnect power (Sheng et al., 2002). For example, as shown in Figure 1, the reconfigurable interconnect in the Xilinx Virtex-2 FPGA consumes more than 70% of the total power dissipated in the device. Power consumption is exacerbated by the necessity of bit-level control for the computational and switch blocks.

Thus, a reconfigurable device that exhibits ASIC-like power characteristics and FPGA-like flexibility is desirable. Recently, the development and use of coarse-grained fabrics for computationally complex tasks has received a lot of attention as a middle ground between FPGAs and ASICs because they typically have simpler interconnects. Many architectures have been proposed and developed, including MATRIX (Mirsky & Dehon, 1996), Garp (Hauser & Wawrzynek, 1997), PipeRench (Levine & Schmit, 2002), and the Field Programmable Object Array (FPOA) (MathStar, MathStar).

Our group has developed the SuperCISC reconfigurable hardware fabric to have low-energy consumption properties compared to existing reconfigurable devices such as FPGAs (Mehta et al., 2006; Jones et al., 2008; Mehta et al., 2006, 2007, 2008). To execute an application on the SuperCISC fabric, the software kernels are converted into entirely combinational hardware functions represented by DFGs, generated automatically from C using a design automation flow (Jones et al., 2005, 2006; Hoare et al., 2006; Jones et al., 2006). Stripe-based hardware fabrics are designed to easily map DFGs from the application into the device. The architecture of the SuperCISC fabric (and other stripebased fabrics such as PipeRench) work in a similar way, retaining a data flow structure, which allows computational results to be computed in one multi-bit functional-unit (FU) and flow onto others in the system. FUs are organized into rows or *computational stripes*, within which each functional unit operates independently. The results of these operations are then fed into *interconnection stripes* which are constructed using multiplexers. Figure 2 illustrates this top-down data flow concept. The process of mapping these DFGs onto the SuperCISC fabric is described in the next section.

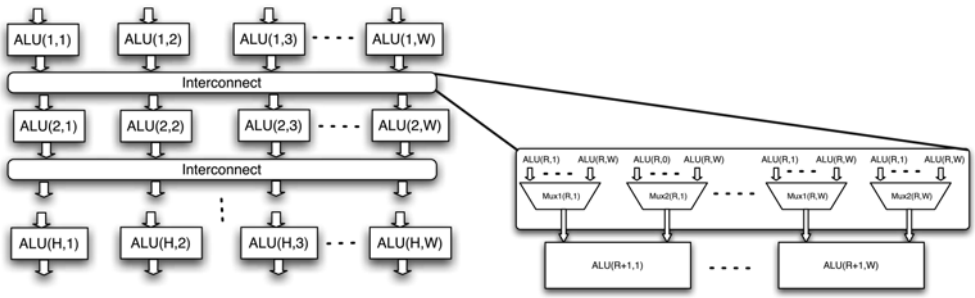
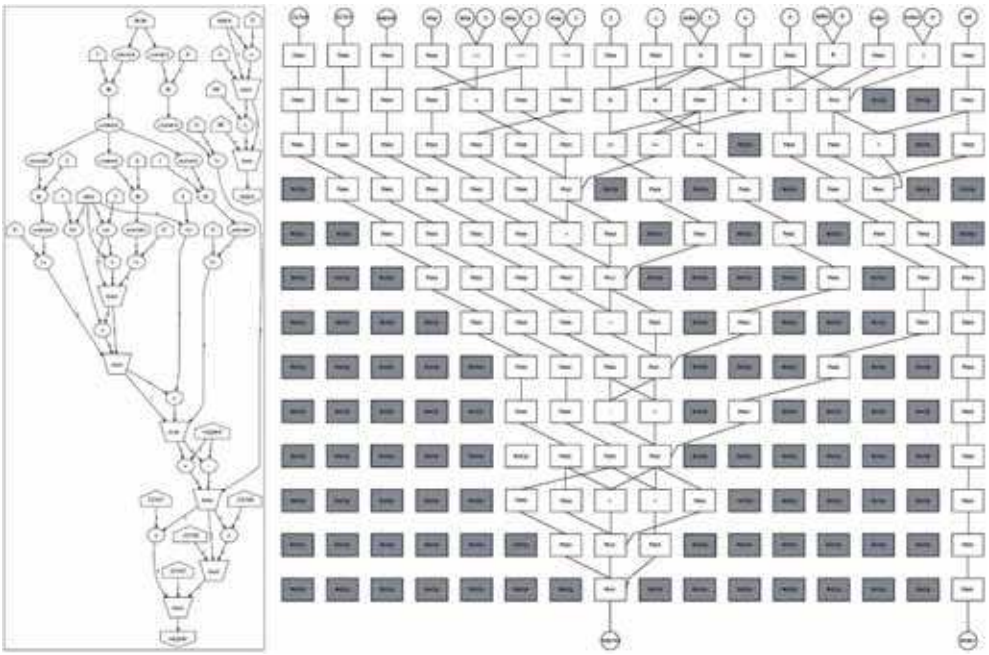


Fig. 2. Fabric conceptual model.

2.1 Mapping

A mapping of a DFG onto a fabric consists of an assignment of operators in the DFG to FUs in the fabric such that the logical structure of the DFG is preserved and the architectural constraints of the fabric are respected. This mapping problem is central to the use of the fabric since a solution must be available each time the fabric is reprogrammed for a different DFG. Because of the layered nature of the fabric, the mapping is also allowed to use FUs as "pass-gates," which take a single input and pass the input value to one or more outputs. In general, not all of the available FUs and edges will be used. An example DFG and a corresponding mapping are shown in Figure 3.



(a) Example data flow graph (DFG). (b) Example mapping.

Fig. 3. Mapping problem overview.

The interconnect design—that is, the pattern of available edges—is the primary factor in determining whether a given DFG can be mapped onto the fabric. For flexibility, it would make sense to provide a complete interconnect with each FU connected to every FU in the next row. The reason for limiting the interconnect is that the cardinality of the interconnect has a significant impact on energy consumption. Although most of the connections are unused, the increased cardinality of the interconnect requires more complicated underlying hardware, which leads to greater energy consumption. For a more detailed description of this phenomenon, see (Mehta et al., 2006), which indicates that this energy use can be significant. Therefore, we consider limited interconnects, which have better energy consumption but make the mapping problem more challenging.

We consider the mapping problem in three forms. We call these problems Minimum Rows Mapping, Feasible Mapping with Fixed Rows and Augmented Fixed Rows. These problems are briefly described in the following subsections.

2.1.1 Minimum rows mapping

Given a fixed width and interconnect design, a fabric with fewer rows will use less energy than one with more rows. As data flows through the device from top to bottom it traverses FUs and routing channels, consuming energy at each stage. The amount of energy consumed varies depending on the operation that an FU performs. However, even just passing the value through the FU consumes a significant amount of energy. Thus, the number of rows that the data must traverse impacts the amount of energy that is consumed. If the final result has been computed, the data can escape to an early exit, which bypasses the remaining rows of the fabric and reduces the energy required to complete the computation. Therefore, it is desirable to use as few rows as possible. Given a fabric width, fabric interconnect design, and data flow graph to be mapped, the Minimum Rows Mapping problem is to find a mapping that uses the minimum number of rows in the fabric. The mapping may use pass-gates as necessary.

We initially formulated a MILP to solve this problem, however, it has only been able to solve nearly trivial instances in a reasonable amount of time (Baz, 2008). We have since developed two heuristic approaches to solve this problem: a deterministic top-down greedy heuristic described in Section 4 and a heuristic that combines the top-down approach with randomization, described in Section 5.

2.1.2 Feasible mapping with fixed rows

One of the more complicated parts of creating a mapping is the introduction of pass-gates to fit the layered structure of the fabric. One approach that we have used is to work in two stages. In the first stage, pass-gates are introduced heuristically and operators assigned to rows so that all edges go from one row to the next. The second stage assigns the operators to columns so that the fabric interconnect is respected. This second stage is called Feasible Mapping with Fixed Rows. Note that depending on the interconnect design, there may or may not exist such a feasible mapping.

We have formulated a MILP approach to solve this problem described in detail in (Baz et al., 2008; Baz, 2008). This formulation can provide us with a lower bound with which to compare our heuristic solutions.

2.1.3 Augmented fixed rows

This problem first tries to solve the Feasible Mapping with Fixed Rows problem. If this is infeasible, then it may add a row of pass-gates to gain flexibility. It then tries to solve Feasible Mapping with Fixed Rows on the new problem. This is repeated until a solution is found or a limit is reached on the number of rows to add.

We have developed a partial sliding MILP heuristic in Section 6 to solve this problem.

2.1.4 Related work

There are two problems in graph theory related to the mapping problems we present. First, Feasible Mapping with Fixed Rows may be viewed as a special case of subgraph isomorphism, also called subgraph containment. The DFG (modified to have fixed rows) may be considered as a directed graph G, and the fabric may be considered as a directed graph H. The problem is to identify an isomorphism of G with a subgraph of H.

Most of the work on subgraph isomorphism uses the idea of efficient backtracking, first presented in (Ullmann, 1976). Examples of more recent work on the problem include

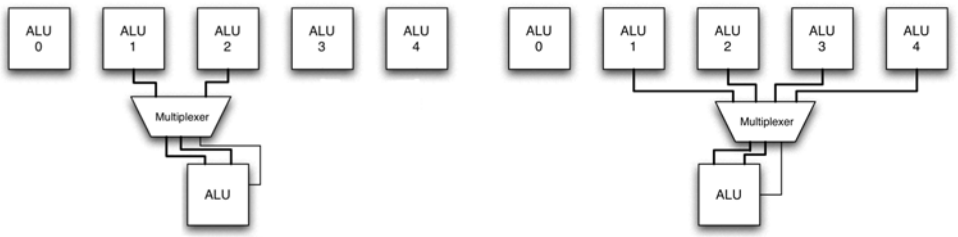
(Messmer & Bunke, 2000; Cordella et al., 2004; Krissinel & Henrick, 2004). In each of these cases, algorithms are designed to solve the problem for arbitrary graphs. In contrast, the graphs for our problem are highly structured, and our approaches take advantage of this structure. Subgraph isomorphism is NP-complete (Garey & Johnson, 1979).

If we fix the number of rows in the fabric, then finding a feasible mapping (but not minimizing the number of rows) may be viewed as a special case of a problem known as directed minor containment (Diestel, 2005; Johnson et al., 2001). The DFG may be considered as a directed graph G, and the fabric may be considered as a directed graph H. Directed minor containment (also known as butterfly minor containment) is the problem of determining whether G is a directed minor of H. Unlike subgraph isomorphism, G may be a directed minor without being a subgraph; additional nodes (corresponding to "pass-gates" in our application) may be present in the subgraph of H. Directed minor containment is also NP-complete. We are not aware of any algorithms for solving directed minor containment on general graphs or graphs similar to our fabric mapping problem.

2.2 Routing complexity

The fundamental parameter in the design of a coarse-grain reconfigurable device for energy reduction is the flexibility and resulting complexity of the interconnect. For example, a simpler interconnect can lead to architectural opportunities for energy reduction (fewer wires, simpler selectors, etc.) but can also make the mapping problem more difficult. As discussed in Section 2.1, the quality of the mapping solution also impacts the energy consumed by the design. Thus, to effectively leverage the architectural energy saving opportunities the mapping algorithms must become increasingly sophisticated.

As previously mentioned, the interconnection stripes are constructed using multiplexers. The cardinality of these multiplexers determines the routing flexibility and the maximum sources and destinations allowed for nodes in the DFG. This is shown in Figure 4. The interconnect shown in Figure 4(a) is built using 2:1 multiplexers, and is said to have a cardinality of two. Similarly, the interconnect in Figure 4(b) is comprised of 4:1 multiplexers, and is said to have a cardinality of four. By comparing these figures, it is obvious that the higher cardinality interconnect is more flexible because each functional unit can receive input from a larger number of sources. Essentially, a higher cardinality interconnect has fewer restrictions, which leads to a simpler mapping problem.



(a) Cardinality of two. (b) Cardinality of four.

Fig. 4. Interconnects of two different multiplexer cardinalities.

While the flexibility of higher cardinality multiplexers is desirable for ease of mapping, these multiplexers are slower, more complex, and dissipate more power than lower cardinality multiplexers. A detailed analysis of the power consumption versus cardinality is conducted in (Jones et al., 2008; Mehta et al., 2007, 2006).

Additionally, when mapping a DFG to a stripe-style structure, data dependency edges often traverse multiple rows. In these fabrics, FUs must often pass these values through without doing any computation. We call these operations in the graph, pass-gates. However, these FUs used as pass-gates are an area and energy-inefficient method for vertical routing. Thus, we explored replacing some percentage of the FUs with dedicated vertical routes to save energy (Mehta et al., 2008; Jones et al., 2008). However, these dedicated pass-gates can make mapping more difficult because it places restrictions on the placement of operators. The work in (Mehta et al., 2008; Jones et al., 2008) only uses the first of the three greedy heuristics presented here and required that the interconnect flexibility be relaxed when introducing dedicated vertical routes. The more sophisticated greedy algorithms were designed in part to improve the mapping with the more restrictive multiplexer cardinalities along with dedicated pass-gates.

The purpose of these heuristics is to provide high quality of solution mappings onto the low-energy reconfigurable device. One way to measure the effectiveness is to examine the energy consumed from executing the device with various architectural configurations and different data sets, etc. We obtain these energy results from extremely time consuming power simulations using computer-aided design tools. However, in this paper we chose to focus our effort on achieving a high quality of solution from the mapping algorithms. Conducting power simulations for each mapping would significantly limit the number of mapping approaches we could consider.

Thus, we can examine two factors to evaluate success: the increase in the total path length of the mapped algorithm and the number of FUs used as pass-gates. The total path length in the mapped design is the sum of the number of rows traversed from each input to each output. Thus, the path length increase is the increase in the total path length from a solution where each computation is completed as early as possible limited only by the dependencies in the graph (see Section 4.1). The number of FUs used as pass-gates is useful in judging success in cases where the fabric contains dedicated pass-gates. Dedicated pass-gates are more energy efficient than complex functional units at passing a value (more than an order of magnitude (Jones et al., 2008)). Thus, when using dedicated-pass gates the fewer Fus used as pass-gates, the better.

To demonstrate that these factors influence the energy consumption of the device, we ran a two-way analysis of variance (ANOVA) on the energy with the number of FUs used as pass-gates and path length as factors to determine the correlation. Using an alpha value of 0.05, both factors significantly influenced the energy ($p<0.01$ and $p=0.031$, respectively).

3. The Fabric Interconnect Model (FIM)

As various interconnection configurations were developed, redesigning the mapping flow and target fabric hardware by hand for each new configuration was impractical (Mehta et al., 2007). Additionally, we envision the creation of customizable fabric intellectual property (IP) blocks that can be used in larger system-on-a-chip (SoC) designs. To make this practical, it is necessary to create an automation flow to generate these custom fabric instances.

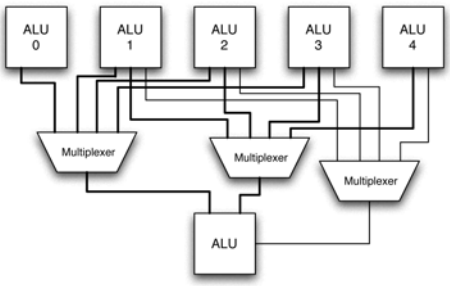
To solve this problem, we created the FIM, a textual representation used to describe the interconnect and the layout and make-up of the FUs in the system. The FIM becomes an input file to the mapper as well as the tool that generates a particular instance of the fabric with the appropriate interconnect.

The FIM file is written in the Extensible Markup Language (XML) (Bray et al., 2006). XML was selected as it allowed the FIM specification to easily evolve as new features and

descriptions were required. For example, while the FIM was initially envisioned to describe the interconnect only, it has evolved to describe dedicated pass-gates and other heterogeneous functional unit structures.

Figure 5(a) shows an example partial FIM file that describes a cardinality five interconnect. A cardinality five interconnect is a specially designed interconnect, which is actually constructed using mirrored 4:1 multiplexers. In Figure 4(b) a single multiplexer is depicted as providing all three inputs to each FU, also known as an ALU (arithmetic logic unit). In reality, each of the three inputs has its own individual multiplexer. By allowing the multiplexers to draw their inputs from different locations, 4:1 multiplexers can be used to create the illusion of a limited 5:1 multiplexer. This limited 5:1 multiplexer provides a surprisingly higher flexibility over a cardinality four interconnect with no cost in terms of hardware complexity.

```
<rowpattern repeat="forever">
 <row>
  <ftupattern repeat="forever">
   <FTU type="ALU">
    <operand number="0">
     <range left ="-2" right ="1"/>
    </operand>
    <operand number="1">
     <range left ="-1" right ="2"/>
    </operand>
    <operand number="2">
     <range left ="-1" right ="2"/>
    </operand>
   </FTU>
  </ftupattern>
 </row>
</rowpattern>
```



(a) FIM file example for 5:1 style interconnect. (b) 5:1 style interconnect implementation.

Fig. 5. Describing a 5:1 multiplexing interconnect using a FIM file.

The pattern in Figure 5(a) repeats the interconnect pattern for `ALU`, whose zeroth operand can read from two units to the left, the unit directly above, and one unit to the right. The first operand is the mirror of the zeroth operand, reading from two units to the right, the unit directly above, and one unit to the left. The second operand, which has the same range as the first operand, serves as the selection bit if the FU is configured as a multiplexer. The resulting cardinality five interconnect implementation is shown in Figure 5(b). As specified in the FIM, the zeroth operand of `ALU` can access `ALU0` through `ALU3`, while the first and second operands can access `ALU1` through `ALU4`.

The ranges in the FIM can be discontinuous by supplying additional range flags. The file can contain a heterogeneous interconnect by defining additional Fabric Topological Units (`FTUs`) with different interconnect ranges. The pattern can either repeat or can be arbitrarily customized without a repeating pattern for a fixed size fabric.

The design flow overview using the FIM is shown in Figure 6. The SuperCISC Compiler (Hoare et al., 2006; Jones et al., 2006) takes C code input, which is compiled and converted into a Control and Data Flow Graph (CDFG). A technique known as *hardware predication* is applied to the CDFG in order to convert control dependencies (e.g. *if-else* structures) into data dependencies through the use of selectors. This post-predication CDFG is referred to as

a *Super Data Flow Graph* (SDFG). The SDFG is then mapped into a configuration for the fabric described by the FIM.

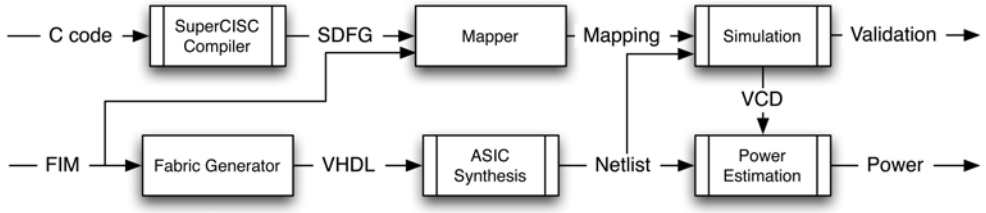


Fig. 6. Interconnect evaluation tool flow.

The FIM is also used to automatically generate a synthesizable hardware description of the fabric instance described by the FIM. For testing and energy estimation, the fabric instance can be synthesized using commercial tools such as Synopsys' Design Compiler to generate a netlist tied to ASIC standard cells. This netlist and the mapping of the application are then fed into ModelSim where correctness can be verified. The mapping is communicated to the simulator to program the fabric device in the form of ModelSim `do` files. A value change dump (VCD) file output from the simulation of the design netlist can then be used to determine the power consumed in the design. However, due to the effort required to generate a single power result we will use mapping quality metrics such as path length increase and FUs used as pass-gates rather than energy consumption to evaluate the quality of our mapping heuristics as described in Section 2.2.

The FIM is incorporated into the mapping flow as a set of restrictions on both the interconnect and the functional units in each row. In addition to creating custom interconnects, the FIM can be used to introduce heterogeneity into the fabric's functional units. This capability is used to allow the introduction of dedicated pass-gates into the target architecture and greedy mapping approaches.

4. Deterministic greedy heuristic

A heuristic mapping algorithm overviewed in Algorithm 1 was developed to solve the problem of Minimum Rows Mapping. The instantiation of this algorithm reads both the DFG and the FIM to generate its mapping result. The heuristic is comprised of two stages of row assignment followed by column assignment, which follows a top-down mapping approach using a limited look-ahead of two rows. In the first line of the algorithm each node is assigned to a row as described in Section 4.1. In the second stage, as shown in the algorithm, the column locations for nodes in each row are assigned starting with the top row. This is described in Section 4.2. After each row is mapped, the heuristic will not modify the locations of any portion of that row.

While the limited information available to the heuristic does not often allow it to produce optimal minimum-size mappings, its relative simplicity provides a fast runtime. By default the heuristic tries to map the given benchmark to a fabric with a width equal to the largest individual row, and a height equal to the longest path through the graph representing the input application. Although the width is static throughout a single mapping, the height can increase as needed.

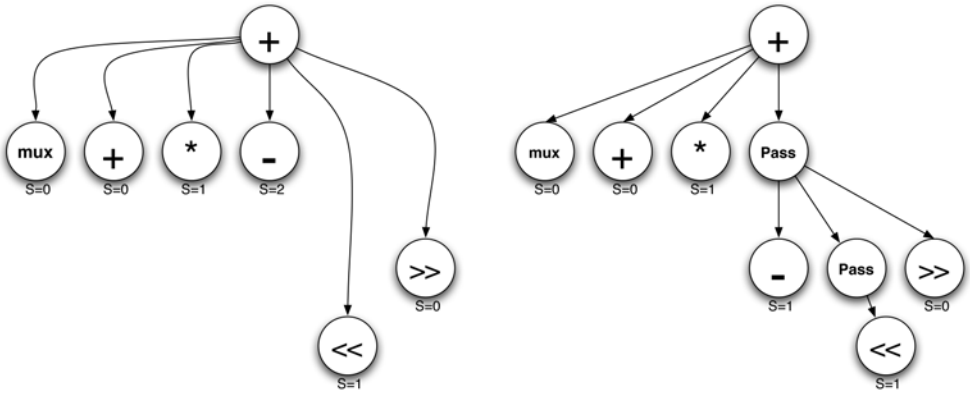
Algorithm 1 Deterministic Heuristic Mapping Algorithm

```
1: Create initial row assignments based on as-soon-as-possible layout.
2: r ← 1
3: while operators are assigned to row r do
4:     Assign Columns in Row r (see Algorithm 2), possibly pushing some operators to later rows.
5:     [pass-gate centering (see description in text).]
6:     r ← r + 1
7: end while
```

4.1 Row assignment

Initially, the row of each node is set to its row assignment in an as soon as possible (ASAP) "schedule" of the graph. Beginning with the first row and continuing downward until the last, each node in the given row is checked to determine if any of its children are non-immediate (i.e. the dependency edge in the DFG spans multiple rows) and as a result they cannot be placed in the next row. If any non-immediate children are present, a pass-gate is created with an edge from the current node. All non-immediate children nodes are disconnected from the current node and connected to the pass-gate. This ensures that after row assignment, there are no edges that span multiple rows of the fabric.

After handling the non-immediate children, each node is checked to determine if its fanout exceeds the maximum as defined by the FIM. If a node's fanout exceeds the limit, a pass-gate is created with an edge from the current node. In order to reduce the node's fanout, children nodes are disconnected from the current node and connected to the pass-gate. To minimize the number of additional rows that must be added to the graph we first move children nodes with the highest slack from the current node to the pass-gate. If the fanout cannot be reduced without moving a child node with a slack of zero, then the number of rows in the solution is increased by one causing an increase of one slack to all nodes in the graph. This process continues for each node in the current row, then subsequently for all rows in the graph as shown in Figure 7. Once row assignment is complete, the minimum fabric size for each benchmark is known. These minimum sizes are shown in Table 1.



(a) Before row assignment. (b) After row assignment.

Figure 7. Row assignment example showing pass-gate insertion.

	GSM	ADPCM Encoder	ADPCM Decoder	IDCT Column	IDCT Row	Sobel	Laplace
Fabric Width	14	17	16	20	17	10	15
Fabric Height	18	16	13	12	10	9	8

Table 1. Minimum fabric sizes with no interconnect constraints.

4.2 Column assignment

The column assignment of the heuristic follows Algorithms 2–4 where items in square brackets [] are included in the optimized formulation. Many of these bracketed items are described in Section 4.3. During the column assignment for each row, the heuristic first determines viable locations based on the dependencies from the previous row. Then, the heuristic considers the impact of dependencies of nodes in the two following rows. The heuristic creates location windows that describe these dependencies as follows:

The *parent dependency window* (PDW) lists all FU locations that satisfy the primary constraint that the current node must be placed such that it can connect to each of its inputs (parents) with the interconnect specified in the FIM. The construction of the PDW is based on the location of each parent node, valid mapping locations due to the interconnect, and the operations supported by each FU (e.g. computational FU versus dedicated pass-gate). Figure 8 shows an example of a PDW dictated by the interconnect description. In this example, an operation that depends on the result of the subtraction in column 6 and addition in column 8 can only be placed in either `ALU 6` or `ALU 7` due to the restrictions of cardinality four interconnect.

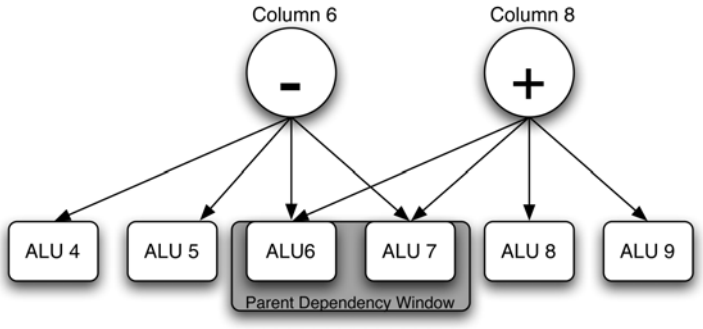


Fig. 8. Parent dependency window.

The *child dependency window* (CDW) lists all FU locations that satisfy the desired but non-mandatory condition that a node be placed such that each of its children nodes in the proceeding row will have at least one valid placement. The construction of the CDW is based on the PDW created from the potential locations of a current node as well as the PDW created from potential locations of any nodes that share a direct child with the current node. Nodes which share a direct child are referred to as *connected nodes*. Again the FIM is consulted to determine if there will be any potential locations for the children nodes based on the locations of the current node and connected nodes. A child dependency window example is shown in Figure 9. In this example, a left shift operation and a right shift operation are being assigned columns. Due to parent dependency window constraints, the

Algorithm 2 Assign Columns in Row r

```
1: Priority set ← ∅
2: while not all operators have column assignments do
3:    Calculate PDW, CDW, [and GDW] for all unplaced operators.
4:    For each unplaced operator, calculate FU Desirability [and Potential Connectivity] for each FU in its PDW.
5:    if there is a unary operator in the Priority set with empty PDW then
6:       Mapping has failed. Abort the process.
7:    else if there is a non-unary operator in the Priority set with empty PDW then
8:       Push it to the next row (r + 1).
9:       Create pass-gates for each of its inputs in the current row.
10:      Cancel all column assignments (for this row).
11:   else if there is an operator not in the Priority set with empty PDW then
12:      Add it to the Priority set.
13:      Cancel all column assignments (for this row).
14:   else
15:      Choose an operator i to map according to the following priorities: those in Priority set with |PDW| = 1 and
         smallest |CDW|, those in Priority set with smallest |CDW|, those not in Priority set with |PDW| = 1 and
         smallest |CDW|, those not in Priority set with smallest |CDW|. [Break any ties based on |GDW|.]
16:      Make Column Assignment for Operator i (see Algorithm 3).
17:   end if
18: end while
```

left shift can be placed in either `ALU 10` or `ALU 11`. Similarly, the right shift can be placed in either `ALU 6` or `ALU 7`. There is a third node (not pictured) which takes its inputs from the two shift operations. In order for this shared child to have a valid placement, the left shift must be placed in `ALU 10` and the right shift must be placed in `ALU 7`. Using this placement, the shared child will have a single possible placement in its PDW, `ALU 8`.

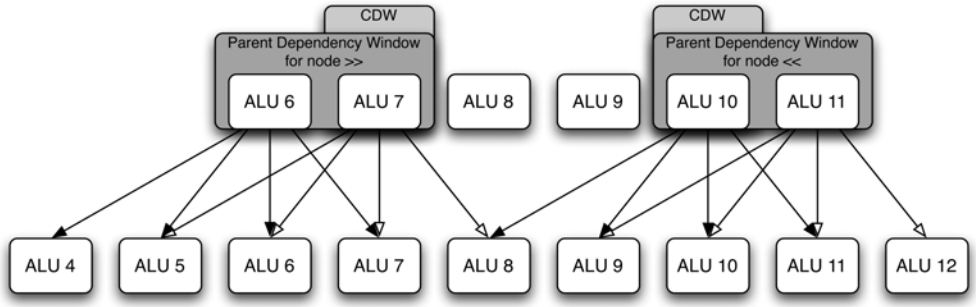


Fig. 9. Child dependency window.

The grandchild dependency window (GDW) provides an additional row of look-ahead. The GDWlists all FU locations that satisfy the optional condition that a node be placed such that children nodes two rows down (grandchildren) will have at least one valid placement. It is constructed using the same method as the CDW.

As nodes are mapped to FU locations, newly taken locations are removed from the dependency windows of all nodes (since no other node can now take those locations), and the child and grandchild windows are adjusted to reflect the position of all mapped nodes. In addition to tracking the PDWs, CDWs, and GDWs of each node, a desirability value is associated with each location in the current row. The desirability value is equal to the number of non-mapped nodes that contain the location in their PDW, CDW, or GDW.

Algorithm 3 Make Column Assignment for Operator i

```
if |PDW| = 1 then
    Assign to the unique column in PDW.
else {|PDW| > 1}
    if |CDW| = 1 then
        Assign to the unique column in CDW.
    else if |CDW| > 1 [and no shared grandchildren] then
        Assign to the column in the CDW with lowest FU Desirability, [ties broken by highest Potential Connectivity,
        further ties broken by lowest Distance to Center.]
    else if |CDW| > 1 and shared grandchildren, or shared grandchildren but no shared children then
        [Use Algorithm 4 to determine column assignment.]
    else if shared children but |CDW| = 0 then
        [Compute Nearness Measure for each FU in PDW based on common children.]
        [Assign to the column with the highest Nearness Measure.]
    else if i does not share children [or grandchildren] with other operators then
        Assign to the column in the PDW with lowest FU Desirability.
    end if
end if
```

Algorithm 4 Assignment Based on Shared Grandchildren

```
if |GDW| = 1 then
    Assign to the unique column in GDW.
else if |GDW| > 1 then
    Assign to the column in the GDW with highest Potential Connectivity, ties broken by lowest FU Desirability,
    further ties broken by lowest Distance to Center.
else if |GDW| = 0 then
    Compute Nearness Measure for each FU in CDW based on common grandchildren.
    Assign to the column with highest Nearness Measure.
end if
```

The mapper then places each node one at a time. To select the next node to place, the mapper first checks for any nodes with an empty PDW, then for any nodes with a PDW that contains only one location. Then it checks for any high-priority nodes in the current row, as these are nodes designated as difficult to map. Finally, it selects the node with the smallest CDW, most connected nodes, and lowest slack. This node is then placed within the overlapping windows while attempting to minimize the negative impact to other nodes.

Column placement also uses the concept of a *priority set*. In the process of placing operators, the algorithm may find that an operator has become impossible to place. If this happens, the algorithm is placed into the priority set and column placement for the row is restarted. Operators in the priority set are placed first. Even then, it may be impossible to place some operators. The last resort for the algorithm is to reassign the operator to the next row and add pass-gates to the current row for the operator's inputs. Unary operators cannot be reassigned because placing the pass-gate for the input would also be impossible. If a unary operator (or pass-gate) in the priority set cannot be placed, then the algorithm aborts.

4.3 Extensions

The initial algorithm was not always able to produce high quality mappings for some of the benchmarks when using more restrictive interconnects such as 5:1. Several extensions to the heuristic were implemented in an effort to increase its effectiveness.

Potential Connectivity: When determining the location to place an operator we consider which locations provide the most potential connectivity to child operators.

Potential connectivity is defined as the number of locations each shared child operation could be placed when the current operation is placed in a particular location.

Nearness Measure: This measure is used when an operator has shared children but the CDW is empty. The goal is to push the operators which share a child as close together as possible; this allows the algorithm to eventually place the child operators in some later row. The measure is the sum of the inverses of the distances from the candidate FU to the operators with common children.

Distance to Center: Used as a tie-breaker only to prefer placing operators closer to the center of the fabric.

Pass-gate centering: The initial algorithm tended to push pass-gates that have no shared child operators toward the edges of the fabric. This makes it harder for their eventual non-pass-gate descendants to be mapped, since their pass-gate parent is so far out of position. After placing an entire row the mapper pushes pass gates toward the center by moving them into unassigned FUs. This is the extension shown in Algorithm 1.

4.4 Results

Higher cardinality interconnects such as 8:1 and higher were easily mapped using the deterministic greedy algorithm. We show results using a 5:1-based interconnect as it exercised the algorithm well. The mapper was tested on seven signal and image processing benchmarks from image and signal processing applications. A limit of 50 rows was used to determine if an instance was considered un-mappable with the given algorithm. Mapping quality was judged on three criteria. The first is fabric size, represented in particular by the number of rows in the final solution. The second is total path length, or the sum of the paths from all inputs to all outputs as described in Section 2.2. The third metric is mapping time, which is the time it takes to compute a solution.

The fabric size is perhaps the most important factor in judging the quality of a solution. The number of columns is more or less fixed by the size of the largest row in a given application. However, the number of additional rows added to the minimum fabric heights listed in Table 1 reflects directly on the capability of the mapping algorithm. Smaller fabric sizes are desirable because they require less chip area, execute more quickly, and consume less power.

As described in Section 2.2, the total path length increase is a key factor in the energy consumption of the fabric executing the particular application. However, fabric size and total path length are related. A mapping with a smaller fabric size will typically have a considerably smaller total path length and thus, also have a lower energy consumption. Thus, the explicit total path length metric is typically most important when comparing mappings with a similar fabric size.

The mapping time is important because it evaluates practicality of the mapping algorithm. Thus, the quality of solution of various mapping algorithms is traded off against the execution time of the algorithms when comparing these mapping algorithms.

We compared two versions of the greedy algorithm. The initial algorithm makes decisions based on the PDW and the CDW and uses functional unit desirability to break ties. This heuristic is represented by Algorithms 1–3 without the sections denoted by square brackets []. The final version of the algorithm is shown in Algorithms 1–4 including the square

bracket [] regions. This version of the heuristic builds upon the initial algorithm by including the GDW, potential connectivity, and centering. The results of the comparison are shown in Table 2.

		GSM	ADPCM Encoder	ADPCM Decoder	IDCT Column	IDCT Row	Sobel	Laplace
Initial Algorithm	Rows Added	3	13	11	*no solution*	*no solution*	1	2
	Time (s)	18	79	9	37	15	< 1	< 1
Final Algorithm	Rows Added	1	5	1	8	4	1	2
	Time (s)	< 1	12	< 1	1	< 1	< 1	< 1

Table 2. Number of rows added and mapping times for the greedy heuristic mapper using a 5:1 interconnect.

Using the initial algorithm, Sobel, Laplace, and GSM can be solved fairly easily, requiring only a few added rows in order to find a solution. However, the solutions for ADPCM Encoder and Decoder require a significant number of additional rows and both IDCT-based benchmarks were deemed unsolvable.

The final algorithm is able to find drastically better solutions more quickly. For example the number of rows added for ADPCM Encoder and Decoder went from 13 to 5 and 11 to 1, respectively. It is also able to find feasible solutions for IDCT Row and IDCT Column. For the other four benchmarks, the final algorithm performs equally well or better than the initial algorithm. The final algorithm is faster in every case decreasing the solution time for all benchmarks to within 1 second except ADPCM Encoder which was reduced from 79 to 12 seconds.

We tried the final deterministic algorithm on a variety of more restrictive interconnects including a cardinality five interconnect with every third FU (33%)replaced with a dedicated pass-gate. The results are shown in Table 3. The fabric size results are actually quite similar in terms of rows added to the 5:1 cardinality interconnect without dedicated pass-gates.

		GSM	ADPCM Encoder	ADPCM Decoder	IDCT Column	IDCT Row	Sobel	Laplace
Final Algorithm	Rows Added	1	6	2	7	5	0	2
	Time (s)	< 1	7	< 1	2	1	< 1	< 1

Table 3. Greedy heuristic mapper results using a 5:1 interconnect and 33% dedicated pass-gates.

While the deterministic heuristic provides a fast valid mapping, it does add a considerable number of rows from the ASAP (optimal) configuration, which leads to considerable path length increases and energy overheads. In the next section we explore a technique to improve the quality of results through an iterative probabilistic approach.

5. Greedy heuristic including randomization

Another flavor of greedy algorithms are greedy randomized algorithms. Greedy randomized algorithms are based on the same principles guiding purely greedy algorithms, but make use of randomization to build different solutions on different runs (Resende & Ribeiro, 2008b). These algorithms are used in many common meta-heuristics such as local

search, simulated annealing, and genetic algorithms (Resende & Ribeiro, 2008a). In the context of greedy algorithms, randomization is used to break ties and explore a larger portion of the search space. Greedy randomized algorithms are often combined with multi-iteration techniques in order to enable different paths to be followed from the same initial state (Resende & Ribeiro, 2008b).

The final version of the deterministic greedy algorithm is useful due to its fast execution time and the reasonable quality of its solutions. However, because it is deterministic it may get stuck in local optimums which prevent it from finding high quality global solutions. By introducing a degree of randomization into the algorithm, the mapper is able to find potentially different solutions for each run. Additionally, since the algorithm runs relatively quickly, it is practical to run the randomized version several times and select the best solution.

The column assignment phase of the mapping algorithm was chosen as the logical place to introduce randomization. This area was selected as the column assignments not only affect the layout of the given row, but also affect the layouts of subsequent rows. In the deterministic algorithm, nodes are placed in an order determined by factors including smallest PDW, CDW, GDW, etc. and once placed, a node cannot be removed. In contrast, the randomized heuristic can explore random placement orders, which leads to much more flexibility.

We investigated two methods for introducing randomization into the mapping heuristic. The first approach makes ordering and placement decisions completely randomly. We describe this approach in Section 5.1. The second leverages the information calculated in the deterministic greedy heuristic by applying this information as weights in the randomization process. Thus, the decisions are encouraged to follow the deterministic decision but is allowed to make different decisions with some probability. We describe this approach in Section 5.2.

5.1 Randomized heuristic mapping

The biggest difference between the deterministic heuristic and the heuristics that incorporate randomization is that the deterministic is run only once and the random oriented heuristics are run several times to explore different solutions. The basic concept of the randomized heuristic is shown in Algorithm 5. First the deterministic algorithm is run to determine the initial "best" solution. Then the randomizer mapper is run a fixed number of times determined by I. If an iteration discovers a better quality solution (better height or same height and better total path length) it is saved as the new "best" solution. This concept of saving and restoring solutions is common in many multi-start meta-heuristics, including simulated annealing and greedy randomized adaptive search procedures (GRASP) (Resende & de Sousa, 2004).

Algorithm 5 Randomized Heuristic Mapping Algorithm

1: Execute deterministic heuristic mapper (Algorithm 1) to create solution S and determine height H and total path length P
2: **for** I iterations **do**
3: Execute random heuristic mapper to determine solution s and calculate height h and total path length p
4: **if** $h < H$ or $(h = H$ and $p < P)$ **then**
5: $S \leftarrow s, H \leftarrow h, P \leftarrow p$
6: **end if**
7: **end for**

The randomized mapping heuristic follows the same algorithmic design as the deterministic heuristic from Algorithm 2. The only major change is to line 15, in which the new algorithm selects the next node to map in a column randomly and ignores all other information.
Although the introduction of randomization allows the mapper to find higher quality solutions, it also discovers many lower quality solutions, which often take a long time to complete. In order to mitigate this problem, one other divergence from the deterministic algorithm allows the mapper to terminate a given iteration once the fabric size of the current solution becomes larger than the current best solution.

5.2 Weighted randomized heuristic mapping

Using entirely random placement order did discover better solutions (given enough iterations) than the deterministic heuristic. Unfortunately, the majority of the solutions discovered were of poorer quality than the deterministic approach. Thus, we wanted to consider a middle ground algorithm that was provided some direction based on insights from the deterministic algorithm but also could make other choices with some probability. This resulted in a weighted randomized algorithm.
Weights are calculated based on the deterministic algorithm concepts of priorities and dependency windows. Again the modification of the basic deterministic algorithm to create the weighted randomized algorithm is based on line 15 of Algorithm 2. The weighted randomized algorithm replaces this line with Algorithm 6 to select the next node to place. The algorithm begins by dividing the unplaced operators into sets based on their PDW size. Each set is then assigned a weight by dividing its PDW size by the sum of all of the unique PDW sizes. Because nodes with small parent dependency windows are more difficult to place, it is necessary to assign them a larger weight. This is accomplished by subtracting the previously computed weight from one. Each set is then further subdivided in a similar fashion based first on CDW sizes and then node slack. The result of this operator grouping process is a weighted directed acyclic graph (DAG) with a single vertex as its root. Starting at the root, random numbers are used traverse the weighted edges until a leaf vertex is reached, at which point an operator will be selected for column assignment.

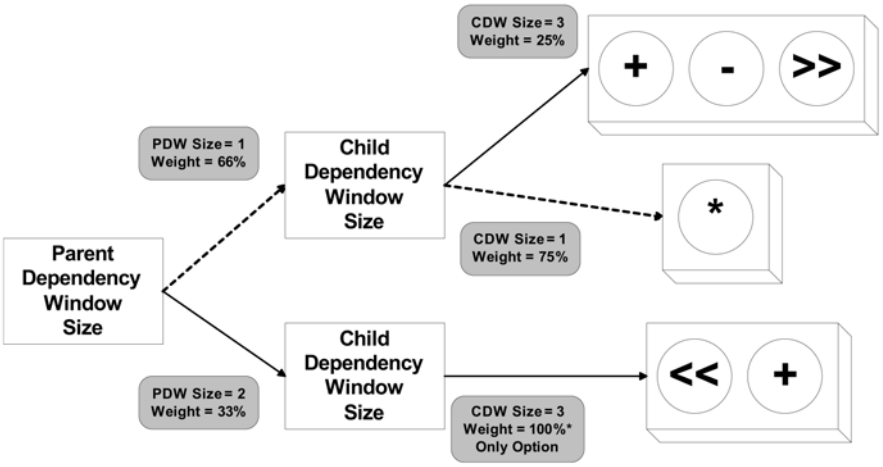


Fig. 10. Heuristic weight system.