Optimizing dataflow graphs with Packetarc

Erik Danielsson

D14, Lund University, Sweden

dat14eda@student.lu.se

Lasse Heemann

D14, Lund University, Sweden

dat14lhe@student.lu.se

Abstract

Packet Architects have created a toolchain to synthesize their high level language PAC into pipelined RTL code which is implemented in network switch hardware. Within the toolchain the language is described in a data flow graph which is stored in the dot format. This project applies optimizations from compiler technology to these data flow graphs in order assess which types of optimization have the potential to decrease costs. Five potential optimizations, Equals_1, Bitwidth Analysis, Remove Duplicates, Constant merging, and Tree Height Reduction are implemented and assessed.

1 Introduction

Routers and switches need to be able to process large amounts of information in the form of data packages which need a destination assigned. Building the hardware to handle these problems is very complex, especially when more efficient smaller solutions is always preferable. Packet Architects is a company that continuously works with tackling this problem. They have developed their own high level language, which they call PAC, as well as a synthesis tool chain that produces pipelined RTL code for hardware implementation.

Before an output for a hardware implementation is generated a data flow graph is created, representing the paths and the operations that will take place. Although the tool is highly efficient, there are still optimizations to be made that could help imptove the implementation. There are many ways to optimize a data flow graph. A lot of work has been done on the subject because it is relevant for multiple parts of the computer science domain.

The data flow graph is represented as a dot file, which can be shown as nodes and directed edges. This dot file is then processed by inserting operations into pipeline steps which will depend on the desired hardware specification. Any information that needs to be saved for future use after a pipeline step has been completed is saved in what is called a "flip-flop". The hardware can therefore be improved by finding representations that either require fewer operations or less flip-flops. The system can also be improved by increasing the amount of operations that can be computed in parallel, resulting in a lower amount of pipeline steps needed. In order to create a better implementation, multiple optimizations have been implemented, as well as evaluated to help packet architects decide what they could focus on to make their product even better. This paper will feature the method used to implement these optimizations as well as the results of the evaluations. First, the framework built to write optimizations will be presented, then the optimizations will be described, and finally a discussion about the results and conclusions will take place.

2 Framework

The project starting point was to operate on a graph file and output a graph file using the dot format [1]. No further tools were specified, which granted much freedom. The choice was made to write a program in the python [3] programming language for the project. In order to work with the graph files, they needed to be parsed into memory so that python could work with them. Due to a few reasons, this task took longer than initially expected.

Firstly the pydot [2] library was used with the idea that this would simplify parsing the input files and writing them according to the dot standard. Each node in a dot file has a list of attributes attached, which for example specifies the color and shape of the node. A problem occurred because the output files needed to be read by the pacopt program written by Packet Architecs. Since the dot language does not specify the order of attributes, they are in a different order in pydot output versus what pacopt expects. Therefore the graphs could not be written to a file using pydot, and this needed to be rewritten manually. At this point, some dependencies on the pydot library were in place so that quite a lot of code needed to be rewritten. In the end, the input is still parsed using pydot.

Course paper, EDAN70, Lund University, Sweden
January 21, 2019.

3 Benchmark

Packet Architects provided a version of their program pacopt for use in this project. Pacopt splits the graph into pipeline stages by taking a graph as input, and returning a graph as output. Furthermore, pacopt presents four statistics about the resulting graph:

- pipeline stages
- critical path delay
- area
- critical path registers

The task for this project was to decrease the area as much as possible. Area is a measure of how many hardware components are necessary to implement the graph. Indirectly, this translates to the cost of production for a given graph.

Packet Architects also provided four example graph files. Benchmarks were performed by optimizing the example files and then comparing the pacopt area calculation for the optimized file against the pacopt area calculation for the original file.

3.1 Problems

During the benchmarks, some problems were encountered. The order in which nodes appeared in the graph files should have been irrelevant, but it affected pacopt's output to a large degree. The results where up to one percent wrong on the area calculation. In order to mitigate this problem, all graphs were sorted before benchmarking. This removed the problem entirely for the equivalent graphs since the sorted files became equivalent. However, this does not mean that the optimized graphs are correctly assessed.

4 Optimizations

4.1 Equals 1

The first optimization which was implemented was 'Equals 1'. Equals 1 is a very simple optimization with the main purpose to test the rest of the program. The optimization was found by looking at the example file `tag_6x10G.dot`. Equals 1 finds comparison operations where 1 bit is compared to a constant 0 or 1. The comparisons are obviously unnecessary and therefore removed, as shown in figure 1 and figure 2. The optimization could be extended to multiple bits although comparisons with 111 or 000 are uncommon.

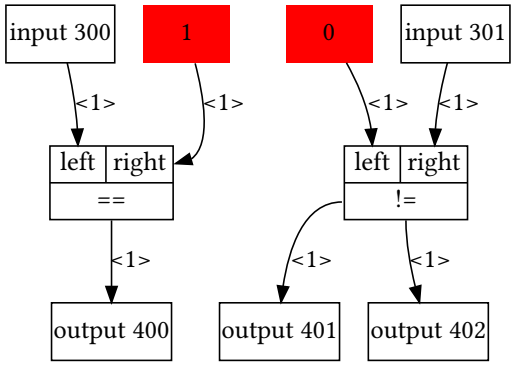

Figure 1. Equals 1 - input

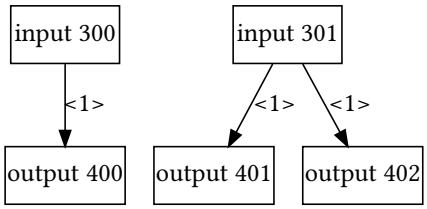

Figure 2. Equals 1 - output

4.2 Bitwidth Analysis

Bitwidth Analysis ensures that no unnecessary bits are stored or calculated upon. For example an ADD of two numbers with bitwidth 6, can result in at most 7 bits output. Operations with no overflow like OR can save another bit. With this in mind it is unnecessary to do a costly calculation with 20 bits when the first 13 bits will be 0 by definition. Furthermore if the following calculation previously had gotten 20 bits as input and now only gets 7 there is potential for this calculation to decrease in cost as well. A similar effect can be achieved in reverse. If an output value only consumes the last 3 bits of a calculation it is again unnecessary to do the previous calculations with more than 3 bits.

Since each decrease in bit width reduces the necessary bits for previous and following operations, the reductions ripple through the graph and decrease overall cost by a large margin. However this is not the only effect bitwidth analysis has on the cost. Recall that pacopt splits the graph into separate pipeline stages, and that the pipeline stages are connected with flip-flops. Each bit that is carried from one pipeline stage to the next requires a flip-flop to store the value. Decreasing the bitwidth wherever possible leads to the effect that inserting pipeline stages becomes cheaper as well.

In the figures 3 and 4 the optimization is show on an example graph. The OR operation with 8 and 6 bits as input can at most output 8 bits, so that the two outgoing edges are reduced to 8 bits. Since output 400 only consumes 3 bits the ADD operation can be reduced to 3 bit width, the input edges of the ADD are therefore reduced to 3 bits. At this point the OR would be reevaluated since its output has changed, but no change will occur since output 402 still requires 8 bits. If now the graph would be separated into two pipeline stages with OR in one and ADD in the next, then previously 20 and 11 bits are now reduced to 8 and 3. Noteworthy is that all operations pad the input with zeroes when the input sizes are different.

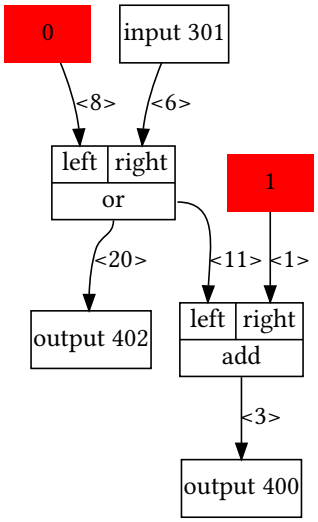

Figure 3. Bitwidth - input

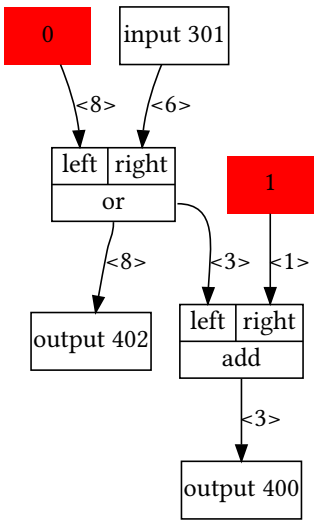

Figure 4. Bitwidth - output

4.3 Remove Duplicates

Remove Duplicates is a simple optimization focusing on finding instances where the outputs of two operations will always be equal. In such cases the outputs of one of the operands can safely be moved to the other operand and the operand can thereafter be removed. Figure 5 shows such a case where two ADD operations have the very same input, computing the same information twice is obviously ineffective. This optimization only finds duplicates with the same inputs, which is a relatively naive approach. More instances could be found by more sophisticated algorithms, as there are ways for operands to have the same output without having the same input. For example if two operations have different outputs that will always be equal, 2+3 and 3+2. This would be solved if something like constant propagation were ever to be implemented.

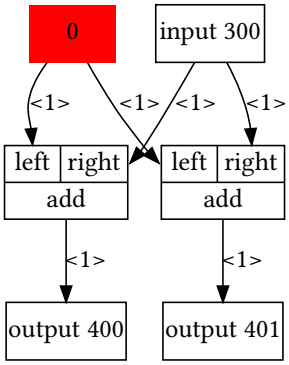

Figure 5. Remove Duplicates - input

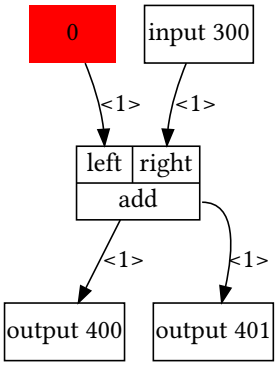

Figure 6. Remove Duplicates - output

4.4 Constant Merging

Constant Merging is a simpler version of constant propagation. The algorithm only looks at two subsequent arithmetic

nodes and determines whether they could be combined into one. Interesting cases were found in `tag_6x10G.dot`, where an addition with 1 was followed by a subtraction of 1 on multiple occasions. The algorithm removes both the addition and the subtraction in this case, as seen on the path between input 301 and output 405 in figures 7 and 8. During merging the bit width of operations might change, at this point in the project a simple worst case solution was implemented so that overflow can never occur. The bitwidth is simply changed to the largest necessary value which is shown in the path from input 300 to output 403 in figures 7 and 8. The choice was made to overestimate costs rather than underestimate.

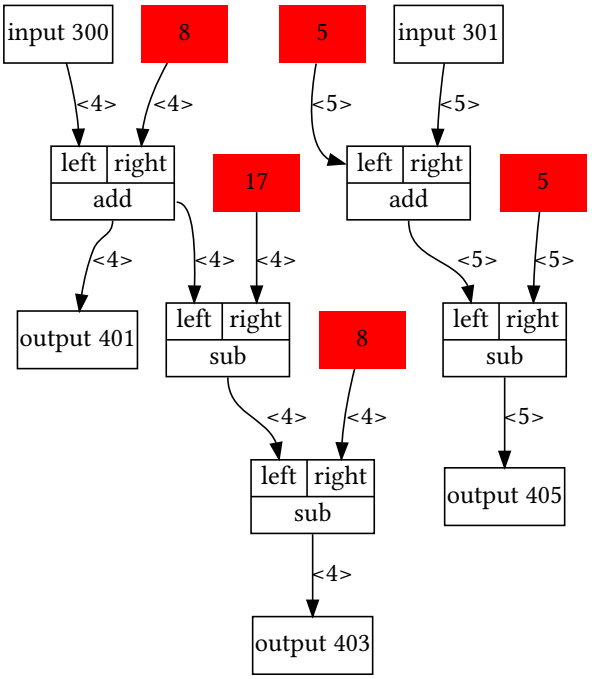

Figure 7. Constant Merging - input

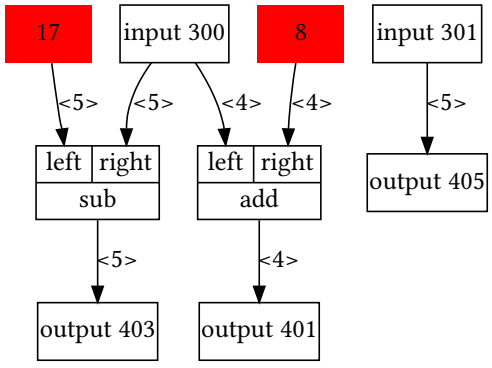

Figure 8. Constant Merging - output

4.5 Tree Height Reduction

In the big dataflow graph multiple subgraphs of similar operations can be found. Such as a series of ADD or SUB operations. These often occur in an unbalanced fashion of additions followed by additions, creating a long line of operations such as in figure 9. The subgraphs are similar to the datastructure called a tree, where the "height" of the tree is equal to the longest distance from its root to one of its leaves. In the case of Figure 9 this height would be 3. This tree height is important for multiple reasons, as it can directly affect the time it takes for each operation to be executed. A balanced version of the subgraph can be seen in Figure 10, where the tree height is only 2. This tree could be executed faster as two ADD operations could be evaluated simultaneously, as opposed to the tree in figure 9 where the ADD operations must wait for the previous operations to be executed. This reduction can be especially important if the subgraph is located on the critical path of the data flow program, as it will directly shorten the execution of the entire program. A more balanced tree can also be beneficial as more information can be processed in the same pipeline step, requiring less time and fewer flip flops.

A notable issue with creating these optimizations is the number of bits being processed. It is well known that many basic operations such as ADD, SUB, MUL and DIV are commutative, i.e. that the order of the inputs can be swapped, 1+2=2=1. However this all changes when the number of bits are different, changing the inputs or the order of the operations might cause the output to be different because of where overflow occurs. Therefore only subgraphs with the same number of bits were processed. In order to decrease the risk of different bitwidth the Bitwidth optimization should be run before the tree height reduction optimization.

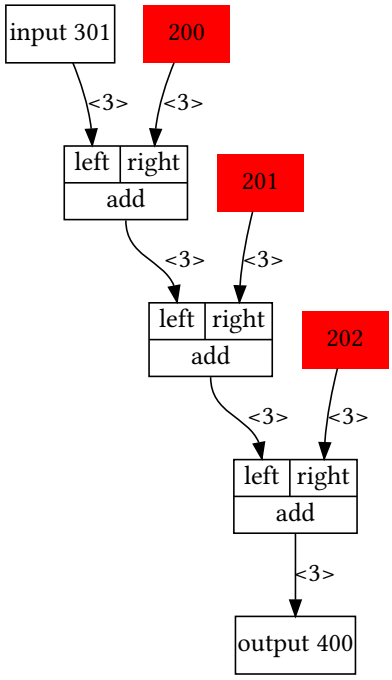

Figure 9. Tree Height Reduction - input

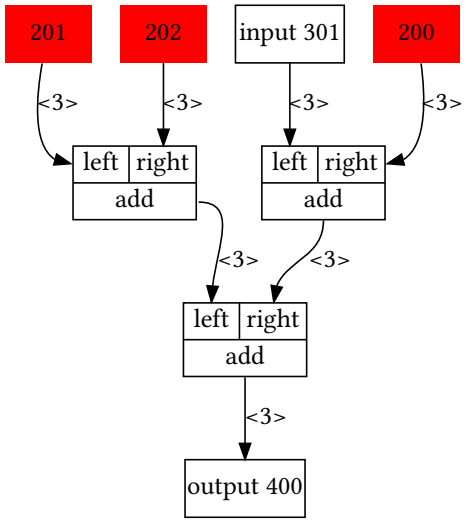

Figure 10. Tree Height Reduction - output

4.6 Unlikely Cases

There were also some optimizations that could prove useful, but during implementation could be found to not be relevant to the graphs this project was based upon. The files which were worked on were thought to be big enough that they would represent a typical program run, however it is possible that different files could have varying results. Two optimizations were looked into that proved to not be useful for the specific files.

4.6.1 Operator optimization

Some operations could be simplified by using bit shift rather than multiplication or divisions. A bitshift operation requires less computing power and would therefore be preferable. A requirement for this is that a multiplication or division by a power of two, something that could quickly be evaluated by computing the number of such instances for the given files. Only one instance was found, a multiplication of 4 that could be optimized by changing X*4 -> X«2. Implementing such an optimization would be quick but not beneficial due to the rarity of the case.

4.6.2 Algebaraic simplification

Some expressions can be optimized to include less operations. For example a*b+a*c can be changed to a*(b+c). But just as the operator optimization this was evaluated to not be relevant to these files.

5 Evaluation

The benchmark results for this project are presented in Figure 11. The values are calculated as the change in area cost divided by the non-optimized area cost. Meaning a 50 percent cost reduction would result in the value 0.5. Each column corresponds to one of the input files that Packet Architects provided whereas each row corresponds to an optimization. The last row presents the best result possible by combining the other optimizations.

The result for the opdelay_add column should not be seen as representative for a real system, since the graph is very small. The results for c1mepp and c1mipp are a much more relevant result.

Specially highlighted in red we see one occurrence of a negative value. This appears in several places and means that the optimization actually made the resulting graph worse. This is very questionable specifically for optimizations such as eq_1 where the graph contains fewer nodes and edges. As discussed in section 3.1, pacopt gives varying results when presented with the same data in a different format. These results should be viewed critically. If possible the data should be reevaluated with more knowledge about pacopt and maybe even a modified version of pacopt.

A further problem is the target delay value. The value can be anywhere inbetween 30 and 1000 according to Packet

Architects. It adds another variable of which the impact is unknown leading to even less trust in the results.

	target delay = 80			
	opdelay_add	tag_6x10G	c1mepp	c1mipp
eq_1	0,0000	0,0025	0,0014	0,0010
bitwidth	0,3085	0,0000	0,0007	0,0208
const_merging	0,5745	0,0160	0,0000	0,0000
remove_duplicates	0,0000	0,0000	----	----
bitwidth & tree height reduction	0,6277	0,0000	0,0011	−0,1320
all	0,7128	0,0025	----	----
best	0,6650	0,0185	0,0014	0,0218

	target delay = 30			
	opdelay_add	tag_6x10G	c1mepp	c1mipp
eq_1	0,0000	0,0010	−0,0346	
bitwidth	0,4438	0,0000	0,0003	
const_merging	0,3034	0,0061	0,0000	
remove_duplicates	0,0000	0,0020	----	
bitwidth & tree height reduction	0,4551	−0,0102	0,0078	
all	0,6685	−0,0093	----	
best	0,6433	0,0060	0,0080	

Figure 11. Benchmark results

6 Conclusion

The project was limited in its scope because program input and output were previously defined as graph files. This allowed for the project to be completed in a short time frame. On the other hand, this limited information about the user input, and somewhat limited information about output because it was given as a graph file. After the compiler has compiled the source code to a graph file, it becomes very hard to interpret what the user defined and what assumptions the compiler made. In the same way it would be relevant to the optimizations where pacopt intends to split the graph. Some optimizations might make sense when applied within a single pipeline stage only. Thus, more information about the process before the dot file was created could have allowed this project to explore additional options.

Overall, the results are too vague in comparison to the error margin to draw any certain conclusions. Out of the applied optimizations, bitwidth analysis gave the best improvement with a 2 percent cost reduction for the largest and most representative graph. There is noticeable difference in effectiveness for the same optimizations on different graphs. For each graph, different optimizations should be applied.

Acknowledgments

Packet architechts gave us the oppertunity of working with their system. They have been really helpful by providing a lot of useful information for us to work with, as well as a detailed description of the documents we have been working on. They also answered any questions we had regarding the project by encouraging a direct communication between us.

References

[1] Graphviz. The graphviz dot language. https://www.graphviz.org/doc/info/lang.html.
[2] Kalinowski, S. pydot library for parsing dot files into python. https://github.com/pydot/pydot.
[3] Python Software Foundation. The python programming language. https://www.python.org/.