Optimization of Module Placement during FPGA Partial Reconfiguration

Jin Hu
Northwestern University

Elaheh Bozorgzadeh
University of California Irvine

Love Singhal
University of California Irvine

Jin@northwestern.edu, {**eli**, **lsinghal**}**@ics.uci.edu**

Abstract. With the novel feature of partial reconfiguration, FPGAs (field programmable gate arrays) have the potential to become even more powerful and versatile. The ability to reconfigure during run-time offers improvement in task flow and reduction for total time required. This development of dynamic programming then leads to the problem of module placement – given a series of tasks, what is the best implementation of each task such that the amount of time needed for partial reconfiguration is minimized? To solve this problem, both the placement and interface of the logic components must be taken into account. This paper not only discusses the behavior of standalone modules but also looks at how they behave when part of a bigger, more complex system.

Keywords. Partial reconfiguration, module placement, FPGA, design implementation

1. Introduction

A built-in feature of the Virtex-4™ FPGA (field programmable gate array) board, partial reconfiguration gives users the luxury to modify the contents of the FPGA board during runtime. Without the usage of partial reconfiguration, only static programming was available – once the board is written to, it cannot be changed. With partial reconfiguration, however, the concept of dynamic programming is introduced – part of the board can be modified whilst the rest remain untouched. Not only does partial reconfiguration allow the support the execution of different tasks on a single FPGA board, it also can potentially decrease the total amount of time needed for task execution.

Partial reconfiguration is a process in which bits are streamed serially and reprograms the FPGA board column by column or frame by frame. To optimize the amount of time needed to partially reconfigure, the number of bits required must be minimized. This then leads to the observation of shared logic components. If two tasks, Task 1 and Task 2, both share a common module, for example a 32-bit multiplier, there is no reason to overwrite the Task1's 32-bit multiplier – the common module could be used for Task 1 and then reused for Task 2. By reusing a module, the number of bits needed to reconfigure from Task 1 to Task 2 potentially decreases.

2. Motivation

The process of partial reconfiguration is advantageous both in terms of time and space. Between any two given tasks, the more shared logic components there are, the higher the potential to save on the number of bits. Even in the worst case, if there are no common components between the two tasks, partial reconfiguration is still cheaper, both in terms of area and time, than reprogramming the entire board.

Not only can partial reconfiguration drastically save on the amount of time needed for task transition, it also can reduce on the amount of real estate needed. The area used to support modules only found in Task 1 can easily be used to support modules only found in Task 2. With a combination of both static and dynamic programming, not only can the flow of task execution be improved but also the total amount of time needed can be reduced.

3. Related Work

In the field of partial reconfiguration, there has been a substantial amount of work done already. In particular, there are two subtopics of interest – bit difference and algorithmic mapping. It has been shown that given a typical reconfiguration bit stream, less than 3% of the total bits differ [2]. However, because the bits are scattered about in many different frames, the reconfiguration cost grow exponentially larger. Following up on this problem, a technique has been developed to alter the granularity of reconfiguration, namely going by individual bits instead of the number of frames. The results show a competitive reduction in costs, around the order of 80%.

The second important piece of research done shows that purposeful (non-arbitrary) mapping significantly reduces reconfiguration costs [1]. The work done generalizes this process using high-level algorithmic blocks which are analogous to the logic components discussed in this paper. The algorithm described specifies a partitioning method to match up components based on both resource usage and execution time. The work done in this paper uses a simplified version of the previously mentioned method. Namely, the mapping is done on a smaller scale and is manually chosen to match up modules that perform the exact same task.

The groundwork of this paper is built upon the previous findings. The work done extends upon that foundation and looks at (ultimately) reconfiguration savings between two designs. The results discussed specifically focuses on the FPGA Virtex-4™ board and its logic components.

4. Problem Description

The overall goal to reduce the total amount of time hinges on the amount of transition time needed between tasks. Specifically, the number of bits needed to partially reconfigure must be minimized. This then leads to the question of module placement – given a series of tasks, what is the optimal placement of logic components such that bit reconfiguration is minimized? Within this problem, there are two main issues to consider: the physical placement and the interface of the modules.

To better understand how each consideration affects the number of bits needed, different constraints must be placed on the logic components. There are three constraints considered: real estate, number of frames and bits, and timing (clock frequency). Each aspect is important to consider, as each affects the module placement and performance differently. The constraints are designed in concurrence with the others, as they are all interrelated.

4.1. Preliminaries

A module is a catch-all term for any logic component that uses both inputs and outputs. This includes, but is not limited to, adders, multipliers, and dividers. This paper mainly focuses on multipliers, as out of the three, they are the dominant components. All synthesis and implementation of the modules is done using the Xilinx® ISE v7.1 software bundle. The modules themselves are generated using CORE Generator and are placed on the Virtex-4™ FPGA board.

4.2. Considerations

To further clarify the constraints imposed on the designs, the following explains in detail what each constraint entails.

4.2.1. Real Estate

The amount of space needed on the FPGA is perhaps the easiest to visualize. The foremost concern is that the module must fit onto the chosen board. It is highly unlikely that in any given design only one module is used, so the amount of room per module should be minimized. The

space is measured in terms of slices or configurable logic blocks (CLBs); two slices equate to one CLB. Note that the area refers to the entire implemented version – namely, it not only includes the logic but also the routing required. The areas reported in this paper only refer to the specified constrained area – the actual area of the module will be slightly greater due to routing.

When implementing modules, not only does physical space of the module matter but also the shape of the module. The Xilinx® software package allow the modules great flexibility, as they can be any rectangular shape. However, with the overall goal in mind, to reduce the number of bits needed to reconfigure, the shape of the module will follow in accordance to frame height, a subject discussed in greater detail in the section below.

4.2.2. Timing

The strictest constraint of all, the clock frequency in which a module operates under does not have as much leeway as the previously mentioned constraints. If timing cannot be met, then the entire design must be modified such that the time the module runs under is acceptable. Thus, whenever using Xilinx®, a tight timing constraint needs to be specified – the tool only checks if the timing constraint is met and will not optimize further if the current design passes. The caveat to this is that if the timing constraint is too tight, then occasionally the tool is not able to meet it. When the tool fails, it fails miserably and gives a much worse 'actual' timing than if a slightly looser timing constraint was given.
The timing constraint also affects the design, namely the routing, of the implemented module in two ways: congestion and wire type. The tighter the timing constraint, the more chaotic the routing becomes. To alleviate this problem, it is possible to loosen the timing constraints slightly to achieve a better implementation. A tighter timing constraint would also employ the use of short (faster) wires.

If too loose of a constraint is imposed, it is possible for Xilinx® to use long (slower) wires and save the short wires for other more modules that have stricter requirements. For all modules implemented, the timing, given as the clock period, is restricted on the nanosecond [ns] scale.

4.2.3. Frames and Bits

During partial reconfiguration, the number of frames to reprogram directly correlates to the number of bits to reprogram.

The conversion factor is as follows: $1\ \text{frame} \times \frac{40\ \text{words}}{1\ \text{frame}} \times \frac{4\ \text{bytes}}{1\ \text{word}} \times \frac{8\ \text{bits}}{1\ \text{byte}} = 1280\ \text{bits}$.

As a rule of thumb, every CLB is approximately equal to 22-23 frames. If even one bit in the entire frame needs to be reprogrammed, then the entire frame needs to be reprogrammed. Thus, it is important that the each frame is used to its fullest extent. For the Virtex-4™ FPGA board, each frame spans a height of 16 CLBs or 32 slices (1 CLB = 2 slices). Thus, it is advantageous that all modules are designed in accordance with frame height (every 16 CLBs). All modules then are designed in the following manner: the height of the module is first determined with respect to the frame height and the width is minimized afterwards.

4.3. Formulation

The approach for minimizing the number of bits during partial reconfiguration is broken down into four main sections: single modules, merged modules, interface, and design integration. The first three sections deal with standalone implementations. Once individual components are understood, they can be integrated within larger and more complex designs.

4.3.1. Analysis of Single Modules

The algorithm begins by looking at the single logic components. These are, but not limited to, adders, multipliers, and dividers. The data of primary interest involves the optimal sizing of the module and the number of frames (and consequently the number bits) needed during partial reconfiguration. The data is measured across different standard (8, 16, and 32) input bit-widths. This paper's discussion primarily focuses on multipliers as they are the dominant component out of the three mentioned modules.

4.3.2. Analysis of Merged Modules

This section focuses on what happens when multiple single modules are merged together as a single module. The purpose is to see if there is any advantage into grouping separate modules as one instead of mapping each single module separately. The discussion focuses on the multiplier-adder, or MAC, a common component used in many designs.

4.3.3. Analysis of Module Interface

Finishing up standalone modules, this section discusses how interface affects module behavior. In this section the modules' interface (inputs and outputs) are bound in specified areas, whilst in the previous two sections the inputs and outputs were left unconstrained. The empirical data collected for this section uses multipliers using similar reasoning as above.

4.3.4. Dataflow Graph Design Integration

After individual analysis of modules and module interface, the logic components are then implemented as part of a bigger design given by dataflow graphs (dfgs). As the name implies, data flow graphs only provide the flow of the data and not the implementation. Thus, using two different dfgs, various arrangements are tested to see the effects of module placement. Different mappings are also done to see how mapping affects the performance of the designs.

5. Individual Analysis

Based on the approach specified above, the methodology of each section is explained in greater detail. The conclusions are then discussed based on the various experiments' results.

As further clarification, the column heading are defined as follows:

- **Area?** – A yes/no field denoting whether area constraints were placed on the module.
- **Time (Actual)** – The given timing constraint and the actual running time (in parenthesis).
- **Width** – The number of slices needed for the module length-wise (x-direction).
- **Height** – The number of slices needed for the module height-wise (y-direction).
- **Total Area** – The total real estate needed on the FPGA board for logic placement ONLY. Note that this does not include extra routing.
- **Frames Needed** – The number of frames needed to reconfigure as given by Bitgen, a program part of the Xilinx® software package.
- **.bit Size** – The size of the .bit file generated by Bitgen which specifies the exact number of bits needed to reconfigure. Note that this field is given in terms of bytes and not bits.

The other clarification point relates to the generated graphs. As shown, the independent axis displays frame height. However, note that there is a data point at a supposed frame height = 0. This, of course, is not feasible. This data point signifies the module when no area constraints are introduced – the purpose is to see what Xilinx® would do if given no boundaries.

5.1. Single Modules

The discussion focuses primarily on 32-bit multipliers, as they not only are commonly used components but also are large enough to show clear trends and results. The data is summarized in the table below.

32-Bit Multiplier						
Area?	**Time (Actual) [ns]**	**Width [slices]**	**Height [slices]**	**Total Area [slices²]**	**Frames Needed [frames]**	**.bit Size [bytes]**
no	12.000 (11.946)	28	48	1,008	633	114,645
yes	12.000 (11.868)	32	26	832	354	63,653
yes	12.000 (11.997)	12	60	720	295	52,289
yes	12.000 (11.951)	8	92	736	316	56,537
yes	12.000 (11.975)	6	120	720	400	75,453
yes	12.000 (11.979)	4	192	768	548	108,201

Table 1: Experimental Data for 32-Bit Multipliers

5.1.1. Real Estate

According to the synthesis report generated by Xilinx®, the 32-bit multiplier needs 591 slices purely for logic. As shown from the table above, the area (along with the dimensions) used clearly exceed this number. The extra space is used for routing during implementation – not only does the logic need to be placed but the routing also must be possible given the logic layout. The area and dimensions displayed in the table (as noted above) is the bounding box specified when implementing the module. Note that for the given data the logic is completely placed within the bounds but some wires may still be outside. This becomes more evident as the module width becomes smaller.

The graph below shows the relationship between frame height and the total area needed. Observe that the when the tool is not given an area constraint, the module uses up substantially more space than any other constrained arrangement. Thus, it is an advantage to specify area constraints as to limit the amount of board usage.

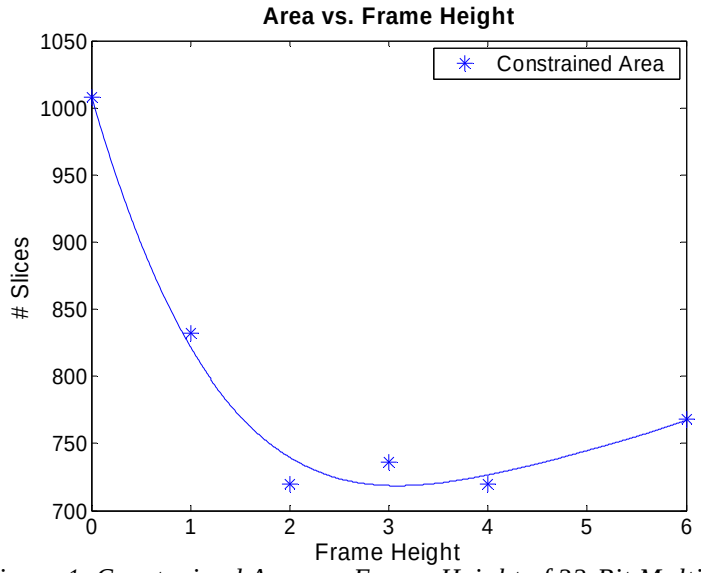


Figure 1: Constrained Area vs. Frame Height of 32-Bit Multipliers

As the frame height increases, note that beyond the first frame height, the area needed for the multiplier stays relatively constant with the exception at the frame height of one. The reason for the slightly larger area is that for a frame height of one (limit 16 CLBs) the multiplier's width must be at least 16 CLBs. This then causes the minimal dimensions to be 16 by 13. Going beyond the first frame height, the required area stays around 750 slices. This then implies that the regardless of the shape or frame height, no extra area is needed. Thus, the module shape is flexible and is not limited or dependent upon the frame height. However, the module must have some type of constraint imposed, as the unconstrained result is worst out of the entire data set.

5.1.3. Timing

The amount of time needed for the multiplier to run falls slightly under 12 nanoseconds consistently for each trial. The imposed constraint of 12 ns is a relatively tight bound, as the multiplier is unable to perform under 11 ns. *Table 1* shows the amount of time needed for each trial. The data shows consistently that for any given 32-bit multiplier, the time required to run is slightly below 12 ns. If necessary, the timing can be reduced slightly, but not much more than the current results.

5.1.4. Frames and Bytes

Below shows the frames and bytes needed to implement a standard 32-bit multiplier. *Figure 2* displays two curves – an estimated value of the number of frames based on the constrained area and the actual number of frames needed for implementation. Notice that toward the smaller frame heights (unconstrained, 1, and 2), the estimate and actual numbers are similar. The estimated curve (blue) shows that the number of frames stays relatively constant, regardless of frame height. However, the actual curve (red) shows that there is a minimum at the frame height of 2 at which the number of frames is minimized. This deviation implies that as the shape of the implemented module becomes taller, the wiring plays a more and more noticeable role in determining the number of frames needed. As the module width becomes narrower, it becomes increasingly difficult to fit all the logic and the wiring inside a given width. Thus, it is important to not only take the logic needed into consideration but also the wiring that goes outside the bounded area.

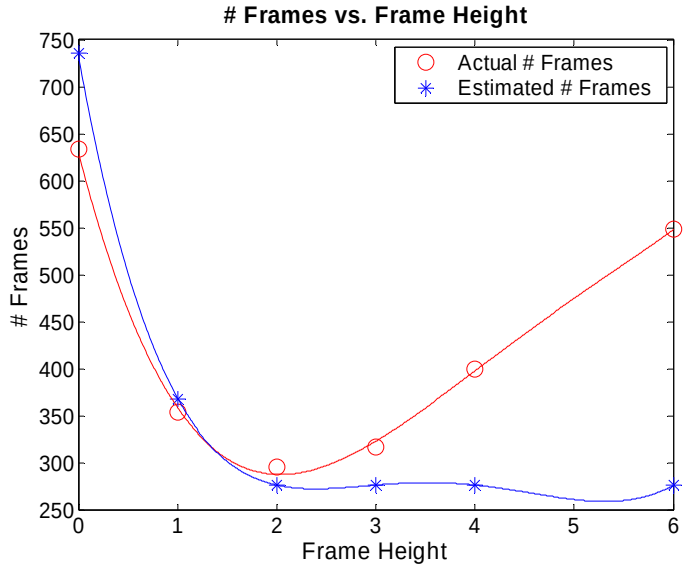


Figure 2: Number of Frames vs. Frame Height of 32-Bit Multipliers

Figure 3 also shows two curves – an estimated number of bytes and the actual number of bytes needed to reconfigure a 32-bit multiplier. The estimated number of bytes is based on the actual number of frames – each data point is the number of frames converted to bytes. Note that there is a difference between the two curves. This shows that in order to reconfigure the multiplier, the number of frames is not the sole, albeit major, contributor to the number of bytes needed. The extra number of bytes needed is the overhead of reconfiguration. This includes, but is not limited to, the addresses of each frame needing to be modified.

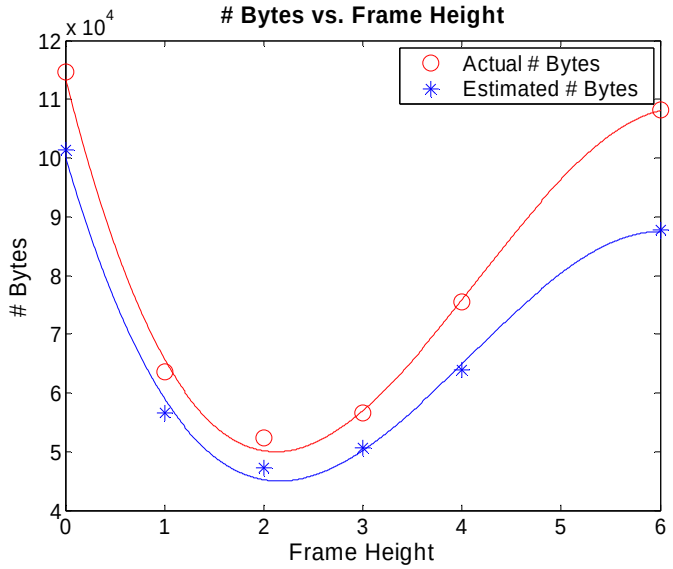


Figure 3: Number of Bytes vs. Frame Height of 32-Bit Multipliers

Another point of interest is that there is a relatively uniform distance between the two curves shown in *Figure 3*. This similar difference implies that the regardless frame height, the overhead needed stays relatively constant. This shows that the overhead needed is not based upon the shape of the module and is a constant amount regardless of implementation choices. From the graph, it is clear that the number of bytes is minimized at the frame height of 2 with 3 coming in at a very close second.

5.2. Merged Modules

The following table summaries the results for 32-bit multiplier-adders or multiplier-accumulators, more commonly known as MACs. Note that the area needed is comparable to the standalone multiplier but uses up more frames and bits. The MAC, if combined, is treated as one module by the tool and is able to fit in less space than the two separate components.

32-Bit Multiplier-Adder						
Area?	**Time (Actual) [ns]**	**Width [slices]**	**Height [slices]**	**Total Area [slices²]**	**Frames Needed [frames]**	**.bit Size [bytes]**
no	12.000 (11.996)	36	22	792	430	74,824
yes	12.000 (11.984)	32	26	832	386	67,392
yes	12.000 (11.907)	12	62	744	337	60,068
yes	12.000 (11.984)	8	92	736	326	56,732
yes	12.000 (11.991)	6	124	744	368	67,224

Table 2: Experimental Data for 32-Bit Multiplier-Adders

5.2.1. Real Estate

The 32-bit multiplier-adder needs a total of 605 slices for logic – 591 from the multiplier and 16 from the adder. However, the data shows numbers greater than the base value. This is again due to the routing and extra wiring needed for implementation. The graph below shows the general trend of how much real estate is needed as frame height increases.

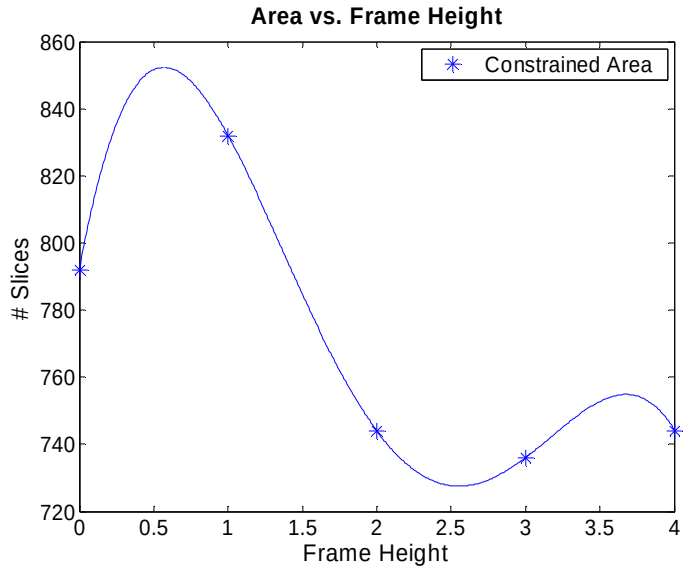


Figure 4: Constrained Area vs. Frame Height of 32-Bit Multiplier-Adders

One interesting point is that when the module is implemented unconstrained, the result is actually better than if the module has a frame height of 1. This could be due to the fact that because the component is very common, the tool knows how to optimize this specific case. Another reason could be that the multiplier-adder is not ideally implemented when forced to have to a frame height of 1. Like the standalone 32-bit multiplier, in order to have a frame height of 1, the width must be at least 16 CLBs (or 32 slices). Given this as a lower bound, the module's height does not stretch to the full 16 CLBs and only needs 13 CLBs. This reduction in height then leads to an unfortunate increase in the number of frames, a topic discussed in more detail in an upcoming section.

Another similarity between the multiplier and the MAC is that beyond the first frame height, the amount of area needed stays relatively constant and settles around 750 slices. This area is very comparable to the amount of space needed for a lone 32-bit multiplier. This then implies that the adder is integrated together with the multiplier to save space on the FPGA board and is cheaper than implementing a multiplier and an adder separately. As the trends for the MAC are similar to that of the standalone multiplier, it is clear that between the multiplier and the adder, the multiplier is the dominant component.

5.2.2. Timing

The table on the previous page lists the time required for the MACs for each frame height. Like the multiplier, the timing constraint of 12 ns is a relatively tight bound. The amount of time needed for the adder is negligible, even if the adder's output is registered. As before, the timing is predominantly influenced by the multiplier, so this component is very hard-pressed to run any faster than using a 12 ns clock cycle.

5.2.3. Frames and Bits

The graphs below depict the trends for the number of frames and the number of bits needed for the MAC. In *Figure 5*, the blue curve shows an estimated number of frames (manually estimated from FPGA Editor) and the curve in red shows the actual number of frames (given by Bitgen). As given by the graph, the both curves lead to different conclusions. The estimated curve levels off at a constant value of around 275 frames but the actual curve finds a minimum at a frame height of 3 (a frame height of 2 comes at a close second). Due to the wiring that falls outside the bounding box, the estimated number of frames is substantially lower than that of the actual.

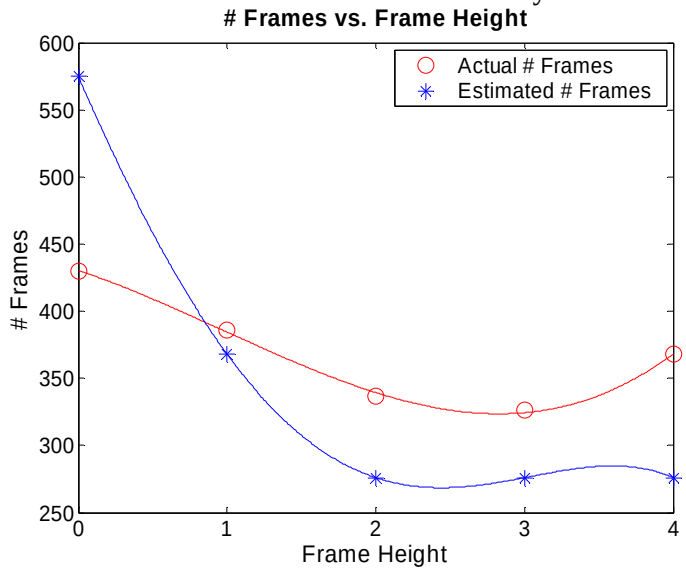


Figure 5: Number of Frames vs. Frame Height for 32-Bit Multiplier-Adders

In *Figure 6* depicts the trends of the actual number of bytes needed (red) and the estimated number of bytes needed based on the number of frames (blue). Note that the gap between the two curves is smaller than the gap for the standalone multiplier but the overall number of bytes is greater. As there are more frames to reconfigure, the number of bytes needed also increases. Because this module is relatively dense (logic is packed tightly), the frames are close to one another, cutting down on the number of addresses needing to be stored significantly. The similar shape of the two curves also implies that the amount of overhead is readily predicable.

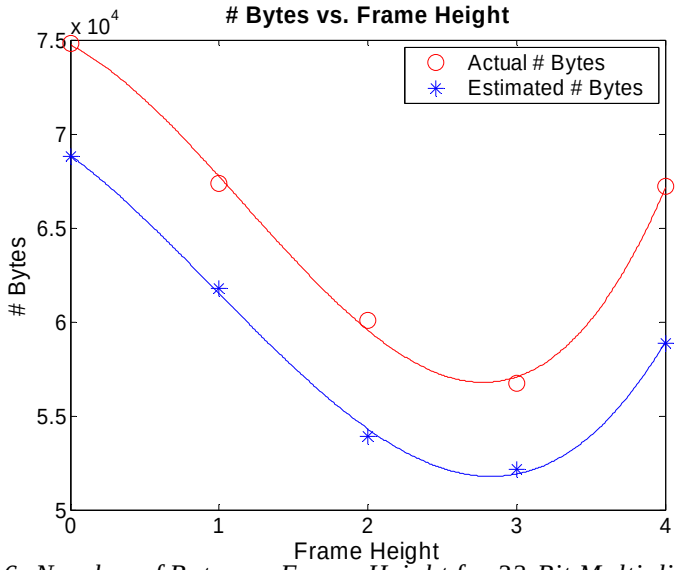


Figure 6: Number of Bytes vs. Frame Height for 32-Bit Multiplier-Adders

5.3. Overhead for Modules

Figure 7 shows the trend between the number of bytes and the number frames needed. The data not only includes a compilation of experiments mentioned above, but also some external experiments that are not discussed in this paper.

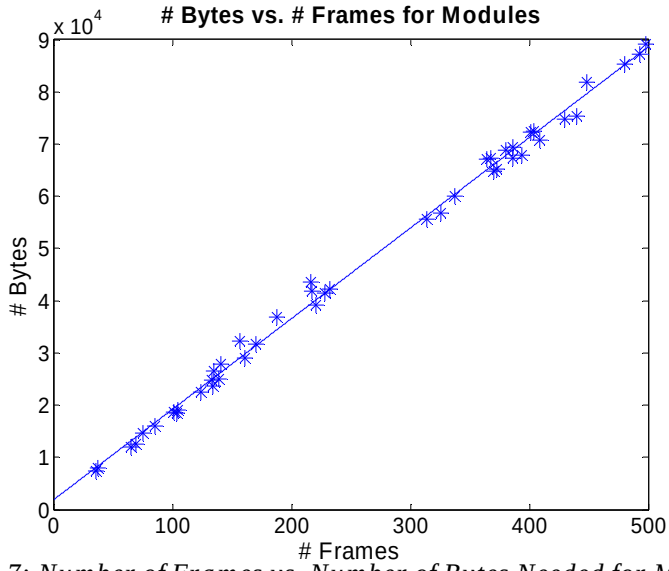


Figure 7: Number of Frames vs. Number of Bytes Needed for Modules

From the graph, there is a strong linear correlation between the number of frames and the number of bytes needed for partial reconfiguration. This data also takes into account the overhead required to implement. The multiplier and the multiplier-adder are both contiguous logic components, so the relationship is predictable – there are no substantial gaps in the logic modules to lead to strong variation.

5.4. Module Interface

The final standalone set of experiments done also involves multipliers on this time incorporates interface. The previous data only controlled the module shape and not where the inputs and outputs are located. In order to effectively control the direction of data, the I/O needs to be specified. This is extremely important during partial reconfiguration, as to be able to reuse a component, not does the function need to match but the interface as well.

To constrain the inputs and outputs, the concept of wrappers is introduced. Wrappers merely restricts where all the input and output pins need to lie relative to the module itself. The advantage of these wrappers is that they are moveable – they can be placed anywhere around the module, even far away (though this is counterproductive to the overall goal). However, once a location is chosen, the wrappers remained fixed and the entire set (module plus wrappers) is reconfigured. For every module (shown in blue), there are two input wrappers (shown in red) and one output wrapper (shown in violet).

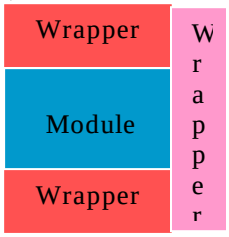


Figure 8: Example Picture of Module plus Wrappers

Table 3 shows the results of a standard 32-bit multiplier with wrappers. Multipliers are used as previously they have demonstrated to be a dominant component. The dimensions of the wrappers are not included in the table as the extra resources used vary, depending upon the placement. Specifically regarding real estate, the wrappers, although taking up room during constraint placement, use up only a small amount of space when implemented. Note that the entire package (module plus wrappers) are all contained with the specified frame height.

32-Bit Multiplier with Wrappers						
Area?	**Time (Actual) [ns]**	**Width [slices]**	**Height [slices]**	**Total Area [slices²]**	**Frames Needed [frames]**	**.bit Size [bytes]**
no	12.000 (11.987)	36+	22+	1218	448	77,375
yes	12.000 (11.972)	32+	26+	1050	421	75,184
yes	12.000 (11.952)	12+	62+	940	385	69,687
yes	12.000 (11.977)	8+	92+	946	429	80,415

Table 3: Experimental Data for 32-Bit Multipliers plus Wrappers

5.4.1. Real Estate

The constrained area trend follows that of a multiplier with no wrappers. The unconstrained data point is extremely high and as frame height increases, the number of slices needed levels off to a relatively constant number. The wrappers also take up extra room on the board, but when implemented, they take up relatively little room and there is no significant increase in the number of frames.

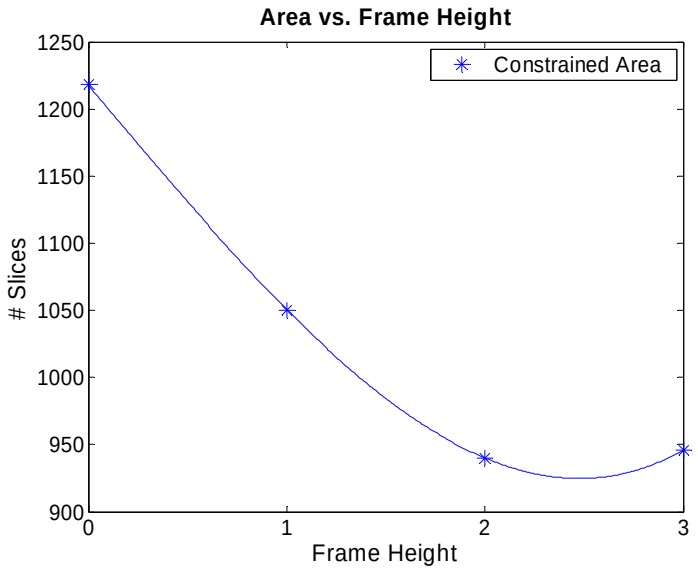
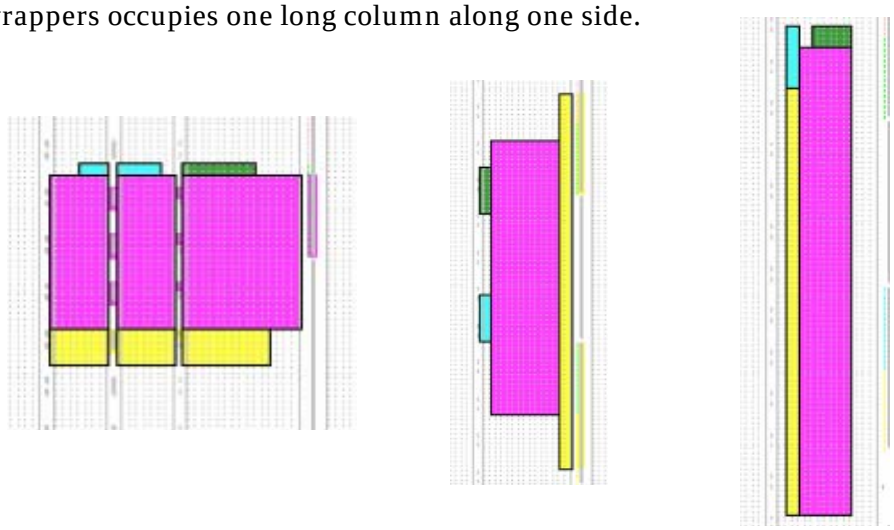


Figure 9: Constrained Area vs. Frame Height of 32-Bit Multipliers plus Wrappers

Note that the total area used has more variation than a standalone multiplier implementation. This is due to the placement of the wrappers – different arrangements cause the logic, and subsequently the routing, of the multiplier to vary. Similar to its non-wrapper counterpart, the multiplier plus wrappers set needs to have a specified area constraint, as the constrained is worse than all other data points.

Figure 10 shows the different optimal arrangements of the different frame heights. Each sub-figure includes the multiplier (purple), two input wrappers (turquoise and green), and one output wrapper (yellow). Notice that with a frame height of 1, the wrappers are able to fit along the top and bottom of the module. However, as the width becomes narrower, the result is best when the wrappers occupies one long column along one side.



Frame Height: 1 ***Frame Height: 2*** ***Frame Height: 3***

Figure 10: Various Optimal Arrangements of Wrappers for 32-Bit Multipliers with Wrappers

5.4.2. Timing

The wrappers themselves are constructed using combination logic, so they do not contribute any extra to timing performance. Thus, they follow the same behavior as a regular multiplier. The timing constraint of 12 ns is a tight bound and each trial, with some variation, meets this timing requirement.

5.4.3. Frames and Bits

The graph below shows the estimated and actual number of frames needed to reconfigure a multiplier plus wrappers. Note that for any frame height, including the unconstrained version, the estimated number of frames stays the same. However, the actual number of frames clearly finds a minimum at a frame height of two. This is consistent with the results found for the standalone 32-bit multiplier.

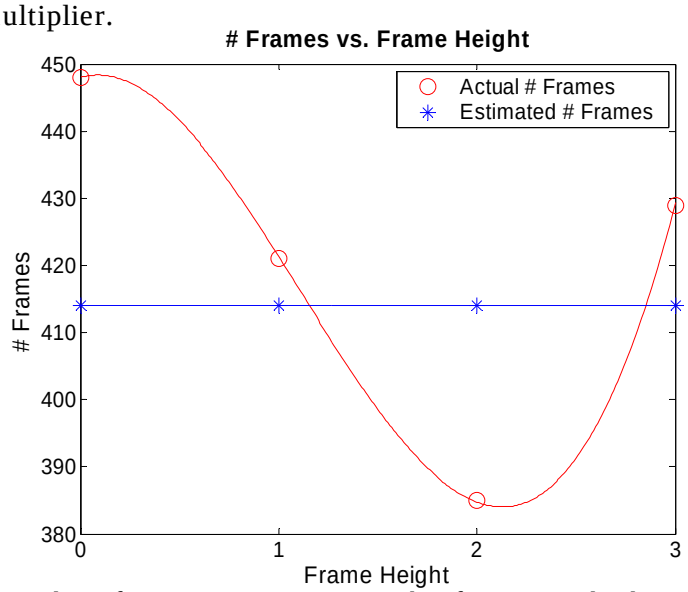


Figure 11: Number of Frames vs. Frame Height of 32-Bit Multipliers plus Wrappers

The graph below shows the estimated number of bytes needed (based on the number of frames) and the actual number of bytes needed. The two curves are remarkably similar in shape. This then implies that the overhead needed is predictable.

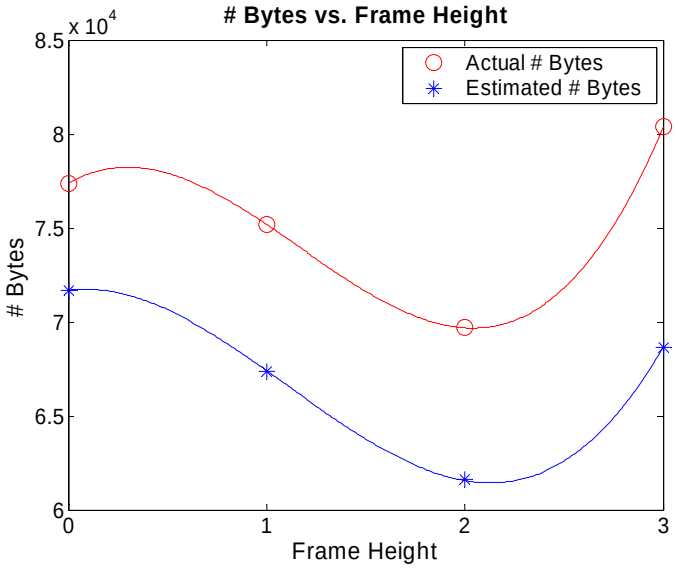


Figure 12: Number of Bytes vs. Frame Height for 32-Bit Multipliers plus Wrappers

This result is consistent with the standalone multiplier, having a minimum at a frame height of 2 but significantly different than the multiplier-adder, which has a minimum at a frame height of 3. However, with the MAC, a frame height of 2 yields a very close result to that of the minimum. Thus, a consensus can be made that across all fields, the optimal design is when the module has a frame height of 2.

5.5. Overhead for Modules with Interface

The graph below shows the relationship between the number of frames and the corresponding number of bytes needed to reconfigure a 32-bit multiplier plus wrappers. The different data points all includes the above mentioned results as well as outside experiments that are omitted in this paper. Note that there is more variation in this set, so the relationship is not as strong as that of the modules without wrappers. The implementation of the wrappers leaves gaps along the sides, causing the module to be slightly uneven. Despite the slight fluctuations, however, the relationship between the number of frames and of bytes is clearly linear.

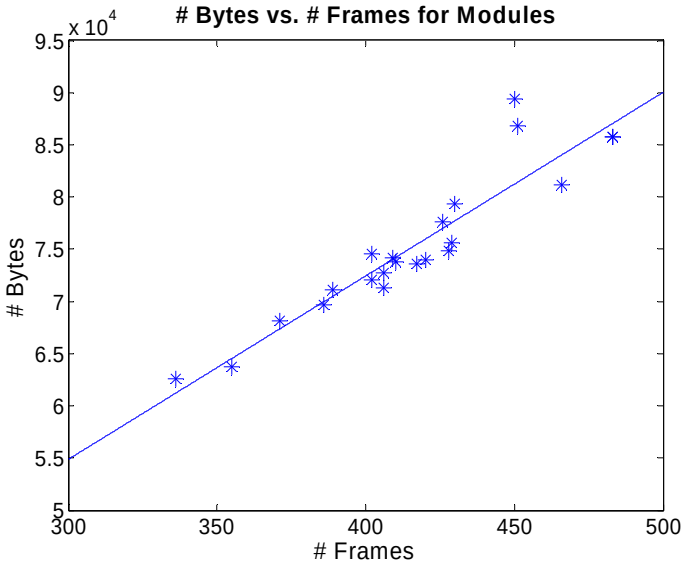
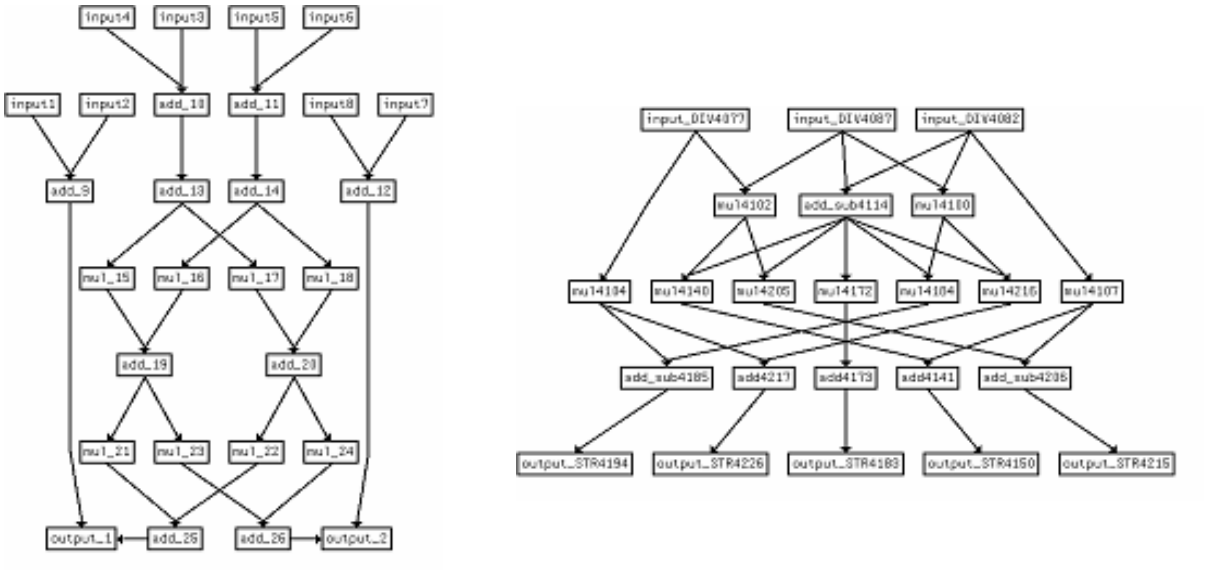


Figure 13: Number of Frames vs. Number of Bytes Needed for Modules with Wrappers

6. Dataflow Graphs

Now that the basic components have been covered, they can be incorporated into more advanced topics, namely dataflow graphs (dfgs). Dataflow graphs very elegantly show the execution path of inputs to outputs. Shown below are two examples of dfgs, both of which are used in the following experiments, where **Design 1** is used as a base design and **Design 2** is the new incoming task. The overall goal is to minimize reconfiguration time, which in turn is to minimize the number of bytes that differ.



Design 1 ***Design 2***

Figure 14: Two Designs of Dataflow Graphs

6.1. Considerations

As dfgs only show the flow of data (like the name implies), the implementation is completely up to the designer. This freedom, albeit flexible, comes with a slew of design considerations. The remainder of this paper focuses on the mapping aspect. Mapping is a process in which, given a module that is common in two designs, that module is kept and reused. Note that this is different than overwriting one module with another module with the same functionality. The following highlights the different subtopics of mapping covered.

6.1.1. Unconstrained Mapping

What will Xilinx® do if left on its own? The tool itself is an optimization tool and if given no area constraints, it can find the best implementation for a particular design. One main reason is to find other aspects that need controlling for further optimization. The mapping is done arbitrarily but only the same components are mapped from one design to the next. This will also be used as a baseline to compare to the other experiments.

6.1.2. Constrained Mapping

Constrained mapping takes into account the concept of floor planning – a process in which each module is given a location. Instead of giving no area constraints and letting the tool go, each component not only has a shape constraint but also has a placement constraint. The tool is then forced to re-optimize based on these new user-specified constraints. This set of experiments will show how floor planning affects bit reconfiguration minimization. The mapping and placement of the modules, like the first set of experiments, are also arbitrary.

If the interface of a module is altered, then the logic and routing of the module will also be altered. Even when a component is mapped, only the logic remains the same. Thus, to eliminate the difference in routing, wrappers are added to direct where the inputs and outputs will go. Due to the different components of the design, only the multipliers will have wrappers. The mappings and placements will be the same as the ones of the constrained mapping.

6.1.3. Reverse Mapping

In the final set of experiments, the roles of **Design 1** and **Design 2** are flipped – **Design 2** will act as the base design and **Design 1** is the new incoming task. The implementation will use constrained mapping and take interface into account. The purpose of this set of experiments is too see if there is an optimal order of designs if given a series of tasks to complete.

6.2. Implementation

The implementation of dataflow graphs all follow a general guideline, as shown by *Figure 15*. The first step is to pick out a specific dfg to implement. Once the graph has been chosen, it is then translated into code, either Verilog or Very high speed integrated circuit Hardware Description Language (VHDL). Sometimes, a user constraint file (.ucf) is added to supplement the code. All the code is then synthesized and implemented using Xilinx®.

The implementation involves three general steps: translation, mapping, and place and route (PAR). Specific functions of each step is explained in great detail in the Development Guide published by Xilinx® but is summarized in the following. The translation, or ngdbuild, step takes the synthesized code and converts it into a logical description of the design in terms of both the hierarchical components used and the lower-level Xilinx primitives. The output, an .ngd file, is then fed into the mapping stage, which maps the design to the target FPGA. The newly generated .ncd file is then fed into the PAR which, when completed, outputs another .ncd file, only this one is fully routed [3]. Finally, the .ncd file is sent through Bitgen which generates a .bit file and determines the number of bits and frames. Note that there are additional output files generated during the implementation stage but is omitted in this discussion.

During mapping and PAR, guide files can also be specified as to help specify the placement of logic components. This step is strictly optional and is only used when one module in one design is reused. It is possible to write a batch file to execute this entire process of synthesis and implementation by running Xilinx® Project Navigator in Command Line Mode. The specific commands for each step are included in the Development Guide.

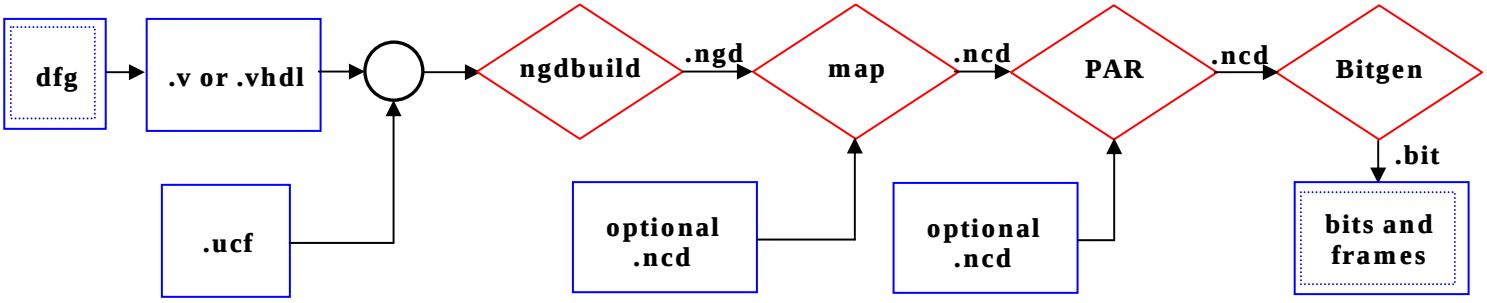


Figure 15: Flow Chart of Design Synthesis and Implementation Process

6.3. Unconstrained Mapping

Xilinx®, given only a timing constraint (no area constraints), removes the hierarchy of the individual components and find the best arrangement of the logic such that performance (time and/or space) is optimized. With the lack of design hierarchy, the logic slices are strewn about the entire board in such a way that each piece is optimized for real estate or overlap usage. This process is known as flattening. The other extreme of module placement is known as floor planning, a topic discussed in greater detail in a following section.

In this set of experiments, ***Design 1*** is implemented with no area constraints (and imposed with an average timing constraint) after which ***Design 2*** is mapped back to ***Design 1***. *Table 4* below shows the different mapping combinations. ***Design 2***, having nine multipliers, must leave one multiplier unmapped, as ***Design 1*** only has eight multipliers. This is considered full mapping. The number of mapped multipliers is then decreased incrementally; this generates partial mapping data. This continues until no multipliers are mapped, in which case is referred to as no mapping. In this particular case, when a multiplier is no longer mapped back to ***Design 1***, it is left unconstrained.

As these particular dataflow graphs involve only multipliers and adders, the multipliers are the components that are removed one by one. They are the dominant components and the results are more noticeable than if removing adders. In the following table, the multipliers listed in black are the original multipliers from ***Design 1*** while the ones in red are the ones found only in ***Design 2***. Notice that in the no mapping column (**o Mapped Multipliers**) all multipliers are colored red.

8 Mapped Multipliers	7 Mapped Multipliers	6 Mapped Multipliers	5 Mapped Multipliers	4 Mapped Multipliers	3 Mapped Multipliers	2 Mapped Multipliers	1 Mapped Multiplier	0 Mapped Multipliers
m_mult_17	*m_mul4100*	*m_mul4100*	*m_mul4100*	*m_mul4100*	*m_mul4100*	*m_mul4100*	*m_mul4100*	*m_mul4100*
m_mul4102	*m_mul4102*	*m_mul4102*	*m_mul4102*	*m_mul4102*	*m_mul4102*	*m_mul4102*	*m_mul4102*	*m_mul4102*
m_mult_21	m_mult_21	m_mult_21	m_mult_21	m_mult_21	*m_mul4104*	*m_mul4104*	*m_mul4104*	*m_mul4104*
m_mult_24	m_mult_24	m_mult_24	m_mult_24	m_mult_24	m_mult_24	m_mult_24	m_mult_24	*m_mul4107*
m_add_11	m_add_11	m_add_11	m_add_11	m_add_11	m_add_11	m_add_11	m_add_11	m_add_11
m_mult_15	m_mult_15	m_mult_15	m_mult_15	*m_mul4140*	*m_mul4140*	*m_mul4140*	*m_mul4140*	*m_mul4140*
m_add_20	m_add_20	m_add_20	m_add_20	m_add_20	m_add_20	m_add_20	m_add_20	m_add_20
m_mult_22	m_mult_22	m_mult_22	m_mult_22	m_mult_22	m_mult_22	m_mult_22	*m_mul4172*	*m_mul4172*
m_add_12	m_add_12	m_add_12	m_add_12	m_add_12	m_add_12	m_add_12	m_add_12	m_add_12
m_mult_16	m_mult_16	m_mult_16	*m_mul4184*	*m_mul4184*	*m_mul4184*	*m_mul4184*	*m_mul4184*	*m_mul4184*
m_add_14	m_add_14	m_add_14	m_add_14	m_add_14	m_add_14	m_add_14	m_add_14	m_add_14
m_mult_18	m_mult_18	*m_mul4205*	*m_mul4205*	*m_mul4205*	*m_mul4205*	*m_mul4205*	*m_mul4205*	*m_mul4205*
m_add_13	m_add_13	m_add_13	m_add_13	m_add_13	m_add_13	m_add_13	m_add_13	m_add_13
m_mult_23	m_mult_23	m_mult_23	m_mult_23	m_mult_23	m_mult_23	*m_mul4216*	*m_mul4216*	*m_mul4216*
m_add_10	m_add_10	m_add_10	m_add_10	m_add_10	m_add_10	m_add_10	m_add_10	m_add_10

Table 4: Unconstrained Full and Partial Mapping from Design 2 to Design 1

Furthermore, all mapping selections are purely arbitrary, including the selections for the adders. The reasoning is as follows: the tool is given almost complete freedom to optimize and should not be limited to specific mapping choices. It is also unreasonable to do an exhaustive compilation of all the different mapping possibilities. Thus, the mappings are chosen at random. The selection of which multiplier to not map is also chosen at random for similar reasoning.

The following table summarizes the results of the different mappings. As noted, the full mapping along with the upper partial mappings failed. This means that Xilinx® was unable to fit Design 2 using the groundwork laid out by Design 1. However, once enough multipliers were removed, the design was able to run successfully.

Mapping	# Frames [Frames]	# Bytes [Bytes]	Time (Actual) [ns]
8 Mapped Multipliers	*Design Failed*	*Design Failed*	*Design Failed*
7 Mapped Multipliers	*Design Failed*	*Design Failed*	*Design Failed*
6 Mapped Multipliers	*Design Failed*	*Design Failed*	*Design Failed*
5 Mapped Multipliers	3,203	610,570	14.000 (13.965)
4 Mapped Multipliers	3,144	567,734	14.000 (13.277)
3 Mapped Multipliers	3,510	610,702	14.00o (13.667)
2 Mapped Multipliers	3,675	556,494	14.000 (13.840)
1 Mapped Multiplier	3,973	667,886	14.000 (13.844)
0 Mapped Multipliers	3,969	671,222	14.000 (13.903)

Table 5: Experimental Data for Full and Partial Mapping

The timing constraint given to this set is not the tightest bound of 12 ns but slightly loosened to that of 14 ns. If too tight of a timing constraint is imposed, the tool may not find a working design. Thus, the timing is relaxed enough such that the tool has some leeway but not enough such that performance is greatly compromised.

The graph below visually displays the number of frames needed per each mapping. Although that the later points do not exist, a mathematical inference can still be made. The trend shows a negative linear relationship between the number of mapped multipliers and the number of frames. The worst result is when no multipliers are mapped back, taking up around 4,000 frames; the best result is when all eight multipliers are mapped back, using an estimated 2,600 frames (in theory). However, based on the result given in *Table 5*, it is clearly not possible using unconstrained mapping.

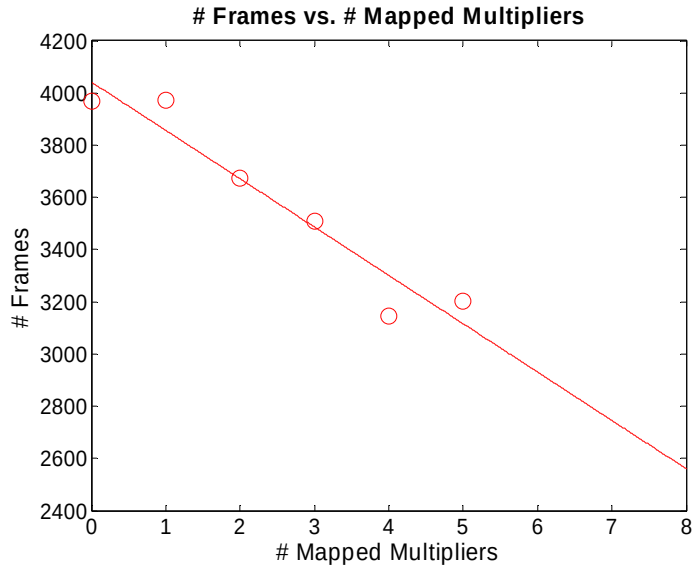


Figure 16: Number of Frames Needed vs. Number of Mapped Multipliers for Unconstrained Mapping

The relationships pertaining to the number bytes needed, however, is not as clear cut as it is for the number of frames. The data points are scattered about with no clear relationship between the number of mapped multipliers and the number of bytes needed.

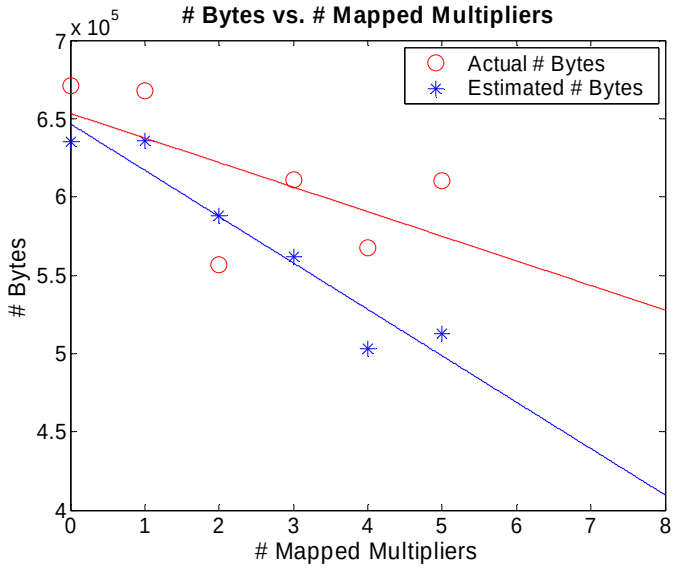
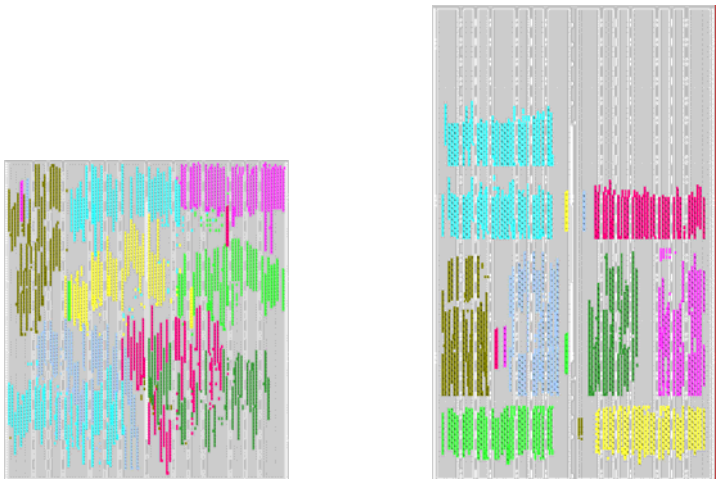


Figure 17: Number of Bytes Needed vs. Number of Mapped Multipliers for Unconstrained Mapping

There is a negative linear relationship between the number of bits and the number of mapped multipliers but the correlation coefficient is low, around 60%. Although it is not clear as to how the mapping specifically affects the number of bytes, it is clear that the more multipliers mapped back, the greater the amount of bit savings.

6.4. Constrained Mapping

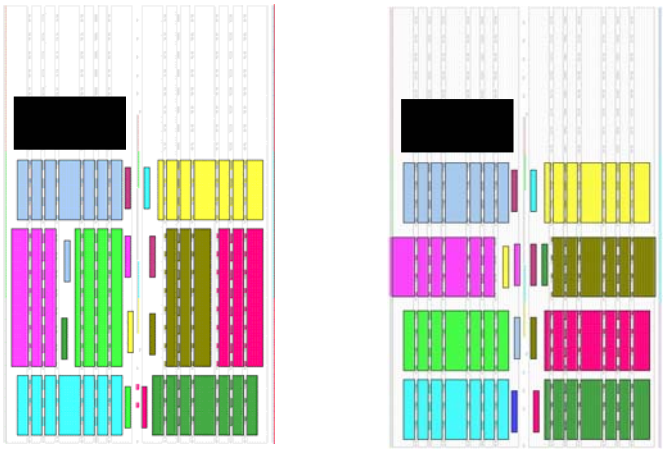
With unconstrained mapping, the design or task is flattened – the hierarchy of each individual component is lost. However, with constrained mapping, not only is the overall hierarchy preserved but also each component has a specific location on the FPGA board. This strategy of constrained mapping comes with both a major pro and con. The advantage is that the designer now has more control over module placement. This leads to an increase in predictability – the variations in results can be mapped back to the modifications of the design alterations. The disadvantage is board usage, as shown in *Figure 18*. Pictured above is the same design except the left is unconstrained while the right had placement constraints. Note that this design, when constrained, takes up more room than the unconstrained version. Note that the top multiplier (turquoise) in the constrained version cannot fit in the bottom half if kept in tact but can fit if separated.



Unconstrained ***Constrained***

Figure 18: Two Different Implementation of Design 2

The other major concern lies within the physical location of each logic component. The placements of each module does affect whether the mapping's success. Take the following example of two different module arrangements of Design 2. The placement of the left picture yielded a successful test run while the one on the right did not. As with any other arrangement, each module must have enough resources both for itself and for its connections to other modules. The failure of the design could be due to lack of available resources – it is possible that the modules that were connected in the dfg could not reach one another due to distance or routing congestion.



Successful ***Unsuccessful***

Figure 19: Two Different Placements of Design 2

The facts that each module has a strictly set location and that the logic of the components is non-sharable leads to a less efficient use of both board space and resource. However, by relaxing area minimization requirements, the design is able to run, even with a complete full mapping from ***Design 2*** to ***Design 1***. The table below shows the different full mappings. Note that every single trial succeeded due to the use of proper placement and area constraints. The unmapped (but still constrained) multiplier in ***Design 2*** is shown in red.

Design 2	Mapping 1	Mapping 2	Mapping 3	Mapping 4	Mapping 5	Mapping 6	Mapping 7	Mapping 8
mul4100	mult_16	mult_15	mult_24	mult_18	mult_16	mult_17	mult_21	*mul4100*
mul4102	mult_15	mult_16	mult_23	mult_23	mult_24	*mul4102*	mult_15	mult_18
mul4104	mult_21	mult_17	mult_22	mult_24	mult_18	mult_21	*mul4104*	mult_15
mul4107	*mul4107*	mult_18	mult_21	mult_15	*mul4107*	mult_24	mult_23	mult_17
add_sub4114	add_10	add_9	add_26	add_9	add_12	add_11	add_10	add_13
mul4140	mult_22	mult_21	mult_18	*mul4140*	mult_22	mult_15	mult_16	mult_21
add4141	add_20	add_10	add_25	add_11	add_25	add_20	add_13	add_14
mul4172	mult_17	mult_22	mult_17	mult_21	mult_23	mult_22	mult_24	mult_16
add4173	add_19	add_11	add_20	add_26	add_13	add_12	add_25	add_20
mul4184	mult_23	mult_23	mult_16	mult_17	mult_21	mult_16	mult_18	mult_24
add_sub4185	add_25	add_12	add_19	add_20	add_10	add_14	add_9	add_12
mul4205	mult_18	mult_24	mult_15	mult_16	mult_17	mult_18	mult_22	mult_23
add_sub4206	add_11	add_13	add_14	add_12	add_9	add_13	add_20	add_10
mul4216	mult_24	*mul4216*	*mul4216*	mult_22	mult_15	mult_23	mult_17	mult_22
add4217	add_26	add_14	add_13	add_25	add_14	add_10	add_12	add_9

Table 6: Different Constrained Full Mappings of Design 2 onto Design 1

6.4.1. Full Mapping without Wrappers

Using the full mappings given, the table below lists the results of each trial. Each mapping uses the successful arrangement as presented in *Figure 19*. Note that for these particular set of experiments, the modules do not have any wrappers around them. There are two reasons for doing so, the first being that if the entire module is mapped, then there should not be any difference in the modules from one design to the other. The second reason is that in the case that there is a difference, these numbers will be used as a baseline for comparison purposes.

Mapping Without Wrappers	# Frames [Frames]	# Bytes [Bytes]	Time (Actual) [ns]
Design 2 (Unconstrained)	5,404	839,408	14.000 (14.000)
Mapping 1	3,250	598,848	14.000 (13.900)
Mapping 2	3,357	609,368	14.ooo (13.861)
Mapping 3	3,205	594,516	14.000 (13.917)
Mapping 4	3,269	594,244	14.000 (13.959)
Mapping 5	3,296	606,724	14.00o (13.936)
Mapping 6	3,253	597,752	14.000 (13.984)
Mapping 7	3,283	607,348	14.000 (13.965)
Mapping 8	3,284	594,749	14.000 (13.875)

Table 7: Experimental Data for Constrained Full Mapping

The timing constraint for these trials, like the unconstrained case, is relaxed slightly to give the tool more leeway when performing optimization algorithms. From the results, it is likely that the recorded times are not optimal, as they do not hit the constraint dead on, but they are close. To see the amount of savings earned by the use of full mapping, the first trial is the second design implemented with no constraints – the tool has every available freedom to perform its own optimization techniques.

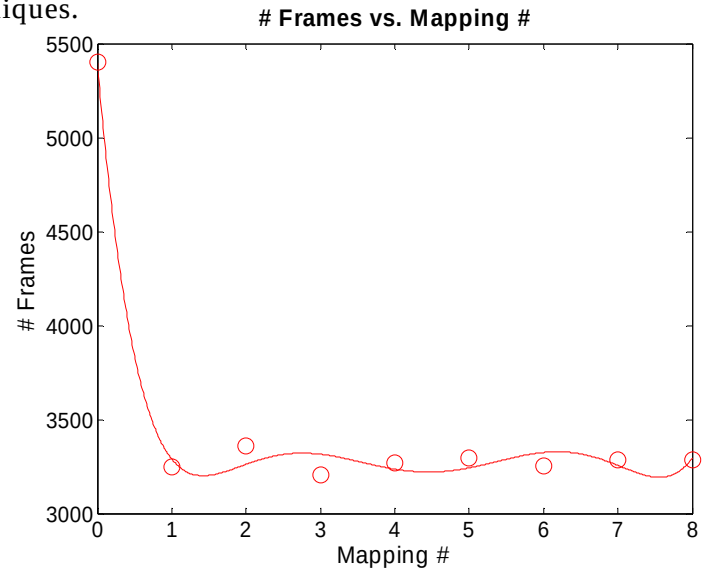


Figure 20: Number of Frames Needed vs. Various Constrained Mappings for Constrained Full Mapping

According to the above graph, the unconstrained version needs slightly less than 5,500 frames to reconfigure. However, compared to any other mapping arrangement, the number of frames drops roughly by 2,000. This drastic reduction in frames then reaffirms the importance of mapping and proper placement when moving from one design to another. Note, however, that

the mapping choices for the multipliers do not significantly affect the savings, as the results were relatively constant.

The graph below displays the actual number (red) and the estimated number (blue) of bytes needed. There is a substantial difference between the two, implying that there is a large overhead involved when doing partial reconfiguration. However, as the difference between each individual point is relatively constant throughout each of the different mappings, the relationship between frames and bytes can be easily modeled by a mathematical equation.

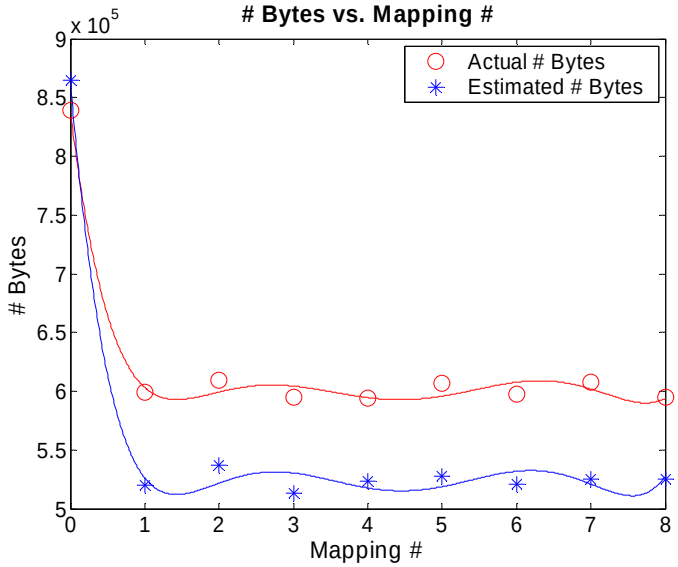


Figure 21: Number of Bytes Needed vs. Various Constrained Mappings for Constrained Full Mapping

6.4.2. Full Mapping with Wrappers

This section uses the exact same placements as the previous section except that both input and output wrappers are included for each multiplier mapped. The purpose of the wrappers is to further control the multipliers when switching from one design to the next. They are employed in order to limit the routing difference caused by the changing positions of the inputs and outputs. The wrappers are placed such that all I/O pins of the modules were facing toward the center of the board. Note that this configuration contradicts the previous statement made about the optimal arrangement of wrappers for multipliers that have a frame height of 1. The reasoning behind this is that all the wiring will meet at one general location and only that location will need to be reconfigured. The table below lists the wrapper results.

Mapping With Wrappers	# Frames [Frames]	# Bytes [Bytes]	Time (Actual) [ns]
Design 2 (Unconstrained)	5,130	760,793	14.000 (13.930)
Mapping 1	1,955	380,565	14.000 (13.095)
Mapping 2	1,983	379,417	14.ooo (13.580)
Mapping 3	1,920	365,837	14.000 (13.753)
Mapping 4	1,972	386,277	14.000 (13.559)
Mapping 5	1,965	375,981	14.00o (13.525)
Mapping 6	1,971	377,725	14.000 (13.095)
Mapping 7	1,977	383,849	14.000 (13.095)
Mapping 8	1,992	385,594	14.000 (13.095)

Table 8: Experimental Data for Full Mapping plus Wrappers

The unconstrained value, having a base value of around 5,000 frames, is significantly higher than any other mapping arrangement. When using wrappers, the number of frames needed for reconfiguration dropped more significantly than before. This further reinforces the effectiveness of the wrapper – they give the design more stability when switching from one task configuration to another.

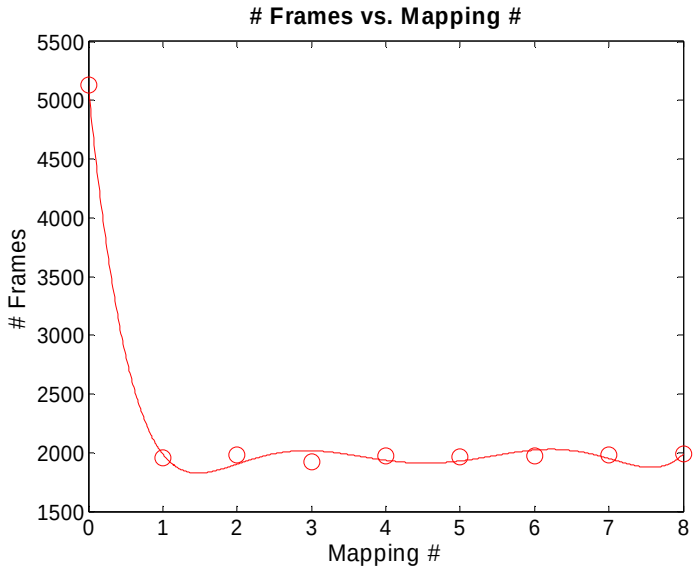


Figure 22: Number of Frames Needed vs. Various Mappings for Full Mapping with Wrappers

Recall that when wrappers were not used, it took roughly 3,500 frames to reconfigure. Note that each data point settles around 2,000 frames. This further reduction of 1,500 frames shows that to have a great result, the dominant modules need wrappers to limit the routing change. From the original 5,500 frames, this model of using area constraints with wrappers has a savings of roughly 3,500 frames, which is a reduction of over 50%.

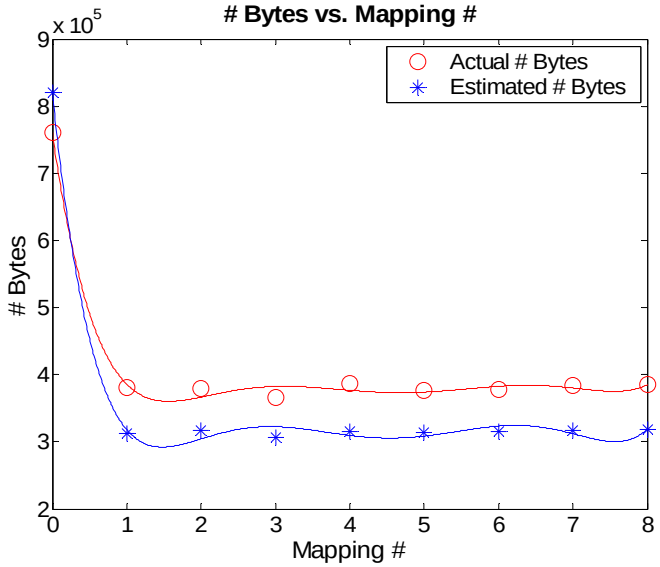


Figure 23: Number of Bytes Needed vs. Various Mappings for Full Mapping plus Wrappers

The number of bytes needed has also been reduced by a sizeable amount, going from around 60,000 bytes to 40,000 bytes, a drop of roughly a third of the total amount. Note that the overhead amount is also lessened. As there are fewer frames to reconfigure, the need to store any extra addresses is not as pressing as before.

6.5. Reverse Mapping

As demonstrated in the previous sections, not only is the placement of the modules important but also the interface needs to be fixed along with the module itself. The final set of experiments tests if there is a preference between mapping ***Design 2*** onto ***Design 1*** or vice versa. The results are based on the methodology described in the previous sections. Recall that ***Design 1*** has one fewer multiplier than ***Design 2***, so for this set of mappings, all multipliers from ***Design 1*** will be mapped back to that of ***Design 2***'s with one random multiplier omitted.

Design 1	Mapping 1	Mapping 2	Mapping 3	Mapping 4	Mapping 5	Mapping 6	Mapping 7	Mapping 8
add_9	*add_9*	add_sub4114	add_sub4185	add4173	add_sub4114	add4141	*add_9*	*add_9*
add_10	add_sub4114	*add_10*	add4141	*add_10*	add_sub4206	*add_10*	add4173	*add_10*
add_11	*add_11*	add_sub4185	add_sub4206	add4217	*add_11*	add_sub4185	add4217	add4173
add_12	add4141	*add_12*	*add_12*	add_sub4114	*add_12*	*add_12*	*add_12*	add_sub4206
add_13	add_sub4206	add4217	add_sub4114	*add_13*	*add_13*	*add_13*	add_sub4114	add4141
add_14	*add_14*	add4141	*add_14*	*add_14*	add_sub4185	add4173	add_sub4206	*add_14*
mult_15	mul4100	mul4216	mul4102	mul4104	mul4140	mul4100	mul4172	mul4205
mult_16	mul4216	mul4107	mul4107	mul4102	mul4100	mul4205	mul4140	mul4216
mult_17	mul4104	mul4184	mul4140	mul4107	mul4102	mul4216	mul4104	mul4172
mult_18	mul4205	mul4100	mul4184	mul4100	mul4104	mul4102	mul4100	mul4107
add_19	add4173	*add_19*	add4173	add_sub4206	add4141	*add_19*	add_sub4185	*add_19*
add_20	*add_20*	add_sub4206	*add_20*	*add_20*	add4173	add4217	*add_20*	add4217
mult_21	mul4172	mul4104	mul4104	mul4140	mul4205	mul4172	mul4102	mul4184
mult_22	mul4107	mul4102	mul4216	mul4216	mul4216	mul4107	mul4184	mul4140
mult_23	mul4140	mul4205	mul4172	mul4205	mul4184	mul4140	mul4216	mul4102
mult_24	mul4184	mul4172	mul4205	mul4184	mul4172	mul4104	mul4107	mul4100
add_25	add_sub4185	*add_25*	add4217	add_sub4185	*add_25*	add_sub4206	add4141	add_sub4114
add_26	add4217	add4173	*add_26*	add4141	add4217	add_sub4114	*add_26*	add_sub4185

Table 9: Different Mappings from Design 1 onto Design 2

The results given below are done using implementations using area constraints with wrappers. Note the actual times relative to the timing constraint is much looser than before, implying that there is room for optimization.

Reverse Mapping	# Frames [Frames]	# Bytes [Bytes]	Time (Actual) [ns]
Design 1 (Unconstrained)	4,930	660,852	14.000 (12.811)
Mapping 1	1,928	327,226	14.000 (13.468)
Mapping 2	1,949	327,118	14.000 (13.468)
Mapping 3	1,938	339,462	14.000 (13.468)
Mapping 4	1,773	303,246	14.000 (13.468)
Mapping 5	1,903	317,534	14.000 (13.468)
Mapping 6	1,962	335,466	14.000 (13.468)
Mapping 7	1,791	301,578	14.000 (13.175)
Mapping 8	1,797	323,479	14.000 (13.468)

Table 10: Experimental Data for Reverse Mapping

When the design mapping is reversed, the total number of frames needed falls around 5,000. However, when constrained and using wrappers, ***Design 1*** only needs around 2,000 frames to reconfigure. This drastic drop in frames is similar to the results obtained during regular mapping. However, there is more variation between the different mappings. As there is a multiplier missing from the first design to the second, some of the mapping arrangements make better, more efficient use of the board than others. As these mappings use up less real estate, they require fewer frames (and thus fewer bytes) to reconfigure.

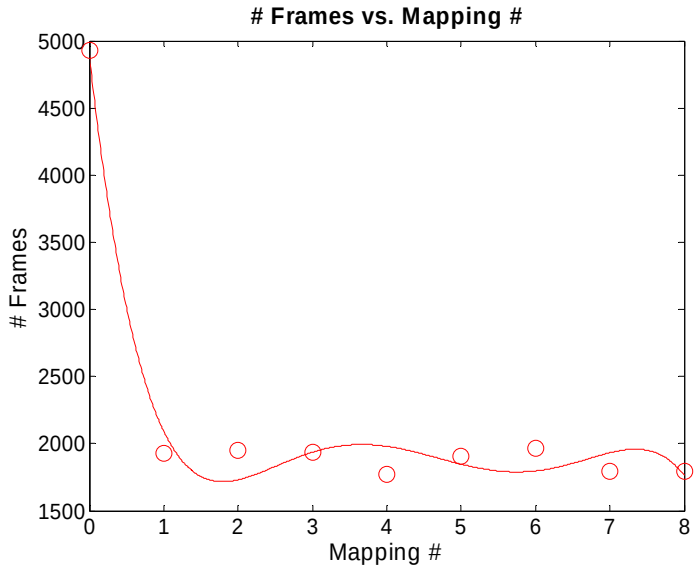


Figure 24: Number of Frames Needed vs. Various Mappings for Reverse Mapping

In *Figure 25*, note the difference between the estimated number of bytes (blue) and the actual number of bytes (red). Compared to the normal mapping, the overhead needed for this design is significantly smaller. Note that the difference is minute and that the overhead needed is substantially less than the overhead needed for the previous design. This significant difference in results implies that the order in which the tasks are reconfigured and mapped plays an important role in optimization.

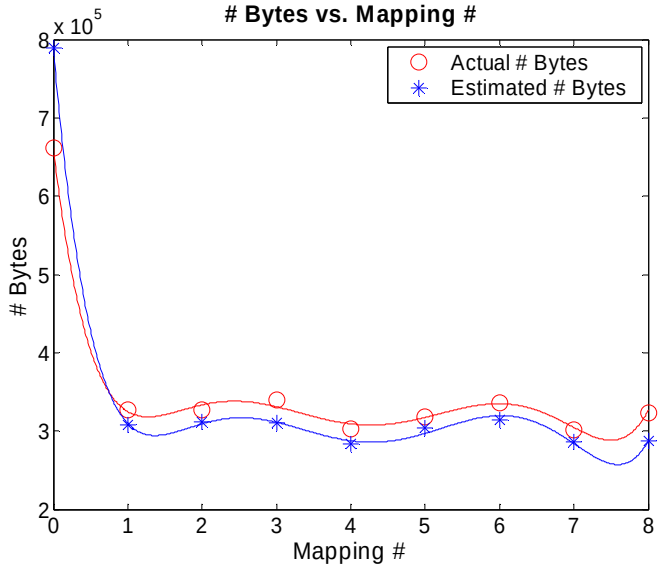


Figure 25: Number of Bytes Needed vs. Various Mappings for Reverse Mapping

7. Future Work

The research reported in this paper has dealt with some of the fundamental issues that are in involved in the minimization of bit usage during partial reconfiguration. Using this as the base knowledge, there is a plethora of different venues to explore. Two of these topics, placement logistics and systematic layout design, are discussed in greater detail below.

The first possible extension lies in the logistics of module placement. As noted earlier in this paper, the placement of the modules within the FPGA board plays a fundamental role in determining the successful implementation of a design. However, the exact reasoning as to why one placement arrangement works and another arrangement fail is still unknown. A variety of reasons, including wire congestion, I/O problems, and module locations, attributes to a higher probability of failure but which setback is the most deadly? Not only do the causes of failure need to be known but also the severity.

Another option is to come up with an algorithm to systematically determine the optimum floor plan of each incoming task if given a sequence of dataflow graphs. The algorithm will start by taking each dfg and finding module arrangements (each related to each other) such that bit reconfiguration is minimized. Building upon the first topic, not only does each arrangement yield the smallest change bit-wise but the design must also run given the previous task's layout.

8. Conclusion

The introduction of partial reconfiguration has immensely increased both the flexibility and versatility of FPGAs. Now with the ability to dynamically change its course and to program on a new task, FPGAs can now accommodate even the most complex schedules. To understand and figure out the benefits and uses of partial reconfiguration then is a matter of great importance.

The optimization of bit minimization must be taken in steps, as the behavior of each individual component must be understood before the applying it to a larger system. This paper has demonstrated that the different constraints and considerations all interrelate to one another. Floor planning, for example, not only affects the minimization of real estate but also whether or not the design is executable. Although listed above are only two future work topics, there is no limit of potentially interesting problems that relate to FPGA partial reconfiguration.

9. References

[1] I. Kennedy. “Exploiting Redundancy to Speedup Reconfiguration of an FPGA”. In *Lecture Notes in Computer Science*, Vol. 2778, pg. 10, 2003

[2] M. Rullmann, S. Siegel, and R. Merker. “Optimization of Reconfiguration Overhead by Algorithmic Transformations and Hardware Matching”. In *19th IEEE International Parallel and Distributed Processing Symposium* (IPDPS ’05), 2005.

[3] Xilinx. “Development System Reference Guide”. *http://support.xilinx.com*.