Reducing Energy by Exploring Heterogeneity in a Coarse-grain Fabric *

Gayatri Mehta, Colin J. Ihrig, and Alex K. Jones
University of Pittsburgh
{gmehta,cihrig,akjones}@ece.pitt.edu

Abstract

This paper explores the impact of heterogeneity on energy consumption in a stripe-based coarse-grain fabric architecture. We examine the benefit of replacing 25-50% of functional blocks with dedicated vertical routes in the fabric. Additionally, we reduce the number of operations supported by the functional units from 23 to 16, 10 and 8. To assist in testing and examining the impact of these different architectures on energy consumption, an automation process was created to automatically generate fabric instances based on a Fabric Instance Model (FIM) written in XML. The FIM is also used as an input parameter to our heuristic mapper in order to program a particular fabric instance. Upon testing these instances, we found that the fabric with ALUs supporting 10-operations and using an 8:1 interconnect with 33% of the functional units replaced with dedicated pass gates provided the best energy versus mappability tradeoff, resulting in a 32% energy improvement and a 47% area savings over the baseline fabric with ALUs supporting 23-operations and using an 8:1 interconnect without dedicated vertical routes.

1 Introduction

Reconfigurable devices mitigate many of the problems encountered with the development of Application Specific Integrated Circuits (ASICs) for hardware acceleration. For example, reconfigurable devices amortize the rapidly increasing mask and non-recurring engineering (NRE) costs over many more generic devices. Computer Aided Design (CAD) flows are often simplified for these devices. The design cycle is also reduced because fabrication is not involved allowing the design to remain flexible, which can significantly decrease the time to market.

The tradeoff for using these reconfigurable devices is a compromise in performance and most notably power/energy consumption. To reduce the energy consumption of a reconfigurable device, particular care must be given to designing both functional units and interconnect of the device.

Stripe-based coarse-grained fabrics mimic the computational style of Data Flow Graphs (DFGs). The data flows from top to bottom in a stripe-based fabric and the DFG can be easily implemented on a stripe-based fabric architecture.

In our previous work, we explored the architectural space of a stripe-based coarse-grained fabric by studying the impact of varying different design parameters such as the data width of the functional units [14], and the granularity of the interconnect for both homogeneous and heterogeneous strategies [13] on power and performance.

In this paper, we explore heterogeneity in the functional unit design by introducing non-uniform arithmetic and logic units (ALUs) that support various operations in a stripe-based coarse-grain reconfigurable fabric. This allows the reduction of operations from 23 to between 8 and 16 and is designed to reduce energy consumption in the device. We also examine the benefit of adding dedicated vertical routes into the architecture called dedicated pass gates to prevent functional units from being used as routing. We examined the replacement of 25%, 33%, and 50% of the functional units with dedicated pass gates. While our technique applies to stripe-based reconfigurable fabrics in general such as PipeRench [11, 17] and Kilocore [16], our technique is demonstrated using the SuperCISC low-energy reconfigurable fabric target [14].

To assist in conducting our experiments, we developed a fabric instance model (FIM) written in XML used in an automation flow capable of generating various fabric instances with different parameters as well as configuring the mapper used to program the fabric with various applications as shown in Figure 1. Upon testing these instances, we show that a fabric with ALUs supporting 10-operations and an 8:1 interconnect with 33% dedicated pass gates provides the best energy versus routability solution.

---

*This work was partially supported by The Technology Collaborative.

Figure 1. Architecture evaluation tool flow.
The remainder of this paper is organized as follows: Section 2 provides some background material in the area of reconfigurable computing and coarse-grain architectures in general. An overview of the fabric target used in this paper to demonstrate the impact of the interconnect is presented in Section 3. The automated fabric model generation and testing using an interconnect model specification is described in Section 4. The heuristic mapping algorithm used to program the reconfigurable device is introduced in Section 5. Section 6 includes results and an analysis of energy consumption for a suite of benchmark circuits. Section 7 discusses conclusions and considers future work.

2 Background and Literature Review

A tremendous amount of effort has been devoted to the area of reconfigurable computing for application acceleration with custom hardware. While FPGAs are the most commonly used general purpose reconfigurable devices, they exhibit relatively poor power characteristics.

Recently, the development and use of coarse-grained fabrics for computationally complex tasks has received a lot of attention as a possible alternative to FPGAs. Many architectures have been proposed and developed both in academia and industry during the last decade such as MATRIX [15], Garp [8], RaPiD [6], PipeRench [11], HFPGA [1], Kilocore [16], and the FPOA [12].

Several methods have been proposed in the past few years for design space exploration of reconfigurable architectures [2, 3, 7]. However, these methods are either technology-dependent or architecture-dependent and do not consider stripe-based fabrics in particular. They deal with a low-level of abstraction and provide only limited design space exploration around their target architecture. Bossuet et al. [4] proposed a design space exploration method that can be used to cover a wide domain of reconfigurable fabrics, from fine-grained to coarse-grained fabrics, as well as heterogeneous fabrics. They used the architectural processing use rate and the communication hierarchical distribution as metrics to investigate a power-efficient architecture.

In our previous work, we studied the impact of various parameters of the fabric such as the data width of the functional units and the interconnect granularity onto power/performance. We power-profiled different architectural techniques for implementing functional units [14]. We studied several system-level interconnection strategies including homogeneous and heterogeneous interconnects for a coarse-grain reconfigurable fabric in our previous work. We studied different interconnects between stripes ranging from fully connected to cardinalities of 4:1 including hybrids between 5:1 and 3:1 cardinalities [13].

In this paper, we study the effect providing dedicated vertical routing in the device coupled with the heterogeneity of operations in the ALUs on the energy consumed in the device. We also describe an automated flow to generate various fabric instances and automatically configure the mapper to program these instances.

3 SuperCISC Fabric System Overview

The SuperCISC fabric is a stripe-based fabric designed to operate within the SuperCISC processor architecture summarized in [10]. The idea is to accelerate the 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. These kernels are converted into entirely combinational hardware functions generated automatically from C using a design automation flow [9].

The hardware functions are represented as as Super Data Flow Graphs (SDFGs). SDFGs are extension to traditional Data Flow Graph that allows certain kinds of control to be implemented as predicated datapaths [9].

3.1 Reconfigurable Fabric Target

Stripe-based hardware fabrics are designed to easily map data flow graphs (DFGs) from the application onto the device. The SuperCISC fabric works in a similar way retaining a data flow structure allowing computational results to be computed in one ALU and flow into others in the system. As shown in Figure 2, ALUs are organized into rows or computational stripes within which each functional unit operates independently. The results of these ALU operations are then fed into interconnection stripes constructed using multiplexers.

A detailed diagram of the interconnection stripe is shown in Figure 3. In addition to determining how many wires are available for routing, the cardinality of the multiplexer.
determines the maximum fanin and fanout possible in the DFG to map onto the structure.

One of the purposes of using a coarse grain fabric is that the architecture removes much of the routing flexibility of a more general device such as an FPGA. While this works in a stripe-based configuration to mimic the dataflow computation style and reduce energy, in doing so, we have created a lot of wasted logic by replicating fully-functional ALUs with many functions that will not be used. Additionally, some types of operations tend to occur more frequently than others. Thus, the ALUs can be tuned to the needs of the class of applications. By allowing each ALU to be different, the number of operations supported per ALU can be significantly reduced, while not dramatically increasing the difficulty of mapping applications onto the fabric.

In the next sections we explore several heterogeneous fabric configurations. We begin with a very simple extension from a homogeneous fabric, the inclusion of specialized vertical routes or dedicated pass gates. Then we explore a much more general heterogeneous ALU concept and discuss how operations are selected for each ALU in the fabric.

3.2 Dedicated Pass Gates

When mapping a DFG to a stripe-style structure, data dependency edges often traverse multiple rows. In these fabrics, ALUs must often pass these values through without doing any computation. We call these operations in the graph, pass gates. Figure 4 provides a comparison of ALUs used in the graph, showing that more than 50% of the ALUs in the fabric will be used for routing by configuring the ALU as a pass gate. Thus, some percentage of the ALUs in the fabric can be replaced with dedicated pass gates to reduce complexity of the device.

3.3 ALU Heterogeneity

Another approach to reducing the complexity of the ALUs is to reduce the number of operations each ALU can support and spread the total number of required operations out over the stripe. While the ALUs within the stripe are heterogeneous, each stripe in the fabric remains identical.

When creating a heterogeneous stripe it is important to distinguish what operations to include, and how often to include each operation in the stripe. Application analysis is used to determine the set of operations to include in the fabric, as well as their frequency. By analyzing representative applications, it is possible to extrapolate the needs of similar applications and create a heterogeneous fabric capable of supporting applications in the same computation class.

Each stripe is capable of supporting a fixed number of total operations. This total is determined by the number of ALUs in the row and the number of operations each ALU can support. Consider a stripe with 20 ALUs each supporting 10 operations, the total number of operation slots within the row is 200. However, each ALU must support NOOP and pass gate operations leaving 160 slots. To fill these slots, the operations’ frequencies from the application suite is calculated as a percentage and used as a weight to fill the remaining slots. For example, if the AND operation is used by 10% of the nodes in the applications, then 16 of the 20 ALUs contain the AND operation evenly distributed across the row. In practice, the number of slots requested by a node could exceed the number of ALUs in the row, in which case the number of slots that can be occupied is capped at the maximum row width. By removing these extraneous operations, additional slots are made available to remaining operations.

4 The Fabric Instance Model (FIM)

The fabric instance model (FIM) was designed as a textual representation to describe the interconnect and the layout and make-up of the ALUs 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) [5]. Although we could have created a more compact proprietary FIM specification, XML was selected in part due to the existence of several XML parsers. It also allows the FIM specification to easily evolve as new features and descriptions are 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 ALU structures.

Figure 5 shows an example partial FIM file that describes a 5:1-based interconnect. The pattern repeats the interconnect pattern for \textit{alu0}, whose zeroth operand can read from two units to the left and one unit to the right, and the first operand is the mirror. The second operand is the selection bit if the ALU is configured as a multiplexer and follows the first operand. The ranges in the FIM can be discontinuous by supplying additional range flags. The file can contain a heterogeneous interconnect by defining additional FTUs with different interconnect ranges and heterogeneous ALUs. The pattern can either repeat or can be arbitrarily customized without a repeating pattern for a fixed size fabric. Thus, by using the FIM we can explore many different architectural configurations with automated instance and mapper generation.

In our experiments we used various FIMs to vary the interconnect flexibility, number of dedicated pass gates, and make-up of heterogeneous ALUs as well as various combinations of these architectural instantiations.

\begin{verbatim}
<rowpattern repeat="forever">
  <row>
    <ftupattern repeat="forever">
      <FTU type="alu0">
        <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>
\end{verbatim}

Figure 5. FIM file for 5:1 interconnect.

5 Mapping

To map applications onto the fabric, we have developed a heuristic mapping algorithm. The implementation of this algorithm, the “mapper,” reads both the algorithm and the FIM to generate its mapping result. The heuristic follows a top-down mapping approach, starting with the top row and each individual row is completely placed using a limited look-ahead of two rows. 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 width equal to the largest individual row, and 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. The heuristic is comprised of two stages of row assignment and column assignment. These are described in detail in the next two subsections.

5.1 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 (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 given node. All non-immediate children nodes are disconnected from the given 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 fan-out exceeds the maximum as defined by the FIM. If a node’s fan-out exceeds the limit, a pass gate is created with an edge from the given node. In order to reduce the given node’s fan-out, children nodes are disconnected from the given 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 given node to the pass gate. If the fan-out 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 given row, then subsequently for all rows in the graph as shown in Figure 6.

5.2 Column Assignment

For each row, the heuristic first creates a parent dependency window for each node, a child dependency window
for nodes that have a child node in the proceeding row, and a grandchild dependency window for nodes with a child node two rows down.

The parent dependency window (PDW) lists all ALU locations which satisfy the primary constraint that the given 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 ALU. Figure 7 shows an example of a PDW dictated by the interconnect description.

The child dependency window (CDW) lists all ALU locations which 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 potential locations of a given node as well as the PDW created from potential locations of any nodes that share a direct child with the given 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 given node and connected nodes. A child dependency window example is shown in Figure 8.

The grandchild dependency window (GDW) provides an additional row of look-ahead. The GDW lists all ALU 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 ALU 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 given row. The desirability value is equal to the number of non-mapped nodes that contain the given location in their PDW, CDW, or GDW.

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 given row, 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.

When creating the dependency windows for heterogeneous FIMs, the mapper must consider both the range restrictions as well as the ALU operational restrictions of the fabric model. While a valid placement for a node may be available based strictly on the interconnect, the node’s operation may not be supported by any of the reachable locations. In this case, the mapper relies on pass gates to propagate the value towards a node in a later row that provides the required functionality. This creates pressure to increase the height of the fabric. To assist in mapping feasibility, the mapper assumes that each ALU supports pass gate and NOOP functionality.

6 Results

In order to evaluate power and performance, a set of core signal processing benchmarks were selected from MediaBench benchmark suite including the ADPCM encoder (enc), ADPCM decoder (dec), GSM channel encoder (gsm), and the MPEG II decoder (row, col). We added the Sobel (sob) and Laplace (lap) edge detection algorithms to the benchmark suite. In the subsequent sections, we study the impact of introducing heterogeneity in the ALU stripes onto
power, performance, energy, and area. We examine the benefit of adding dedicated vertical routes in the fabric and reducing the number of operations supported by the ALUs in terms of energy and area savings.

The design flow overview using the FIM is shown in Figure 1. The SuperCISC Compiler [9, 10] takes C code input which is compiled and converted into an SDFG. The SDFG is then mapped into a configuration for the fabric described by the FIM. The FIM is also used to automatically generate the VHDL for the fabric instance described by the FIM. The fabric instance VHDL is synthesized into an Oki cell-based ASIC design with a feature size of 0.16 µm using Synopsys Design Compiler. This netlist and the mapping of the application are then fed into Mentor Graphics ModelSim where correctness can be checked. A value change file (VCD) output from the simulation along with the design netlist can then be used to determine the power consumed in the design. We used Synopsys PrimePower tool to estimate the power consumption of the device.

6.1 Including Dedicated Pass Gates

We took the top performers 8:1 and 5:1 interconnects from our interconnect cardinality study which ranged from 32:1 to 3:1 cardinalities with both homogeneous and heterogeneous interconnect structures [1], and varied the percentage of ALUs replaced with dedicated pass gates at levels of 25% (1 out of 4), 33% (1 out of 3), and 50% (1 out of 2) dedicated pass gates. While 25% dedicated pass gates stretched the 5:1 interconnect to its limits, the 8:1 interconnect has enough flexibility and the benchmarks contain enough need for pass gates as seen by Figure 4 that 33% and 50% may be reasonable, particularly for sparse subgraphs.

The fabric size requirements for including dedicated pass gates are shown in Table 1. The width value includes full ALUs and dedicated pass gates, so a width of 20 with 25% dedicated pass gates contains 15 full ALUs and 5 dedicated pass gates. While the energy consumed in the fabric is impacted by the height of the benchmark implementation, this is not the only factor; and in these results, the implementation sizes do not always correlate well to the energy consumed in the fabric. One reason is that the overall size can remain constant while the length of non-critical paths can lengthen a great deal.

Reducing the number of ALUs used as pass gates reduces energy. We assume that the amount of energy consumed by a dedicated pass gate is negligible compared to an ALU configured to be a pass gate. Increasing the path length increases the energy. Table 2 contains the impact of adding dedicated pass gates on the number of ALUs used as pass gates and Table 3 shows the increase in path length over the 0% dedicated pass gate case for each interconnect cardinality.

Figure 9 shows energy results for varying each of the percentage of dedicated pass gates for the two chosen interconnects. The trend for 5:1 shown in Figure 9(a) shows that including dedicated pass gates does not help the overall energy improvement which follows from spikes in the path length. In fact, when using 25% dedicated pass gates, the energy increases. When the level of dedicated pass gates is 50%, this tradeoff returns to a similar energy compared to no dedicated pass gates. The trend for 8:1 is better. The path length was not as dramatically increased in this case and the energy dropping below the non-dedicated pass gate result with 25% dedicated pass gates and dropped further with 33%. The change between 33% and 50% is nominal.

To confirm that the energy consumed by each implementation is affected by the number of ALUs used as vertical

| Table 1. Fabric size (Width x Height) for mapping benchmarks onto interconnects with different percentages of dedicated pass gates. |
|---|---|---|---|---|---|---|---|
| enc | dec | row | col | gsm | sob | lap |
| 5:1 (0% DP) | 20x18 | 16x13 | 20x8 | 17x13 | 20x19 | 20x19 | 10x9 |
| 5:1 (25% DP) | 17x19 | 16x14 | 20x9 | 18x15 | 20x20 | 14x18 | 10x9 |
| 5:1 (33% DP) | 20x18 | 20x14 | 19x10 | 20x14 | 20x19 | 20x18 | 11x9 |
| 5:1 (50% DP) | 20x18 | 16x14 | 25x11 | 21x14 | 21x20 | 14x19 | 20x13 |
| 8:1 (0% DP) | 17x16 | 16x13 | 15x8 | 17x10 | 20x12 | 14x18 | 10x9 |
| 8:1 (25% DP) | 17x16 | 16x13 | 17x8 | 17x10 | 20x12 | 14x18 | 10x9 |
| 8:1 (33% DP) | 17x16 | 16x13 | 19x8 | 17x11 | 20x12 | 14x18 | 11x9 |
| 8:1 (50% DP) | 17x16 | 16x13 | 25x8 | 21x10 | 21x12 | 14x18 | 15x9 |

| Table 2. Number of ALUs used as pass gates for various interconnect strategies. |
|---|---|---|---|---|---|---|
| enc | dec | row | col | gsm | sob |
| 5:1 (0% DP) | 170 | 85 | 103 | 165 | 138 | 20 |
| 5:1 (25% DP) | 110 | 56 | 86 | 104 | 80 | 8 |
| 5:1 (33% DP) | 77 | 22 | 43 | 77 | 55 | 4 |
| 5:1 (50% DP) | 57 | 28 | 36 | 50 | 48 | 5 |
| 8:1 (0% DP) | 137 | 81 | 58 | 88 | 129 | 19 |
| 8:1 (25% DP) | 78 | 44 | 22 | 56 | 82 | 6 |
| 8:1 (33% DP) | 57 | 30 | 16 | 40 | 63 | 2 |
| 8:1 (50% DP) | 26 | 11 | 8 | 12 | 32 | 1 |

| Table 3. Increase in overall path length by addition of dedicated pass gates. |
|---|---|---|---|---|---|---|
| enc | row | gsm | sob |
| 5:1 (25% DP) | 5 | -1 | 0 | 2 |
| 5:1 (33% DP) | 1 | -1 | 0 | 2 |
| 5:1 (50% DP) | -1 | 1 | 2 |
| 8:1 (25% DP) | 1 | 0 | 0 | 0 |
| 8:1 (33% DP) | 1 | 0 | 0 | 1 |
| 8:1 (50% DP) | 1 | 0 | 0 | 2 |
routes and the change in the overall path length (i.e. length of all output paths), we ran a two-way analysis of variance (ANOVA) on the energy with the number of ALUs 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 \( (p < 0.01 \text{ and } p = 0.031, \text{ respectively}) \).

6.2 Heterogeneous ALUs

We studied the impact of reducing the number of operations supported by the ALUs in the fabric on energy. Figure 10 shows the energy results for varying the number of operations supported by the ALUs from 23 to 16, 10, and 8. The 8:1 bar is our baseline and it corresponds to the ALU that supports 23 operations. As the number of operations reduces from 23 to 16, there is a marked decrease in energy consumption but as we go from 16 to 10, the decrease is less dramatic. However, when we further reduced the number of operations to 8, the mapping problem became more difficult and required a larger fabric to fit all of the benchmarks leading to an increase in the energy consumption. Thus, the fabric with 10-operations ALU is the best candidate as it has the lowest energy for all the benchmarks.

6.3 Combining Heterogeneity with Dedicated Pass Gates

Based on our study where we varied the percentage of ALUs replaced with dedicated pass gates at levels of 25%, 33%, and 50%, the results clearly indicate that 8:1 interconnect with 33% dedicated pass gates requires the minimal energy. We selected our top performer from Section 6.2 and studied the combined effect of using dedicated vertical routes and reducing the number of operations supported by the ALUs. We tried the 16-operation ALU and 10-operation ALU with 8:1 interconnect cardinality and 33% dedicated pass gates. The results are shown in Figure 11. The 10-operation ALU with the 33% dedicated pass gates configuration consumes the least energy among all the candidates considered for all the benchmarks.
least area. It provides area savings of 50% over the baseline case. However, 10-operations with 33% dedicated pass gates is not much larger and saves 47% over the baseline architecture.

Table 4. Area results for various instances.

<table>
<thead>
<tr>
<th>Fabric instances</th>
<th>Area (µm²)</th>
<th>Area with 33% DP</th>
</tr>
</thead>
<tbody>
<tr>
<td>23 ops</td>
<td>8362981</td>
<td>6057959</td>
</tr>
<tr>
<td>16 ops</td>
<td>4940466</td>
<td>5194891</td>
</tr>
<tr>
<td>10 ops</td>
<td>4190375</td>
<td>4463338</td>
</tr>
<tr>
<td>8 ops</td>
<td>5145491</td>
<td>no mapping</td>
</tr>
</tbody>
</table>

7 Conclusions

In this paper, we describe the benefit of adding dedicated vertical routes and operation heterogeneity into the fabric architecture. The 8:1 interconnect with 33% dedicated pass gates consumes least energy among 8:1 and 5:1 baseline interconnects and their extensions with different percentages of dedicated pass gates. This is due to a dramatic increase in path length (leading to an increase in energy) without a sufficient decrease in ALUs used as pass gates (which would reduce the energy) in the 5:1 case.

We fixed the interconnect to be 8:1 and we explored heterogeneity in the functional units by introducing heterogeneous ALUs into the fabric. The results clearly indicate that the fabric having ALUs that support 10-operations with 8:1 interconnect consumes the least amount of energy among our 8:1 heterogeneous architectures reducing the energy consumed by 24% when compared with the baseline 8:1 architecture. We also studied the combined effect of adding dedicated pass gates and heterogeneous ALUs. We found out that the fabric with 10-operations ALUs and 8:1 with 33% dedicated pass gates provides the best energy result among all the cases examined here. This reduces the energy consumption by 32% over the baseline 8:1 architecture. The heterogeneity in the fabric architecture not only helps in reducing energy, but also provides area savings. The fabric with ALUs that support 10-operations and 8:1 interconnect with 33% dedicated pass gates provides area savings of 47% over the baseline case.

To assist in testing and examining the impact of these different architectures on energy consumption, an automation process was created to automatically generate fabric instances and to configure mappers based on a FIM, or textual model of the functional units and interconnect in the system.

References


