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.

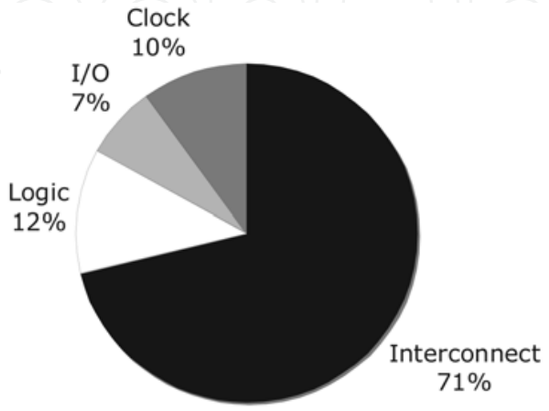


Fig. 1. Power consumption features of a Xilinx Virtex-2 3000 FPGA (Sheng et al., 2002).

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.

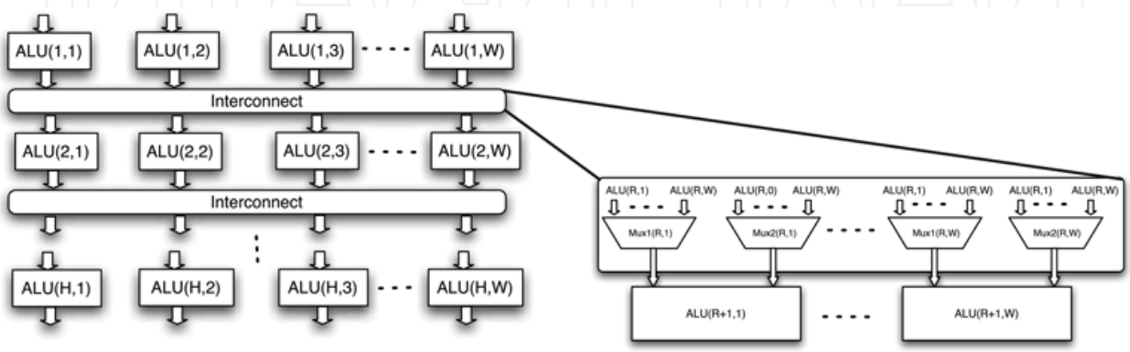
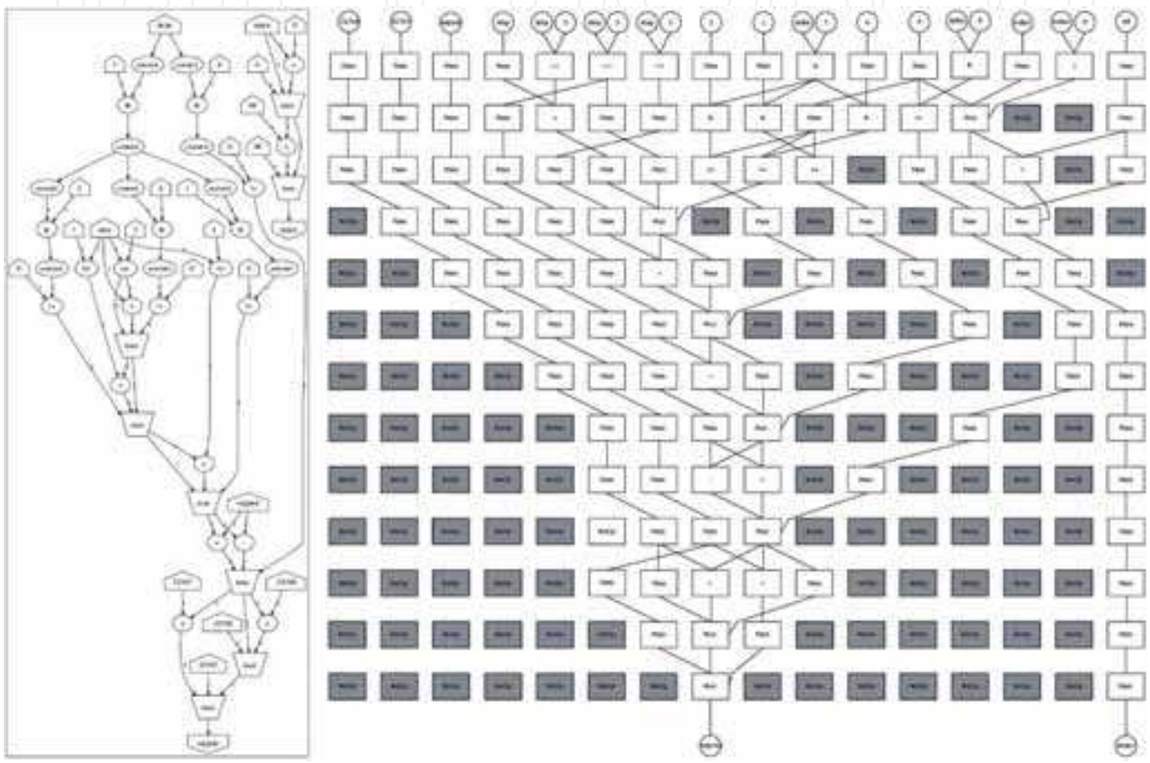


Fig. 2. Fabric conceptual model.

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

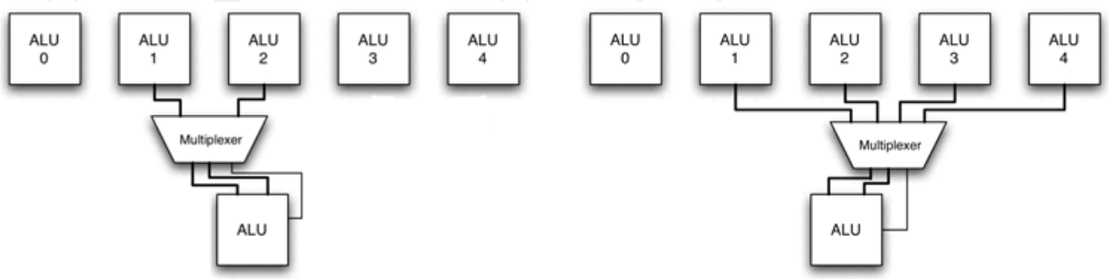
(Messmer & Bunke, 2000; Cordella et al., 2004; Krissinel & Henrick, 2004). In each of these cases, algorithms are designed to solve the problem for arbitrary graphs. In contrast, the graphs for our problem are highly structured, and our approaches take advantage of this structure. Subgraph isomorphism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

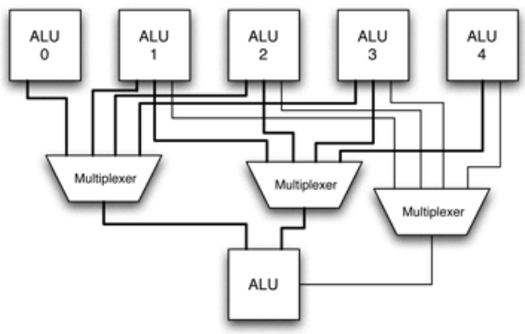
To solve this problem, we created the FIM, a textual representation used to describe the interconnect and the layout and make-up of the FUs in the system. The FIM becomes an input file to the mapper as well as the tool that generates a particular instance of the fabric with the appropriate interconnect.

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.

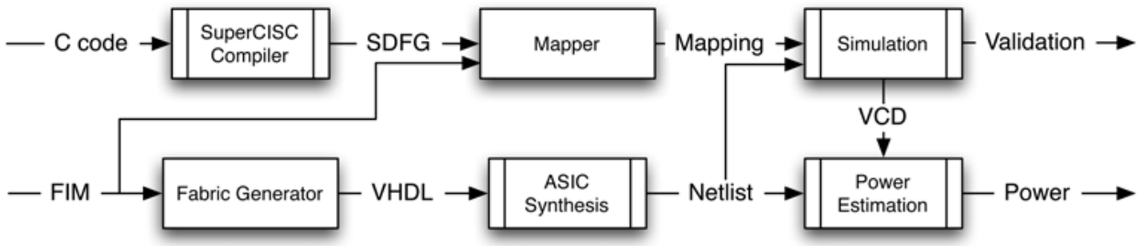


Fig. 6. Interconnect evaluation tool flow.

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.

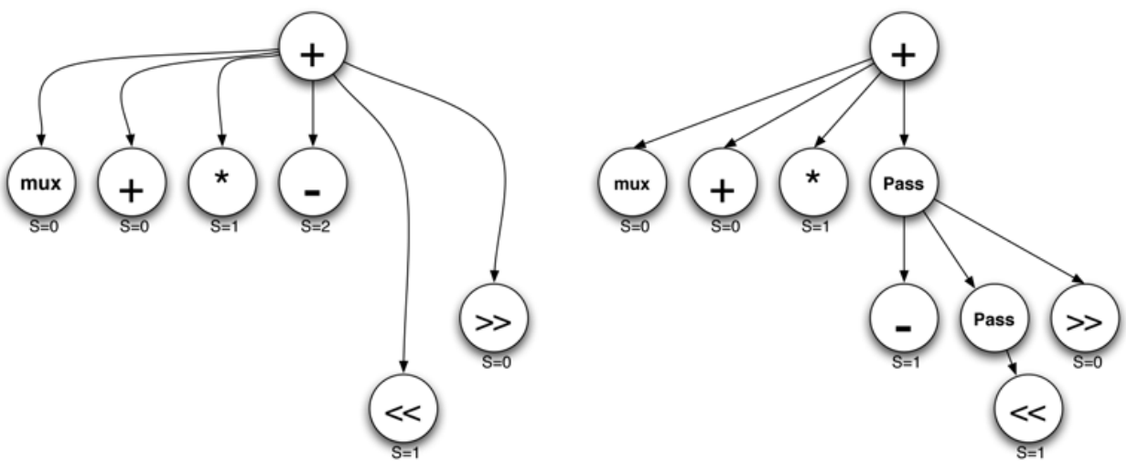
Algorithm 1 Deterministic Heuristic Mapping Algorithm

```
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.

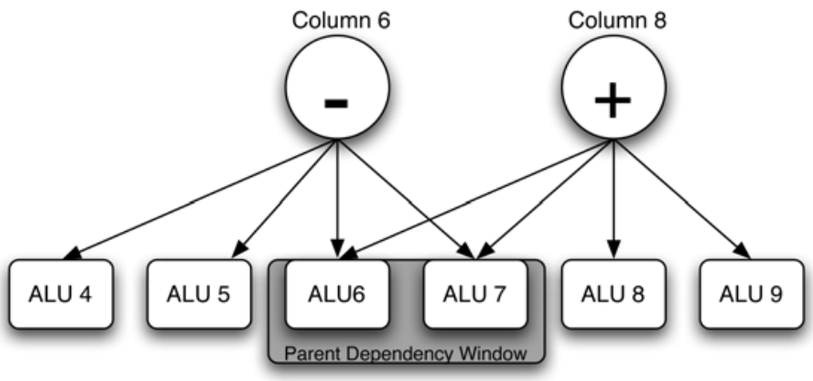


Fig. 8. Parent dependency window.

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`.

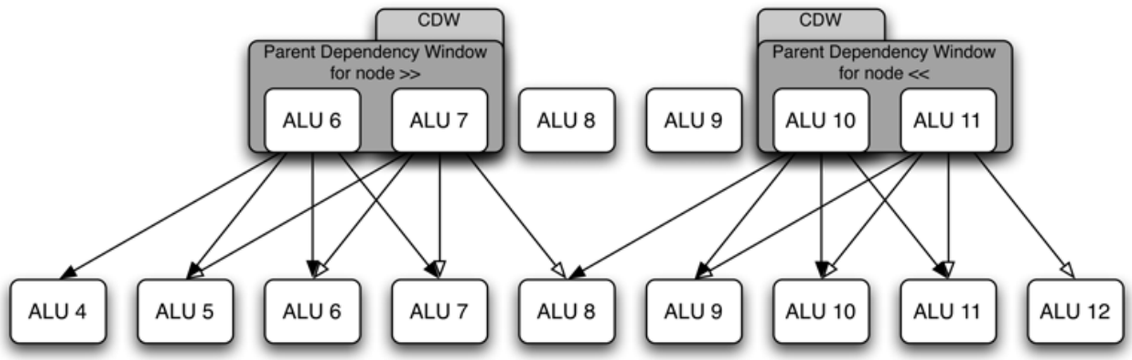


Fig. 9. Child dependency window.

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	Rows Added	3	13	11	*no solution*	*no solution*	1	2
Algorithm	Time (s)	18	79	9	37	15	< 1	< 1
Final	Rows Added	1	5	1	8	4	1	2
Algorithm	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	Rows Added	1	6	2	7	5	0	2
Algorithm	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.

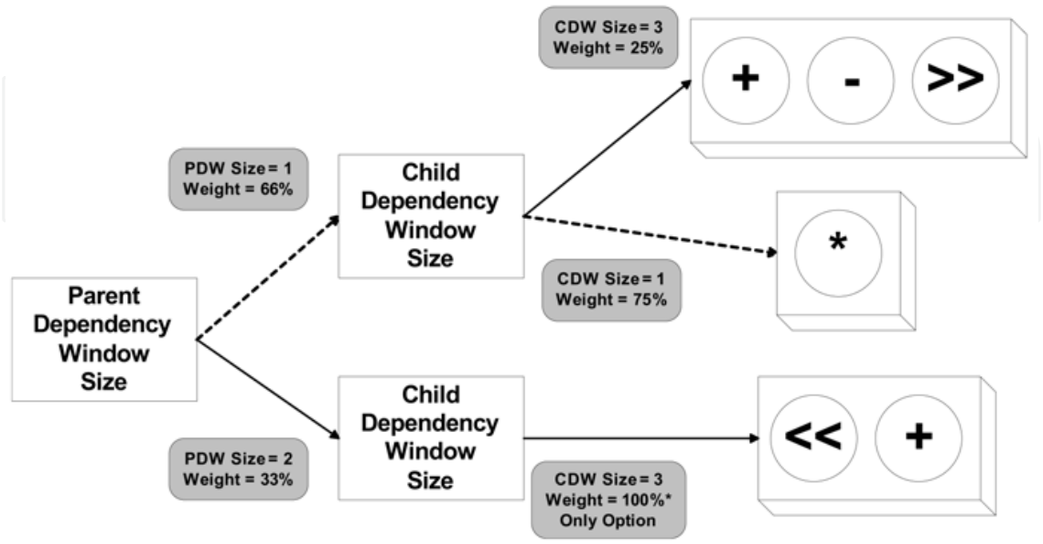


Fig. 10. Heuristic weight system.

An example of the weighting system used in the randomized mapper is shown in Figure 10. In this example, nodes priorities are assigned based on PDW size followed by CDW size. Slack is not considered in this example for simplicity. The deterministic heuristic would always assign the highest priority to the multiplier because it has the smallest parent window as well as the smallest child window. In Figure 10 this behavior is indicated by the dashed arrows. By introducing probability into the heuristic, the multiplier is still given the highest priority with a 50% chance of being selected. However, a node from the top group has a 17% chance of being selected and a node from the bottom group has a 33% chance of being selected instead. In the event that several nodes are assigned the same priority level, one node is chosen randomly with equal weight from the set.

Algorithm 6 Weighted Randomized Heuristic Operator Selection Algorithm

```
1: sum_p ← 0
2: for each unplaced operator OP do
3:    if W_|PDW| is ∅ then
4:       sum_p ← sum_p + |PDW|
5:    end if
6:    add OP to W_|PDW|
7: end for
8: Assign normalized weights based on formula 1 − (|PDW| ÷ sum_p)
9: for each key i in W do
10:   sum_c,i ← 0
11:   for each operator OP in W_i do
12:      if W_i,|CDW| is ∅ then
13:         sum_c,i ← sum_c,i + |CDW|
14:      end if
15:      add OP to W_i,|CDW|
16:   end for
17:   Assign normalized weights based on formula 1 − (|CDW| ÷ sum_c,i)
18: end for
19: for each key i in W do
20:   for each key j in W_i do
21:      sum_s,i,j ← 0
22:      for each operator OP in W_i,j do
23:         if W_i,j,slack is ∅ then
24:            sum_s,i,j ← sum_s,i,j + slack
25:         end if
26:         add OP to W_i,j,slack
27:      end for
28:      Assign normalized weights based on formula 1 − (slack ÷ sum_s,i,j)
29:   end for
30: end for
31: Select W_i based on weighted random selection of i
32: Select W_i,j based on weighted random selection of j
33: Select W_i,j,k based on weighted random selection of k
34: Select a node randomly from W_i,j,k for placement
```

5.3 Mapper early termination

The randomized and weighted randomized mappers require significantly longer run times than the deterministic heuristic due to their multiple iterations. Additionally, the runtime of these algorithms is hampered by another effect. For any given row, it is possible that the mapper will not be able to place all of the nodes. When this happens, the mapper will start

the row over, attempting to place the problem node(s) first, this is shown in Algorithm 2 lines 11–13. We call each occurrence of this behavior a *row placement failure*. The random oriented algorithms tend to have iterations that have a large volume of row placement failures, exacerbating run times.

To limit the runtime overheads from row placement failure, during the deterministic mapper run we record the number of row placement failures across all of the rows. The randomized versions of the mapper then use this n value as an upper limit on the number of total row placement failures permitted per iteration. Once this limit of n row placement failures is eclipsed, the mapper aborts and moves on to the next iteration.

5.4 Results

Table 4 compares the fabric size, path length increase, and mapping time for the deterministic, randomized, and weighted randomized mappers. The two iterative mappers were run for 500 iterations each. In order to further gauge the performance of the randomized and weighted randomized mappers, the average mapping time per iteration is also reported. The completely randomized mapper outperformed or equaled the deterministic mapper in terms of fabric size and total path length for all of the benchmarks. The weighted randomized mapper was able to find significantly better solutions than both of the other mappers. The weighted randomized mapper was also, on average, 48% faster than the randomized mapper.

		GSM	ADPCM Encoder	ADPCM Decoder	IDCT Column	IDCT Row	Sobel	Laplace
Deterministic Algorithm	Rows Added	1	5	1	8	4	1	2
	Path Length Increase	2	11	1	42	26	1	2
	Total Time (s)	< 1	12	< 1	1	< 1	< 1	< 1
Random Algorithm	Rows Added	0	5	0	6	3	0	0
	Path Length Increase	0	7	0	34	18	0	0
	Total Time (s)	978	5,109	1,378	3,006	1,169	51	75
	sec/iteration	2	10	3	6	2	< 1	< 1
Weighted Algorithm	Rows Added	0	0	0	3	1	0	0
	Path Length Increase	0	1	0	25	9	0	0
	Total Time (s)	366	4,019	394	920	549	36	55
	sec/iteration	< 1	8	< 1	2	1	< 1	< 1

Table 4. Deterministic, randomized, and weighted randomized mapper comparison using a 5:1 interconnect.

Table 5 illustrates the effects of implementing the early termination mechanism. Once again, the random and weighted randomized mappers were run for 500 iterations. The randomized mapper was still able to outperform the deterministic mapper in some cases, but not by the same margin as before. The reduction in execution time was roughly 38% for the randomized mapper. The weighted randomized mapper still performed well, only requiring one additional row to be added in IDCT Row. The weighted randomized mapper, which was already faster than the randomized mapper, saw a 12% improvement in terms of mapping time.

The randomized and weighted randomized algorithms were also tested using the 5:1 interconnect with 33% of the functional units replaced by dedicated pass-gates. These results are presented in Table 6. Again the random and weighted randomized mappers were run for 500 iterations with early termination enabled. As with the basic 5:1 interconnect, the randomized mapper performed as well or better than the deterministic mapper in terms of

		GSM	ADPCM Encoder	ADPCM Decoder	IDCT Column	IDCT Row	Sobel	Laplace
Deterministic Algorithm	Rows Added	1	5	1	8	4	1	2
	Path Length Increase	2	11	1	42	26	1	2
	Total Time (s)	< 1	12	< 1	1	< 1	< 1	< 1
Random Algorithm	Rows Added	0	4	0	8	4	0	0
	Path Length Increase	0	9	1	42	26	0	0
	Total Time (s)	561	4,262	338	1,148	781	40	63
	sec/iteration	1	9	< 1	2	2	< 1	< 1
Weighted Algorithm	Rows Added	0	0	0	3	2	0	0
	Path Length Increase	0	1	0	22	18	0	0
	Total Time (s)	327	3,803	216	731	517	36	56
	sec/iteration	< 1	8	< 1	1	1	< 1	< 1

Table 5. Deterministic, randomized, and weighted randomized mapper comparison using a 5:1 interconnect and early termination.

fabric size and path length increase. However, the weighted randomized mapper was superior to both of the other mappers. Since this interconnect contains dedicated pass-gates, the number of FUs utilized as pass-gates are also included in the results. A similar trend was observed where the weighted randomized mapper performed the best (fewest FUs used as pass-gates), followed by the randomized mapper and the deterministic mapper.

		GSM	ADPCM Encoder	ADPCM Decoder	IDCT Column	IDCT Row	Sobel	Laplace
Deterministic Algorithm	Rows Added	1	6	2	7	5	0	2
	Path Length Increase	2	13	3	37	33	0	2
	ALUs as Pass-gates	75	116	37	77	47	2	3
	Total Time (s)	< 1	7	< 1	2	1	< 1	< 1
Random Algorithm	Rows Added	1	6	2	7	3	0	1
	Path Length Increase	2	13	3	37	27	0	1
	ALUs as Pass-gates	75	116	37	77	39	2	5
	Total Time (s)	500	5,578	331	2,052	949	42	100
	sec/iteration	1	11	< 1	4	2	< 1	< 1
Weighted Algorithm	Rows Added	0	2	0	5	3	0	2
	Path Length Increase	0	4	1	44	23	0	2
	ALUs as Pass-gates	74	83	30	65	36	2	3
	Total Time (s)	367	3,989	233	1,397	618	42	92
	sec/iteration	< 1	8	< 1	3	1	< 1	< 1

Table 6. Deterministic, randomized, and weighted randomized mapper comparison using a 5:1 interconnect, 33% dedicated passgates, and early termination.

6. Sliding partial MILP heuristic

The sliding partial MILP heuristic is a greedy approach to solve the augmented fixed rows mapping problem. As discussed in Section 2.1.2, we created a MILP that solved the feasible mapping with fixed rows problem for the entire device in a single formulation but the run times were prohibitively long (hours to days). However, an MILP that solves a limited scope of the mapping problem can run much faster (seconds). Thus, the partial sliding MILP heuristic creates the middle ground between the greedy heuristics and the full MILP

formulation. It has similarities to the deterministic and randomized heuristics from Sections 4 and 5 in that it follows a top to bottom approach, and rows that have been visited cannot be adjusted. However, while these earlier heuristics place a single node at a time, the sliding approach uses an MILP for an optimal solution for an entire row or multiple rows in one step. Thus, the sliding heuristic, while still greedy, has a much larger scope than the earlier greedy algorithms. Pseudocode for the sliding partial MILP heuristic is shown in Algorithm 7.

Algorithm 7 Sliding Partial MILP Heuristic

```
1: while there are violated edges do
2:    Find a violated edge (highest row).
3:    Solve a partial MILP to fix the violation(s) or to push the violation(s) down.
4:    if the violation(s) is not fixed and cannot be pushed down then
5:       if the number of pass-gates < limit then
6:          Add a row of pass-gates.
7:       else
8:          Exit (heuristic cannot generate a mapping).
9:       end if
10:   end if
11: end while
```

The heuristic starts with an "arbitrary placement" where operations are placed in the earliest row possible (ASAP) and the operations are left justified. The heuristic follows a top-down approach and continues until it fixes all of the violations. We define violations as edges connecting FUs between rows that cannot be realized using the routing described in the FIM. When a violated edge is located, a window of rows is selected as shown in Figure 11. Within this window, an MILP is solved to attempt to correct the violations by moving the column locations of the nodes. We call this a partial MILP as it only solves the mapping problem for part of the fabric (i.e. the limited window). Because the heuristic takes a top-down approach, any previously existing violations above the MILP window should have already been corrected. However, it is possible that violations exist below the window. Selection of the window of rows is discussed in Section 6.2.

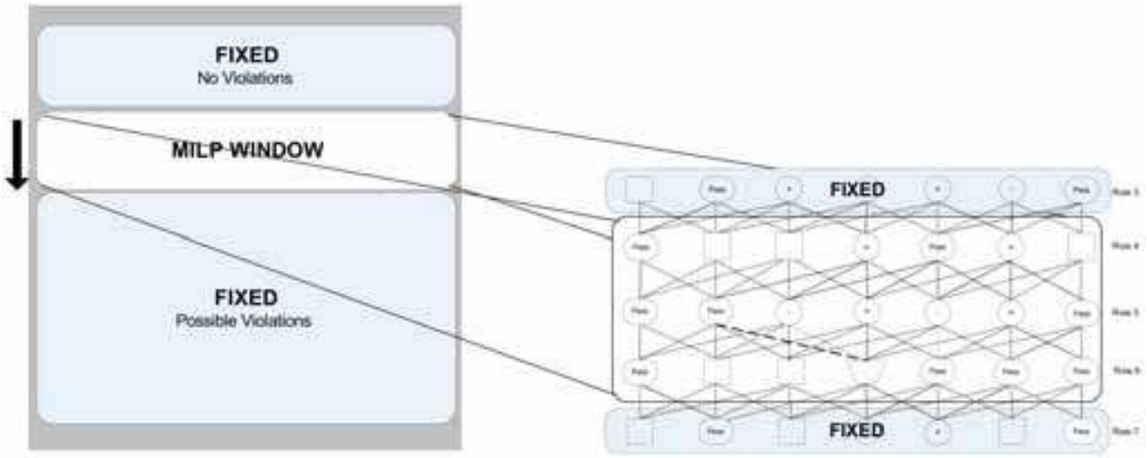
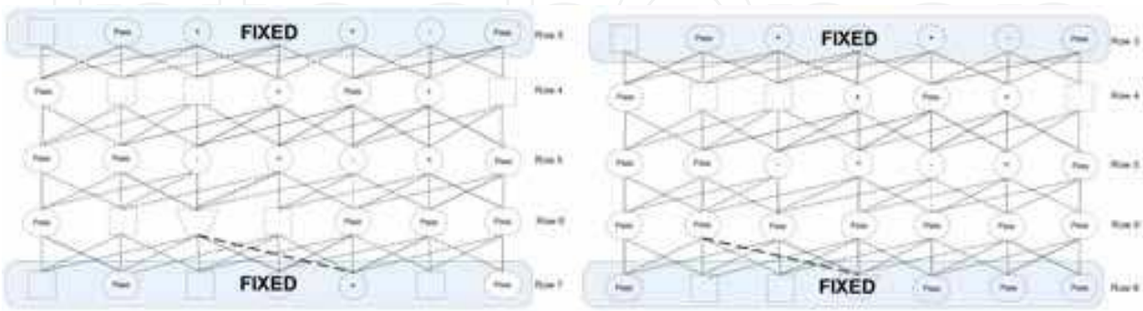


Fig. 11. Nearly feasible solution.

There are three possibilities at the end of the run on a particular window:
Case 1: The partial MILP fixes the violation(s).

Case 2: The partial MILP cannot fix the violation(s), but pushes the violation(s) down in the window so that subsequent windows may be able to find a solution. This case is shown in Figure 12(a) where the violating edge, represented by the bold dashed line between rows 5 and 6 in Figure 11, has been pushed down between rows 6 and 7.

Case 3: If the partial MILP cannot fix or push down the violation then a row of pass-gates is added to increase the flexibility, and the partial MILP is run again. This is illustrated with the addition of row 5' containing all pass-gates in Figure 12(b).



(a) Violation from Figure 11 is pushed down.

(b) A row of pass-gates is added to the window from Figure 11.

Fig. 12. Resolutions when the partial MILP cannot eliminate a violation.

6.1 Partial MILP formulation

The heuristic generates valid mappings by using small, fast MILPs on the selected rows (Figure 11). The partial MILP formulation, parameters, sets, and variables used in the heuristic are described below:

The objective function (0) minimizes the number of edges used that violate the interconnect design. Constraint (1) ensures that an edge can be located in only one place. Constraint (2) ensures that an operator can be placed in only one column. Constraint (3) states that there can be at most one operator in a column for a given row k. The final two constraints relate the operator, x, and edge, z, variables. Constraint (4) states that edge (i, t) can only be placed starting at column j if operator i is at column j. Constraint (5) states that edge (i, t) can only be placed to end at column k if the ending operator t is at column k. We fix the locations of the operators in the first and last row of the MILP window by setting $x_{ij} = 1$ where i is the operator and j is the column.

Parameters:

r: Number of rows

c: Number of columns

$p_{i,t,j,k}$: Objective coefficients

a_i: Index of the first operator in row i

Sets:

C: Set of columns (1 to c)

V: Set of operators in the MILP window

E: Set of edges in the MILP window

R: Set of rows in the MILP window (1 to r)

Variables:

x_{ij}: Binary variable for operator assignment. If operator i is in column j, then $x_{ij} = 1$, otherwise it is 0.

$z_{i,t,\ j,k}$: Binary variable for edge assignment. If starting operator i is in column j and ending operator t is in column k, then $z_{i,t,j,k} = 1$, otherwise it is 0.

Partial MILP formulation:

$$\min \quad \sum_{(i,t)\in E}\sum_{j\in C}\sum_{k\in C} p_{i,t,j,k} z_{i,t,j,k} \tag{0}$$

$$s.t. \quad \sum_{j\in C}\sum_{k\in C} z_{i,t,j,k} = 1 \quad \forall (i,t) \in E \tag{1}$$

$$\sum_{j\in C} x_{ij} = 1 \quad \forall i \in V \tag{2}$$

$$\sum_{i\in V, a_k \le i < a_{k+1}} x_{ij} \le 1 \quad \forall k \in R, j \in C \tag{3}$$

$$\sum_{k\in C} z_{i,t,j,k} \le x_{ij} \quad \forall (i,t) \in E, j \in C \tag{4}$$

$$\sum_{j\in C} z_{i,t,j,k} \le x_{tk} \quad \forall (i,t) \in E, k \in C \tag{5}$$

$$x_{ij} \ge 0 \quad \forall i \in V, j \in C \tag{6}$$

$$z_{i,t,j,k} \ge 0 \quad \forall (i,t) \in E, j \in C, k \in C \tag{7}$$

The effect of "pushing a violation down" when solving the MILP can be achieved with proper objective coefficients ($p_{i,t,\ j,k}$) in the formulation. The objective coefficients for the violations in the upper rows are much higher than the ones in the lower rows. For example, assume there is a violation between rows 5 and 6, and that the MILP window size is three rows as shown in Figure 11. In other words, the columns of the operators in rows 4, 5, and 6 can be adjusted while the locations of the operators in rows 3 and 7 are fixed. The objective coefficients are 10,000 for edges between rows 3 and 4 and between rows 4 and 5, 100 for the edges between rows 5 and 6, and 1 for the edges between rows 6 and 7. This avoids having violations in the higher rows, and may push the violation down to the rows between 6 and 7. Thus, objective values > 10,000 show an improvement. Of course, if a objective value of 0 is reached, then all violations are eliminated.

6.2 Determining window size

In the sliding partial MILP heuristic, a window of rows is selected for optimization. We tested different alternatives of numbers of rows to optimize in this window and used a cardinality five interconnect as the target for the tests. The more rows that are optimized simultaneously, the longer the MILP takes. However, it may not be possible to solve the violations if too few rows are included in the optimization window. We consider window sizes from one to five rows, as well as some approaches that change the number of rows. We did not exceed five rows since it started to take too long to solve the MILP.

Optimizing a single row is a special case. Since all of the variables are binary and the locations of the operators in other rows are fixed, this formulation can be solved directly as a Linear Program (LP), which can be solved much more efficiently than an MILP. Since a violated edge connects two rows, either of these rows can be targeted for optimization. Thus, we attempt to optimize the top row first, and if unsuccessful, attempt to optimize the bottom row. Unfortunately, the single row method was not capable of solving the majority of the benchmarks. Using such a small window size often resulted in the LP being unable to find a feasible solution for a given row. In these cases, rows of pass-gates would be continuously added in the same location and the algorithm was unable to make progress.

When optimizing two rows, the window would contain the two rows connected by the violating edge. Additionally, we attempted to correct the violations by optimizing previous rows. For example, if there is a violation between rows 5 and 6, first rows 4 and 5 are optimized. If the violation is not fixed, rows 5 and 6 are optimized. This example is shown in Figure 13. Unfortunately, most of the benchmarks also could not be solved by using a window size of two.

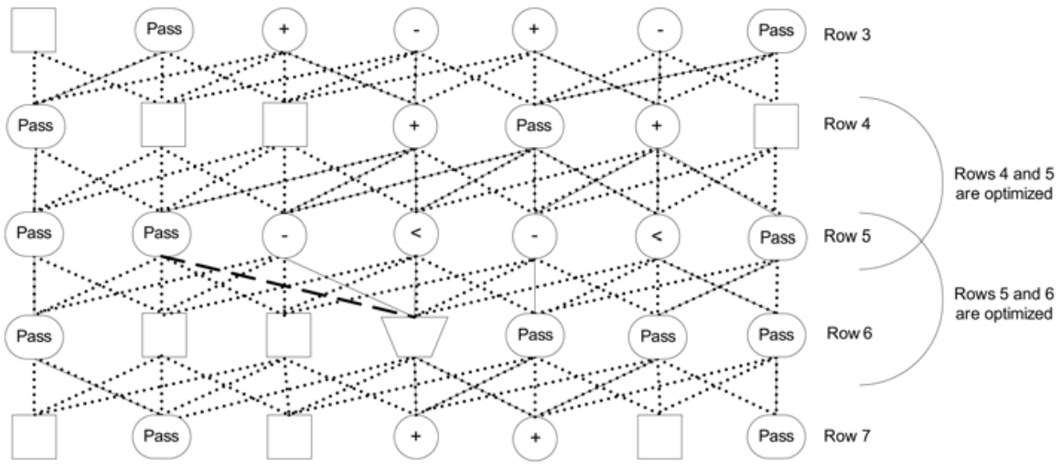


Fig. 13. Optimization of two rows.

Next, a window size of three was tested as shown in Figure 11. In this example, there is a violation between rows 5 and 6. The MILP fixes the operators in rows 3 and 7, while attempting to optimize the operators in rows 4, 5, and 6. This approach was able to solve five of the seven benchmarks. The other two benchmarks could not be solved because they each have an operator with grandparents that are prohibitively far from each other. This case could not be solved by our algorithm even by adding rows of pass-gates.

Having grandparents or great-grandparents far from each other causes problems. To solve this problem we introduced graded objective coefficients. The farther the violation is, the more it will cost. Based on this idea, the objective coefficients are multiplied by the absolute distance difference between the column numbers of the edges' operators. Thus, operators with the same grandchildren are more likely to placed close to each other. Even if the grandparents or great-grandparents are far from each other, violations can be fixed with the graded objective function by adding enough rows of pass-gates. After adding this feature, "graded optimize three" can solve all of the instances. However, it adds seven rows of pass-gates for one of the benchmarks.

Optimizing three rows is successful because rows of pass-gates are utilized efficiently. When a row of pass-gates is added, the locations of the operators in the preceding and succeeding rows can be adjusted. In contrast, when two rows are optimized, only the locations of the row of pass-gates and another row can be adjusted, which does not always help. In other words, sometimes the violation cannot be fixed and rows of pass-gates are constantly added between the same rows.

The four row optimization approach performs well. All of the instances were solved by adding at most five rows of pass-gates. Using the increased window size allows rows of pass-gates to be utilized more efficiently than in the three row optimization scenario. Finally, optimizing five rows can solve all of the benchmarks by adding two rows at most. However, the solution times are significantly longer than those of the "optimize four rows" version.

Based on the tests "graded optimize four" is chosen as the best option. Optimizing a window of four rows does not take long. Additionally, it does not add too many rows of pass-gates since the rows of pass-gates are utilized efficiently.

6.2.1 Dedicated pass-gates

Based on the "graded optimize four" window size, the same window size was applied to interconnects with dedicated pass-gates. While the heuristic was successful with more relaxed interconnects (e.g. 8:1 cardinality with various percentages of dedicated pass-gates) the results for the more restrictive 5:1 cardinality interconnect with dedicated pass-gates led to several infeasible solutions. This is a limitation of the algorithm operating on the entire row and not being able to move individual operations into different rows like the deterministic and randomized heuristics.

6.3 Extensions

To improve the quality of the partial MILP heuristic, we explored some logical extensions. In the next two subsections, respectively, we describe an iterative method to improve the runtime and retain similar quality of results and a premapping step to potentially improve the quality of results.

6.3.1 Iterative approaches

Solving partial MILPs with smaller window sizes is much faster but is less effective at removing violations than larger window sizes. Thus, in the iterative approach we use variable sized windows starting with small window sizes and escalating to larger window sizes if necessary. Thus, the window size is increased if the violation(s) cannot be fixed or pushed down with the current size. For instance, in "iterative 1234" first one row is optimized. If the violation(s) cannot be removed, the window size is increased to two rows and the MILP is run again. This continues through a window size of four rows. If the MILP cannot be solved for four rows, a row of pass-gates is added. These iterative approaches perform well and are competitive with the "optimize four rows" version.

6.3.2 Two-pass sliding partial MILP heuristic

We discovered that the sliding partial MILP heuristic can be more effective if it starts with a nearly feasible solution when compared with an arbitrary solution. Thus, we created a two-pass extension of the sliding partial MILP heuristic. The one-pass heuristic sometimes requires adding a row of pass-gates to fix violations. Thus, in the first pass of the two-pass heuristic, the option to add a row of pass-gates is removed and this pass runs partial MILPs to minimize the number of violated edges. However, some violations may remain. We used this pass on the arbitrary solutions to create better starting points for the sliding partial MILP heuristic (i.e. the second pass). We tested this heuristic approach for one, two, and three row windows, respectively for the first pass and "graded optimize four rows" in the second pass.

6.4 Results

This section presents the results of tests on the sliding partial MILP heuristic for 5:1 cardinality interconnects for both the one-pass and two-pass instantiation. Table 7 summarizes the number of rows added and the run times for the heuristics "optimize one row," "optimize two rows," "optimize three rows," "graded optimize three rows," "optimize four rows," "graded optimize four rows," and "graded optimize five rows" starting from an arbitrary solution. The "optimize one row" and "optimize two rows" methods only solve Sobel and Laplace, the two smallest benchmarks. The other benchmarks cannot be solved even after adding 20 rows of pass-gates. The "optimize three rows" method solves five out of seven benchmarks. The "graded optimize three rows" approach

solves all of the instances. The longest run time for "graded optimize three rows" is 102 seconds and it adds at most seven rows. The "graded optimize four rows" solutions are always as good as the "optimize four rows" solutions in terms of fabric size. For ADPCM Encoder and both of the IDCT benchmarks, "graded optimize four rows" adds fewer rows than the "optimize four rows" approach. There are not significant differences in the run times. Even though arbitrary solutions are used as the starting points, the run times are not significantly long. IDCT Column has the longest runtime of approximately ten minutes. On the other hand, "graded optimize five rows" adds fewer rows than the other heuristics but at the price of much longer run times. It adds at most two rows of pass-gates, however, the run times are long enough to not be a practical option in many cases.

		GSM	ADPCM Encoder	ADPCM Decoder	IDCT Column	IDCT Row	Sobel	Laplace
Optimize	Rows Added	> 20	> 20	> 20	> 20	> 20	4	3
One Row	Time (s)	-	-	-	-	-	< 1	< 1
Optimize	Rows Added	> 20	> 20	> 20	> 20	> 20	3	3
Two Rows	Time (s)	-	-	-	-	-	4	3
Optimize	Rows Added	2	> 20	3	6	9	0	> 20
Three Rows	Time (s)	24	-	37	88	88	4	-
Graded Optimize	Rows Added	1	3	2	7	2	0	1
Three Rows	Time (s)	31	40	31	102	22	5	95
Optimize	Rows Added	0	3	0	6	2	0	0
Four Rows	Time (s)	43	152	150	674	343	3	12
Graded Optimize	Rows Added	0	2	0	5	1	0	0
Four Rows	Time (s)	80	143	32	635	373	4	87
Graded Optimize	Rows Added	0	1	0	2	1	0	0
Five Rows	Time (s)	258	318	34	1,598	5,721	15	159

Table 7. Tests on arbitrary instances.

The results for the iterative "graded 1234," "graded 234," and "graded 34" heuristics are shown in Table 8. The "iterative graded 1234" and "iterative graded 234" heuristics behave similarly since optimizing one row rarely eliminates violations. They add at most six rows. When comparing the "graded optimize four rows" and "iterative graded 34" heuristics, "graded optimize four" is better for GSM while "iterative graded 34" is better for IDCT Column in terms of the number of rows added.

Based on all of the computational tests, graded objective coefficients helped to find better mappings in terms of rows added and mapping time. The "graded optimize four rows" and "iterative graded 34" approaches were found to be the best of this group. Thus, to examine the heuristic extensions we will retain the "graded optimize four rows" method for the remaining tests.

		GSM	ADPCM Encoder	ADPCM Decoder	IDCT Column	IDCT Row	Sobel	Laplace
Iterative	Rows Added	1	3	0	6	3	0	2
Graded 1234	Time (s)	18	109	37	209	401	3	12
Iterative	Rows Added	1	3	0	6	3	0	2
Graded 234	Time (s)	17	103	37	195	398	2	12
Iterative	Rows Added	1	2	0	3	1	0	0
Graded 34	Time (s)	40	93	43	242	48	7	90

Table 8. Tests of iterative versions on arbitrary instances.

In the two-stage sliding partial MILP heuristic, a nearly feasible solution could be found in the first stage using either "heuristic one row," "heuristic two rows," or "heuristic three rows." Table 9 summarizes the number of rows added and the run times of the two-stage heuristic, with the run times separated into time for the first stage and second stage. The run times of the first stage are less than one second for "heuristic one row," at most ten seconds for "heuristic two rows," and less than two minutes for "heuristic three rows." Starting the sliding partial MILP heuristic from solutions found using "heuristic one row" is not much better than starting from an arbitrary solution. However, "heuristic two rows" and "heuristic three rows" solutions provide more benefit in the first stage. The sliding partial MILP heuristic starting from "heuristic two rows" or "heuristic three rows" adds fewer rows with shorter solution times than starting from an arbitrary solution.

		GSM	ADPCM Encoder	ADPCM Decoder	IDCT Column	IDCT Row	Sobel	Laplace
Heuristic	Rows Added	0	2	0	3	1	0	0
One Row	Time (s)	$< 1 + 4$	$< 1 + 78$	$< 1 + 48$	$< 1 + 195$	$< 1 + 112$	$< 1 + 1$	$< 1 + 13$
Heuristic	Rows Added	1	1	1	2	1	0	0
Two Rows	Time (s)	$7 + 4$	$10 + 104$	$4 + 108$	$9 + 140$	$5 + 151$	$1 + 5$	$1 + 36$
Heuristic	Rows Added	0	0	0	1	1	0	0
Three Rows	Time (s)	$36 + 3$	$42 + 81$	$51 + 39$	$106 + 90$	$42 + 25$	$6 + < 1$	$10 + 1$

Table 9. Tests on optimized instances.

When the sliding partial MILP heuristic starts from an arbitrary solution, it adds more rows and solution times are 0-11 minutes. The two-stage version adds fewer rows and total run times are less than four minutes. So, the best option was found to be running the "heuristic three rows" to generate a starting point and then using the "graded optimize four rows" sliding partial MILP heuristic to generate a valid mapping.

Table 10 shows that 8 out of 21 instances cannot be solved by the sliding partial MILP heuristic for cardinality five interconnect with dedicated pass-gates (for 25%, 33%, and 50%). However, by adding rows of pass-gates the heuristic can solve four of the instances—ADPCM Encoder and Laplace for 33% dedicated pass-gates and ADPCM Decoder and Laplace for 50% dedicated pass-gates—that are proven infeasible for the feasible mapping with fixed rows solution (Baz, 2008). The heuristic adds at most two rows of pass-gates for these solutions and they are shown in bold in Table 10. When we consider the instances solved by the heuristic, the longest run time is 2,130.5 seconds. The two-stage heuristic did not find mappings not found by the one-stage heuristic. This is due to dense structures in these graphs that cannot be separated without moving individual nodes across rows (see Section 7).

		GSM	ADPCM Encoder	ADPCM Decoder	IDCT Column	IDCT Row	Sobel	Laplace
25% Dedicated	Rows Added	0	0	0	-	-	0	0
Pass-gates	Time (s)	< 1	19	5	-	-	< 1	< 1
33% Dedicated	Rows Added	0	**2**	0	-	-	0	**1**
Pass-gates	Time (s)	60	322	42	-	-	7	203
50% Dedicated	Rows Added	1	-	**2**	-	-	-	**2**
Pass-gates	Time (s)	40	-	2,130	-	-	-	116

Table 10. Tests on 5:1 interconnect with 25%, 33%, and 50% dedicated pass-gates.

7. Conclusion

In this chapter we have presented three greedy heuristics for mapping applications onto a reconfigurable device oriented for low-energy execution. The three heuristics are a deterministic top-down greedy algorithm described in Section 4, a greedy algorithm with randomization discussed in Section 5 based on the deterministic algorithm flow, and a partial MILP greedy heuristic presented in Section 6.

Here we compare the deterministic, randomized, weighted randomized, and sliding partial MILP heuristics described in Section 4, Section 5.1, Section 5.2, and Section 6, respectively. The comparisons are made using a 5:1-based interconnect and are shown in Table 11. The results compare the different heuristics in terms of fabric size, path length increase, and mapping time.

		GSM	ADPCM Encoder	ADPCM Decoder	IDCT Column	IDCT Row	Sobel	Laplace
Deterministic	Rows Added	1	5	1	8	4	1	2
Algorithm	Path Length Increase	2	11	1	42	26	1	2
	Time (s)	< 1	12	< 1	1	< 1	< 1	< 1
	Rows Added	0	4	0	8	4	0	0
Random	Path Length Increase	0	9	1	42	26	0	0
Algorithm	Total Time (s)	561	4,262	338	1,148	781	40	63
	sec/iteration	1	9	< 1	2	2	< 1	< 1
	Rows Added	0	0	0	3	2	0	0
Weighted	Path Length Increase	0	1	0	22	18	0	0
Algorithm	Total Time (s)	327	3,803	216	731	517	36	56
	sec/iteration	< 1	8	< 1	1	1	< 1	< 1
Sliding	Rows Added	0	3	0	6	2	0	0
MILP Heuristic	Path Length Increase	0	6	0	40	8	0	0
	Time (s)	79	142	32	635	373	3	87
Two-stage	Rows Added	0	0	0	2	1	0	0
Sliding	Path Length Increase	0	0	0	8	8	0	0
MILP Heuristic	Time (s)	37	123	89	196	67	6	11

Table 11. Comparison of greedy mapping techniques targeting a 5:1 cardinality interconnect.

Each heuristic provides different advantages and disadvantages. For example the deterministic approach provides a solution quickly but not of the highest quality as measured by required fabric size and total path length. The partial MILP heuristic was able to out perform the deterministic approach due to its much larger window size considering entire rows of nodes versus a single node, respectively. Actually, the weighted randomized algorithm provides better qualities of solution than the partial MILP heuristic but the run times are much higher. The two-stage partial MILP heuristic performs the best overall with reasonable run times (actually better than the one-stage partial MILP heuristic in many cases). Thus, if generating mappings in seconds is essential, the deterministic heuristic can be used. If energy consumption is critical and run times in minutes are acceptable, the two-stage sliding partial MILP heuristic should be used.

However, the large multi-row window size for the MILP heuristic became a disadvantage for restrictive interconnects with dedicated pass-gates, for which the randomized greedy heuristic provides the best results and the partial MILP heuristic is not able to solve many cases.

To better understand the sliding partial MILP heuristic performance for this interconnect, we analyzed the eight instances which cannot be solved by the heuristic. The benchmarks IDCT Row and IDCT Column are infeasible because they have nodes which have four commutative

children. In a situation like this, dedicated pass-gates with a cardinality five interconnect is too restrictive. In fact for a cardinality five interconnect with 50% dedicated pass-gates, the partial MILP was unable to map a majority of the benchmarks including one of the smallest ones (Sobel). To be able to solve these cases the operator assignments must be revised such that each node can have at most three children. This requires a pre-processing step to use the sliding partial MILP heuristic that will enforce these input and output restrictions, which will increase the overall path length and possibly the number of rows in the solution.

7.1 Future work

From the exploration of the heuristics described in this chapter there are clear tradeoffs between the three main heuristics. The deterministic approach is fast but far from optimal. The partial MILP heuristic (particularly the two-stage version) is strong for cardinality five and requires a reasonable time (seconds to minutes) to map but has problems when introducing dedicated pass-gates. The weighted randomized heuristic performed reasonably well for solution quality and could map the dedicated pass-gate interconnect, but the run times were too long.

In our future work we plan to investigate methods to improve the runtime of the weighted randomized heuristic. For example, currently the heuristic re-evaluates the weights after the placement of each node. To make the decision faster, the heuristic could select multiple nodes to place based on the current weights before recalculating. Additionally, the early termination can be revised to avoid losing good solutions but also to create more candidates for early termination to improve performance. One example might be to relax the number of row placement failures but to terminate if the path length increase exceeds the current best solution (currently we use solution size).

To improve the performance of the partial MILP heuristic, we can develop a pre-processing pass that relaxes infeasible constructs so that they can be mapped. We also may consider expanding the MILP to allow nodes to move between rows as well as columns. However, this is expected to significantly increase the runtime of the partial MILPs and may require use of a smaller window size. Additionally, we plan to explore other "first-stage" passes for the two-stage heuristic. We may explore using the full fabric MILP to generate a nearly feasible fixed rows mapping or investigate other approaches such as simulated annealing for this first stage.

8. References

Baz, M. 2008. Optimization of mapping onto a flexible low-power electronic fabric architecture. Ph.D. thesis, University of Pittsburgh, Pittsburgh, Pennsylvania.

Baz, M., Hunsaker, B., Mehta, G., Stander, J., and Jones, A. K. 2008. Application mapping onto a coarsegrained computational device. *European Journal of Operations Research.* in submission and revision since April 2007.

Bray, T., Paoli, J., C. M. Sperberg-McQueen, E. M., and Yergeau, F. 2006. Extensible markup language (xml) 1.0 (fourth edition) - origin and goals. Tech. Rep. 20060816, World Wide Web Consortium.

Cordella, L. P., Foggia, P., Sansone, C., and Vento, M. 2004. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence 26,* 10 (October), 1367– 1372.

Diestel, R. 2005. *Graph Theory*. Springer-Verlag: Heidelberg. 3rd edition.

Garey, M. and Johnson, D. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W. H. Freeman.

Hauser, J. R. and Wawrzynek, J. 1997. Garp: A MIPS processor with a reconfigurable coprocessor. In *IEEE Symposium on FPGAs for Custom Computing Machines*, K. L. Pocek and J. Arnold, Eds. IEEE Computer Society Press, Los Alamitos, CA, 12–21.

Hoare, R., Jones, A. K., Kusic, D., Fazekas, J., Foster, J., Tung, S., and McCloud, M. 2006. Rapid VLIW processor customization for signal processing applications using combinational hardware functions. *EURASIP Journal on Applied Signal Processing 2006*, Article ID 46472, 23 pages.

Johnson, T., Robertson, N., Seymour, P. D., and Thomas, R. 2001. Directed tree-width. *Journal of Combinatorial Theory. Series B 82,* 1, 138–154.

Jones, A. K., Hoare, R., Kusic, D., Fazekas, J., and Foster, J. 2005. An FPGA-based VLIW processor with custom hardware execution. In *ACM International Symposium on Field-Programmable Gate Arrays (FPGA)*.

Jones, A. K., Hoare, R., Kusic, D., Mehta, G., Fazekas, J., and Foster, J. 2006. Reducing power while increasing performance with supercisc. *ACM Transactions on Embedded Computing Systems (TECS) 5,* 3 (August), 1–29.

Jones, A. K., Hoare, R. R., Kusic, D., Fazekas, J., Mehta, G., and Foster, J. 2006. A vliw processor with hardware functions: Increasing performance while reducing power. *IEEE Transactions on Circuits and Systems II 53,* 11 (November), 1250–1254.

Jones, A. K., Mehta, G., Stander, J., Baz, M., and Hunsaker, B. 2008. Interconnect customization for a hardware fabric. *ACM Transactions on Design Automation for Electronic Systems (TODAES)*. in press.

Krissinel, E. B. and Henrick, K. 2004. Common subgraph isomorphism detection by backtracking search. *Software—Practice and Experience 34*, 591–607.

Levine, B. and Schmit, H. 2002. Piperench: Power and performance evaluation of a programmable pipelined datapath. In *Presented at Hot Chips 14*.

MathStar. Field programmable object array architecture. http:/ / www.mathstar.com/ literature.html.

Mehta, G., Hoare, R. R., Stander, J., and Jones, A. K. 2006. Design space exploration for low-power reconfigurable fabrics. In *Proc. of the Reconfigurable Architectures Workshop (RAW)*.

Mehta, G., Ihrig, C. J., and Jones, A. K. 2008. Reducing energy by exploring heterogeneity in a coarse-grain fabric. In *Proc. of the IPDPS Reconfigurable Architecture Workshop (RAW)*.

Mehta, G., Stander, J., Baz, M., Hunsaker, B., and Jones, A. K. 2007. Interconnect customization for a coarse-grained reconfigurable fabric. In *Proc. of the IPDPS Reconfigurable Architecture Workshop (RAW)*.

Mehta, G., Stander, J., Lucas, J., Hoare, R. R., Hunsaker, B., and Jones, A. K. 2006. A low-energy reconfigurable fabric for the SuperCISC architecture. *Journal of Low Power Electronics 2,* 2 (August).

Messmer, B. T. and Bunke, H. 2000. Efficient subgraph isomorphism detection: A decomposition approach. *IEEE Transactions on Knowledge and Data Engineering 12,* 2, 307–323.

Mirsky, E. and Dehon, A. 1996. Matrix: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*.

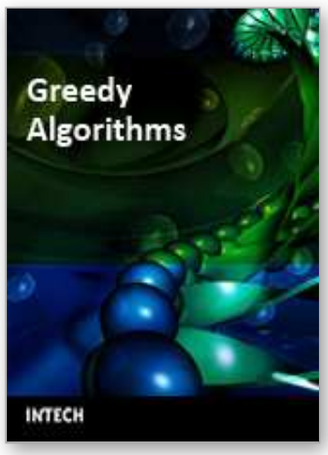
Resende, M. and de Sousa, J. 2004. *Metaheuristics: Computer Decision-Making*. Kluwer Academic Publishers.

Resende, M. and Ribeiro, C. 2008a. *Handbook of Metaheuristics, 2nd Edition*. Springer Publishers.

Resende, M. and Ribeiro, C. 2008b. *Search Methodologies, 2nd Edition*. Springer Publishers.

Sheng, L., Kaviani, A. S., and Bathala, K. 2002. Dynamic power consumption in virtex-II FPGA family. In *FPGA*.

Ullmann, J. R. 1976. An algorithm for subgraph isomorphism. *J ACM 23,* 1, 31 42.



Greedy Algorithms

Edited by Witold Bednorz

ISBN 978-953-7619-27-5
Hard cover, 586 pages
Publisher InTech
Published online 01, November, 2008
Published in print edition November, 2008

Each chapter comprises a separate study on some optimization problem giving both an introductory look into the theory the problem comes from and some new developments invented by author(s). Usually some elementary knowledge is assumed, yet all the required facts are quoted mostly in examples, remarks or theorems.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Colin J. Ihrig, Mustafa Baz, Justin Stander, Raymond R. Hoare, Bryan A. Norman, Oleg Prokopyev, Brady Hunsaker and Alex K. Jones (2008). Greedy Algorithms for Mapping onto a Coarse-grained Reconfigurable Fabric, Greedy Algorithms, Witold Bednorz (Ed.), ISBN: 978-953-7619-27-5, InTech, Available from: http://www.intechopen.com/books/greedy_algorithms/greedy_algorithms_for_mapping_onto_a_coarse-grained_reconfigurable_fabric

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen