Linear Placement for Static / Dynamic Reconfiguration in JBits

Vamsi Krishna Marreddy **Sharareh Noorbaloochi** **Kia Bazargan**

Department of Electrical and Computer Engineering
University of Minnesota, Minneapolis, MN 55455
kia@ece.umn.edu

ABSTRACT

Placement of functional units on an FPGA fabric is a challenging problem for runtime reconfigurable computing systems. We introduce the concept of physical contexts to greatly reduce the complexity of the placement and routing problems. We have implemented static and dynamic linear placement methods for expression trees placed in physical contexts. Our placement algorithms are implemented in the JBits environment, creating a layer of a hardware operating system for future reconfigurable computing systems.

1. INTRODUCTION

A truly dynamic reconfigurable computing system would be able to decide - on the fly - what computations to perform on the FPGA and what operations to run on the CPU device. Programming such systems would be a daunting task unless high levels of abstraction are provided to application developers. Hardware operating systems (HWOS) should be developed to handle automatic placement of required cores on the FPGA, reuse idle cores already on the chip, perform cache manager for the instantiated cores, and so on. Hauck et. al. have discussed some of these features in their works (e.g., [3]).

The input to the hardware operating system would be expressions (compound "instructions") to be evaluated on arrays of data. The HWOS will provide on the fly estimations on how fast, if at all, the expressions can be evaluated on the FPGA. The application will decide whether to perform the computations on hardware or software, and in case of hardware, the HWOS would automatically instantiate cores (e.g., adders and multipliers) and schedule data transfer to the datapaths.

Each layer of such a hardware operating system should be transparent to the higher levels. For example, the core cache manager does not have to know anything about the physical placement of the cores on the FPGA. It would ask for estimations on resource availability and issue insertion / deletion requests to the placement engine. The placement engine would decide on the location of the new cores. The details of routing the busses between cores are left to the routing engine. The focus of this work is developing fast algorithms for the placement engine in such a framework.

Callahan, et. al. [2] introduced a fast mapping and placement algorithm for datapaths, but they assumed that the FPGA chip has enough free space to allow for contiguous placement of the whole datapath. Bazargan, et. al., [1] introduced two-dimensional placement methods for datapaths, but their method has two drawbacks: 1) they do not plan for data transfer between external/internal memories and the datapaths on the FPGA, and 2) their placement runtimes become long as more cores are instantiated on the FPGA (in the order of 100µs).

In this work, we have implemented a linear placement method (similar to [2]) in JBits, where we try to minimize the delay of the datapath by minimizing critical wire lengths. Furthermore, we reuse already instantiated cores to reduce reconfiguration time, and also enable placement of a datapath even if no contiguous free space large enough to accommodate it is available.

2. PHYSICAL CONTEXTS

To ensure fast runtime placement of the expressions, we have to limit the search space and develop pre-defined rules on how the cores should be placed. We divide the FPGA chip into a number of horizontal strips, each with a fixed height, and input / output memory blocks that will be used as buffers for the data to be processed. The height of the context can vary, depending on the width of data to be processed by the contexts (e.g., a 16-bit context must be able to accommodate a 16-bit multiplier). The HWOS can determine how many of each type of contexts are needed by the applications. Figure 1-a shows an example of an FPGA fabric divided into two contexts. IM and OM are input and output memories. The data from input memories are transferred to individual "input registers" instantiated in the contexts. Data flows from left to right through the datapath pipeline and stored in the output memory. IM and OM blocks are similar to the block RAMs of Xilinx's Virtex chip [4].

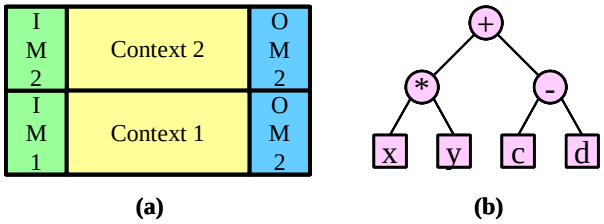

Figure 1 (a) FPGA divided into physical contexts (b) expression tree of xy+(c-d) to be mapped to a context

In addition to making the placement problem much simpler, the contexts provide opportunities for mapping independent threads of execution at the application or operating system level to the FPGA device. We assume that the assignment of the expressions to the contexts and scheduling of data transfer to and from the IM and OM memories are done at a higher level of the HWOS. We only focus on the placement of datapaths within contexts.

3. STATIC PLACEMENT

When an expression is given to the placement engine (our implementation takes the expression as a post-order expression), the placement engine calculates the total width of the datapath and if the context has a large enough contiguous free space, the datapath is placed in that empty region. We call this scheme static, as the placement of the expression can be arranged at compile time.

The static placement algorithm first calculates the delay at each node in the tree (maximum delay of its children, plus its node delay). The subtree with greater delay is placed closer to the root node. For example, in Figure 1-b, the left subtree has smaller delay, and hence having longer wires connecting its output to the root node is less likely to increase the overall delay of the whole datapath. So, the function cores placed from left to right in the context are going to be *, - and + respectively. The same technique can be applied to pipelined designs, but in this case we have to consider individual pipeline stage delays. Input and output registers are treated as cores that should be connected to the input/output memories.

4. DYNAMIC PLACEMENT

There are two problems with the static placement scheme: 1) there might not be enough contiguous free space in the context to place the datapath as one big chunk, and 2) there might be some idle cores left on the context from previous expressions. We can use these idle cores to reduce reconfiguration time. Our dynamic placement method addresses these issues by first generating the static placement, and then finding a set of idle modules that can be reused for the new expression. The busses connected to the idle cores are unrouted and rerouted to make connections to the new datapath.

The set of idle cores to be reused is selected in a way to minimize deviation from the static placement. Delay is allowed on busses connecting faster children to the root nodes of a subexpression. For example, in Figure 1-b, an idle subtractor that is far from the expression is more likely to be reused compared to an idle multiplier that cannot be connected to an adder close to it (either reused or instantiated).

5. EXPERIMENTAL RESULTS

We created a number of expressions for our placement tool and placed them in a context. Both static and dynamic placement methods were tested using the expressions shown in Table 1. Our methods were implemented in JBits and the datapaths were simulated using BoardScope. The column labeled “Inactive Cores” shows how many idle cores were in the context before the expression was placed. “Time” shows total reconfiguration time reported by JBits. Our placement algorithm runtimes were negligible (always reported as 0ms). With the exception of expression 4ii, reusing inactive cores reduces configuration time. Perhaps in 4ii unrouting and routing the signals took significant time resulting in the partial reconfiguration time being longer than the original configuration of the whole datapath. Figure 2 shows a snapshot of the cores of expression 5i placed statically, and Figure 3 shows the activity bits of the datapath simulation in BoardScope.

6. CONCLUSION AND FUTURE WORK

We presented a placement engine for dynamically reconfigurable systems. We plan to improve the dynamic placement method and our delay estimation process. We also plan to implement cache algorithms and integrate them with the placement engine into a mini HWOS.

7. REFERENCES

[1] Bazargan, K., and Sarrafzadeh, M., "Fast Online Placement for Reconfigurable Computing Systems", In *IEEE Symp. Field Programmable Custom Computing Machines*, pp. 300-302, 1999.

[2] Callahan, T. J., Chong, P., DeHon, A. and Wawrzynek, J., “Fast Module Mapping and Placement for datapaths in FPGAs”, in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 1998.

[3] Z. Li, K. Compton, S. Hauck, "Configuration Cache Management Techniques for FPGAs", FCCM’00, pp. 22-36.

[4] http://www.xilinx.com

Table 1 Reconfiguration times for test expressions. A – Adder, M – Multiplier, I – Input register, O – Output reg.

exp.No	Expression	Inactive Cores	Time (ms)
1 i	a + b	None	1336.0
2 i	(a * b) + (e * f)	None	2339.6
2 ii		1M, 1I	2129.0
3 i	(a * b) + (c + d + e)	None	2271.4
3 ii		2A, 3I, 1O	2205.0
3 iii		1A,1M,1I,1O	1895.0
4 i	((a * b) + (c * d)) * (n + m)	None	5389.8
4 ii		1A, 1M, 2I	5786.0
4 iii		1A, 2M, 4I	3507.2
5 i	(((a + b) * (c + d)) + ((e * f) + (k + g))) * (x + y)	None	5293.6
5 ii		2A, 1M, 3I	4666.8
5 iii		4A, 3M,6I,1O	3769.4

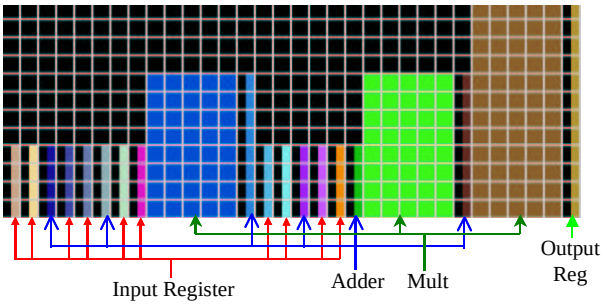

Figure 2 Snapshot of Jbit’s BoardScope showing the static placement of expression 5i (refer to Table 1).

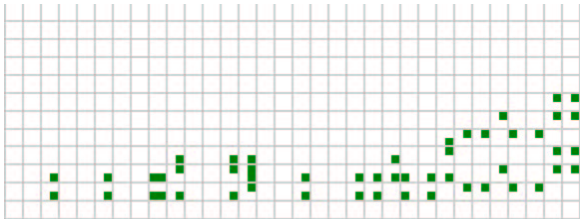

Figure 3 Activity bits of the placement of Figure 2 as shown in Jbit’s BoardScope. LUTs are recolored from blue to white to enhance visibility.