Pipeline Morphing and Virtual Pipelines

W. Luk, N. Shirazi, S.R. Guo and P.Y.K. Cheung

Department of Computing, Imperial College, 180 Queen's Gate,
London SW7 2BZ, UK

Abstract. Pipeline morphing is a simple but effective technique for reconfiguring pipelined FPGA designs at run time. By overlapping computation and reconfiguration, the latency associated with emptying and refilling a pipeline can be avoided. We show how morphing can be applied to linear and mesh pipelines at both word-level and bit-level, and explain how this method can be implemented using Xilinx 6200 FPGAs. We also present an approach using morphing to map a large virtual pipeline onto a small physical pipeline, and the trade-offs involved are discussed.

Introduction

Pipeline architectures are commonly used in high-performance designs. This paper introduces morphing, a technique for enhancing the efficiency of reconfigurable pipelines at run time. We shall also describe the use of morphing in the emulation of large virtual pipelines by small physical pipelines, and explain how temporary storage can be used to improve performance.

Implementing pipeline architectures using reconfigurable devices is attractive for several reasons. Many FPGAs facilitate the realisation of pipelines, since they have a regular structure and an abundance of registers. Moreover, leading-edge FPGAs often provide built-in support for fast reconfiguration. An example is the wildcarding facility for Xilinx 6200 devices [1], which allows concurrent reconfiguration of a block of FPGA cells. With such facilities, it is possible to reconfigure each pipeline stage rapidly at run time to implement multiple functions. For a system operating in an unpredictable environment, this possibility enables the selection of functions adaptively.

Partial reconfiguration [4] is a powerful method of exploiting the flexibility of FPGAs such as the Xilinx 6200: one part of the FPGA can be reconfigured while other parts are continuing to function. Pipelines provide a simple but effective scheme for partial reconfiguration, since pipeline registers isolate one pipeline stage from another so that computation and reconfiguration can take place at the same time without interference. The regular structure of pipelines also simplifies the development of hardware operators which can be relocated to different regions of a pipeline, maximising the re-use of design effort.

An obvious method for reconfiguring an n-stage pipeline involves three steps. First, one needs to complete the current computation and clear the pipeline, which takes n cycles. Then reconfiguration can take place. Finally one has to wait for n cycles for the result to flow through the newly-configured pipeline. This method of reconfiguring a pipeline leads to a latency of $2n$ cycles, in addition to the time for reconfiguring all the pipeline stages. In highly-pipelined systems when n is large, the pipeline latency cycles and reconfiguration time will have a significant impact on performance.

Pipeline Morphing

We present a method, called pipeline morphing, for reducing the latency involved in reconfiguring from one pipeline to another. The basic idea is to overlap computation and reconfiguration: the first few pipeline stages are being reconfigured to implement new functions so that data can start flowing into the newly-configured stages of the pipeline, while the rest of the pipeline stages are completing the current computation. Instead of changing the entire pipeline at once, our method involves morphing one pipeline to another, just as one can morph two images by interpolating them. In our case, the pipeline registers isolate one pipeline stage from its neighbours, enabling computation and reconfiguration to take place concurrently in different stages.

Figure 1 shows how a five-stage pipeline F can be morphed into a pipeline G in five steps. It should be clear from this example that during morphing, the flow of reconfiguration is synchronous with the flow of data, and hence the pipeline latency cycles are eliminated. If the time for reconfiguration is longer than the pipeline processing time, the pipeline will need to include flow control mechanisms to slow down the rate of data flow while morphing is taking place. We shall explain later how this can be achieved in Xilinx 6200 FPGAs.

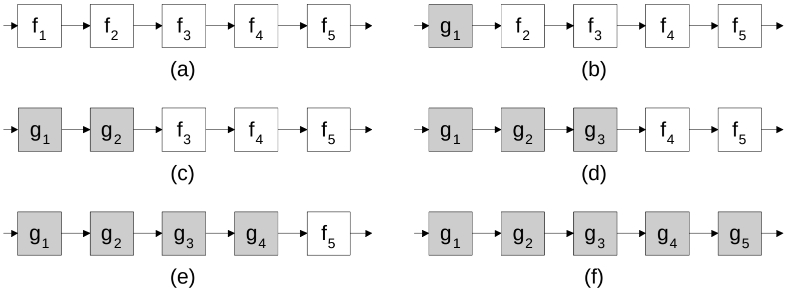

Fig. 1. Given the pipeline F with stages $f_1, \ldots, f_5$ and the pipeline G with stages $g_1, \ldots, g_5$, the diagram shows how F can be reconfigured into G in five steps using the morphing technique. Only one stage of the pipeline is being reconfigured in each step.

Whether morphing is used or not, a designer's task is to ensure that the slowing down due to run-time reconfiguration will not affect system performance. For instance in video processing, there may be sufficient time for reconfiguration between the end of one image frame and the beginning of the next frame.

Because of the elimination of latency cycles, pipeline morphing will improve the performance of systems that reconfigure at run time. It is particularly suitable for devices supporting rapid reconfiguration, and it works best when reconfiguration time is comparable to the pipeline computation time. To meet this condition, the user can build single-cycle reconfigurable structures in an FPGA [7]. FPGAs specially-designed for supporting rapid reconfiguration [10], [11] will particularly benefit from pipeline morphing. Morphing can also be applied to systems with multiple FPGAs arranged as a pipeline.

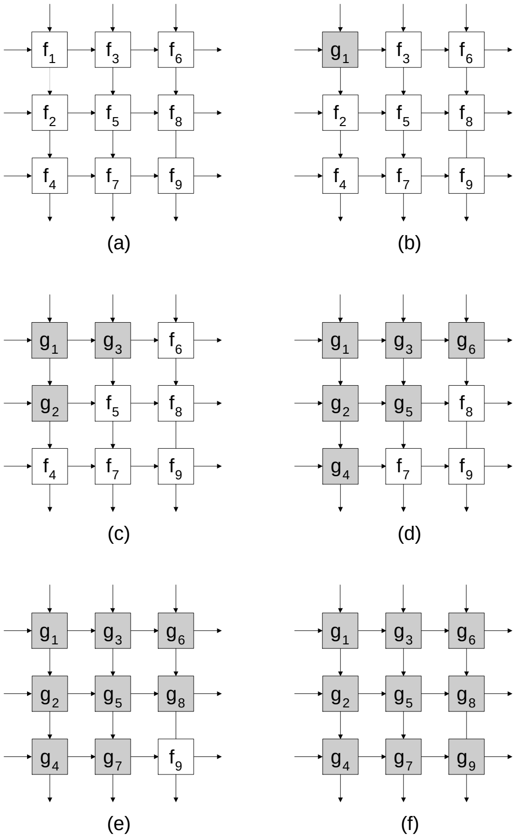

Fig. 2. Given the square mesh F with components $f_1, \ldots, f_9$ and the mesh G with stages $g_1, \ldots, g_9$, the diagram shows how F can be reconfigured into G in five steps using the morphing technique.

Our method is not confined to linear pipelines. It can be applied to pipelines of other shapes, such as two-dimensional meshes or tree-shaped designs. Figure 2 shows the steps of morphing from one mesh to another, given that every component in the mesh has a pipeline register at each of its two outputs. This approach can also be applied to bit-level operators. In the next section, we shall explain how a pipelined adder can be morphed into a pipelined subtractor.

Morphing Pipelines on Xilinx 6200 Devices

We illustrate the morphing reconfiguration technique by showing how it can be applied to reconfigure a six-bit, three-stage pipelined adder to a pipelined subtractor of the same size on a Xilinx 6200 FPGA. The pipelined adder is shown in Figure 3a and the resulting pipelined subtractor is shown in Figure 3d.

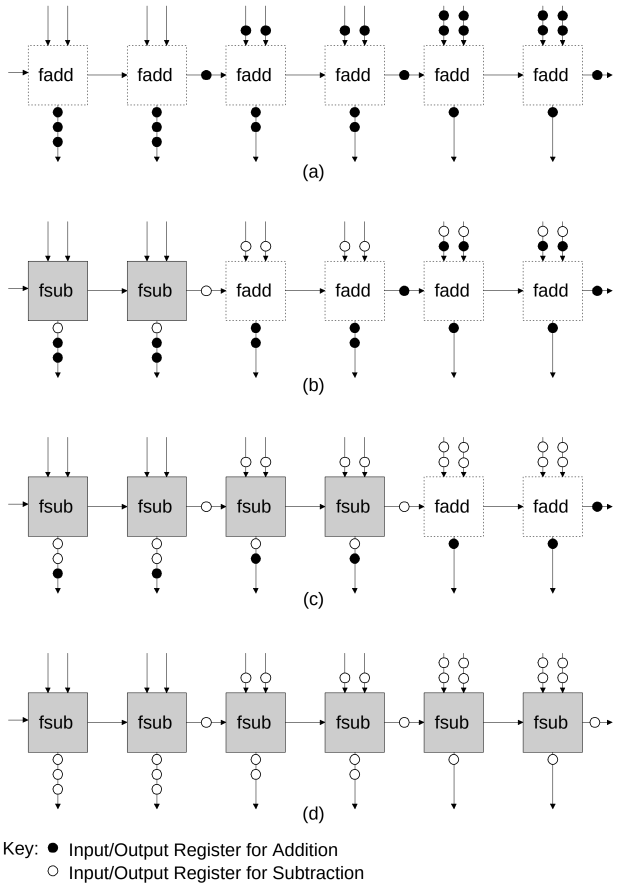
fadd
fsub
(a)
(b)
(c)
(d)
Key: ● Input/Output Register for Addition
○ Input/Output Register for Subtraction

Fig. 3. Morphing a pipelined adder to become a subtractor.

As explained above, if all n stages of a pipelined adder (n=3 in the above example) are reconfigured into a pipelined subtractor at once, an additional $2n$ clock cycles would be needed in order to flush the pipeline and to refill it. In order to avoid this delay, the reconfiguration is performed in three steps where each stage of the pipelined adder is reconfigured followed by one clock cycle of computation. These three reconfiguration steps are shown in Figures 3b, 3c and 3d. The partial configuration information involved in these steps was obtained using tools described in [8].

A dual clocking scheme is used in order not to clock invalid data values into the pipeline registers during reconfiguration. The two operand values for the adder are stored in two six-bit registers. When using the processor interface [1] on the Xilinx 6200 FPGA, a pulse is produced whenever a register is written with a value, and this pulse can be used as a clock for the registers within the design. The input registers are set up so that a clock pulse is generated when the second operand is written into the register. An additional configuration clock is used to control the reconfiguration of the logic in each pipeline stage. The reconfiguration sequence is therefore broken down into three steps. First, a stage of the pipeline is reconfigured; on completion the operands are written into the input registers; the write action then triggers the clock for the pipeline registers. This sequence is continued until all the stages are reconfigured.

In the above example, it takes four cycles to reconfigure the pipelined adder to a pipelined subtractor. Without morphing it takes an additional three cycles to flush the pipeline and three cycles to refill it; hence a total of ten cycles are needed to perform the reconfiguration and to begin producing correct results. The morphing technique therefore improves the reconfiguration time by 2.5 times; Table 1 summarises these results. Clearly the higher the degree of pipelining, the larger the improvement that can be obtained using the morphing technique, since it takes more cycles to evacuate and to refill the pipeline.

Table 1. Comparing morphing and non-morphing reconfiguration of a pipelined adder into a pipelined subtractor. N/A is short for "not applicable".

	With morphing (number of cycles)	Without morphing (number of cycles)
Reconfigure Figure 3a to Figure 3b	2	N/A
Reconfigure Figure 3b to Figure 3c	1	N/A
Reconfigure Figure 3c to Figure 3d	1	N/A
Reconfigure Figure 3a to Figure 3d	N/A	4
Time to flush and refill the pipeline	N/A	6
Total reconfiguration time	4	10
Speedup factor	2.5	1

Virtual Pipelines

An advantage of adopting a pipeline structure is the ease of mapping a large virtual pipeline onto a small physical pipeline. Our approach involves feeding back partial results to the physical pipeline which morphs between different sections of the virtual pipeline. The performance of such a system can often be enhanced by a temporary storage (Figure 4), as we shall discuss later.

Let us begin with an example: the mapping of a six-stage virtual pipeline onto a three-stage physical pipeline (Figure 4). The first three stages of the virtual pipeline are time-multiplexed with the last three stages. Morphing is used to replace one of the two pipeline configurations by the other.

This design operates as shown in Figure 4. Note that the physical pipeline operates in two modes: the "fill" mode and the "feedback" mode. In the fill mode, the first pipeline stage is connected to the external input and data start filling up the pipeline. Once the pipeline is filled up, partial results will emerge and will be stored in the temporary storage. When all input data have been processed by the first three stages of the virtual pipeline or when the temporary storage is full, the pipeline will operate in the feedback mode.

When an n-stage physical pipeline first starts in the feedback mode, its first stage will be reconfigured to become the $(n + 1)$-th stage of the virtual pipeline (Figure 4b). After reconfiguration is completed in the first pipeline stage, it is provided with the partial results from the temporary storage. When the computation is completed, the second stage of the physical pipeline will be reconfigured to become the $(n+2)$-th stage of the virtual pipeline (Figure 4c), and so on. For the example in Figure 4, eventually the partial results will flow through the physical pipeline configured to be the last three virtual pipeline stages (Figure 4d).

When the virtual pipeline has been emulated once, the result will start to emerge at the external output. When new data can be accepted, the physical pipeline will operate in fill mode again and will morph back to the first three virtual pipeline stages. Note that adequate flow control is necessary to stop the external input while the pipeline is operating in feedback mode. The next section will describe the use of the temporary storage to optimise pipeline performance.

Temporary Storage

First, note that if the physical pipeline only supports global reconfiguration, the temporary storage shown in Figure 4 is needed to hold partial results while the entire pipeline is being reconfigured. If the pipeline can be partially reconfigured at run time, then the temporary storage is not necessary as the pipeline itself can provide storage of partial results.

However, a small temporary storage will result in frequent reconfiguration, since the physical pipeline has to operate in "feedback" mode (see previous section) once the temporary storage is full. It will remain in the "feedback" mode until outputs are ready which will then free up space for further inputs. If the combined storage in the pipeline and the temporary storage is large enough

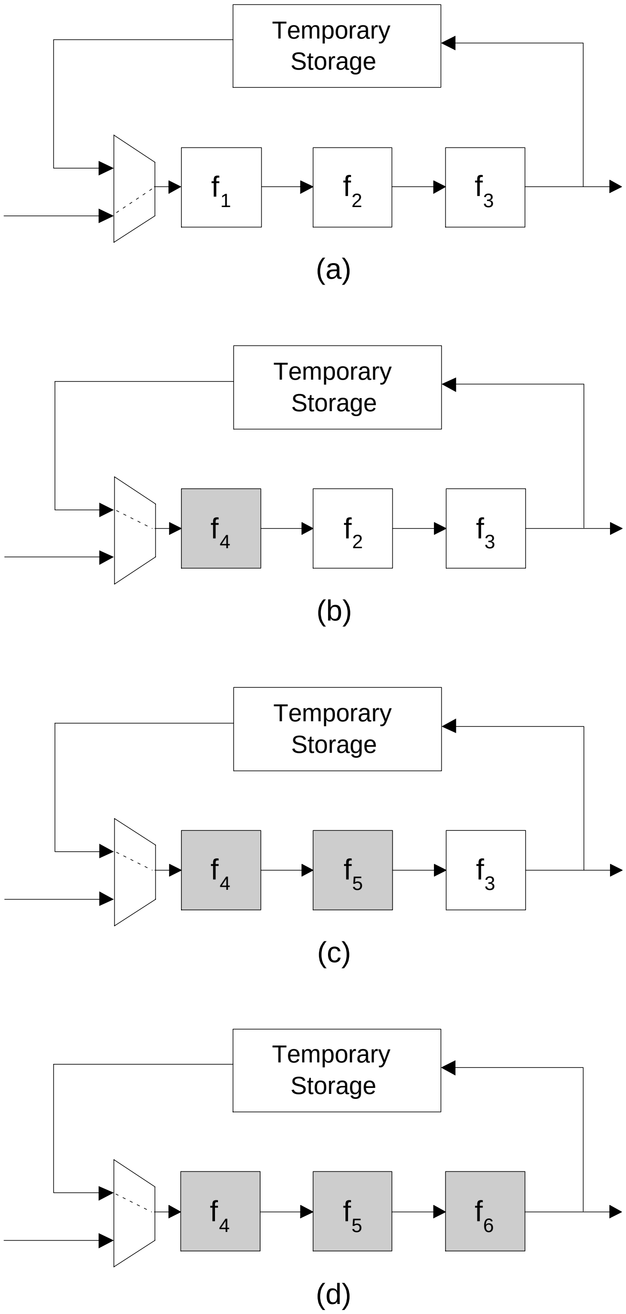

Fig. 4. Emulating a six-stage virtual pipeline $f_1, \ldots, f_6$ using a three-stage physical pipeline. The pipeline is in the fill mode for Step (a), and it is in feedback mode for Steps (b) to (d). The control to the switch that selects the external or the feedback data is not shown.

to contain all input data, then each virtual pipeline stage will only need to be emulated once. Having sufficient temporary storage is particularly important for pipelines which require a long reconfiguration time, since it would be desirable to minimise the frequency of reconfiguration for these pipelines.

Let us now consider different ways of implementing the temporary storage. If a large amount of temporary storage is required, then external memory can be used; otherwise on-chip registers or embedded memories within the FPGA may be sufficient. As explained above, pipelines supporting rapid reconfiguration can afford a small temporary storage. When this happens, the feedback connections can be made entirely on-chip, possibly using global connections in the FPGA so that output data from the last stage can be fed back to the first stage in the feedback mode. Global connections are provided in most FPGAs; such connections can themselves be pipelined to ensure high performance.

Another way of implementing a physical pipeline with little or no temporary storage is to partition the physical pipeline into half, and "fold" one half of the pipeline onto the other half by interleaving the components (Figure 5). This method avoids global connections at the expense of requiring an efficient integration of non-neighbouring elements in a pipeline structure.

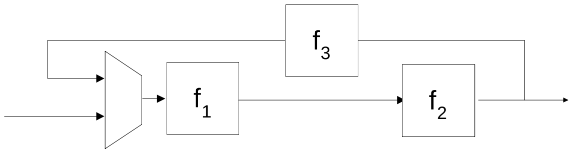

Fig. 5. A folded version of the physical pipeline in Figure 4a, which does not have temporary storage and avoids using global connections.

Virtual Pipelines on Xilinx 6200 PCI System

The viability of virtual pipelines has been demonstrated by a PCI board supplied by Xilinx Development Corporation, which contains a Xilinx 6216 or a Xilinx 6264 device and four 8-bit wide memories organised into two banks [7]. Each bank of memory can be accessed from either of the two separate address busses (Figure 6), and each of the four memories can be controlled individually. This memory architecture allows multiple modes of operation to be set-up by selecting multiplexers and bus switches for flow control in the desired manner.

This system provides a flexible platform for implementing virtual pipelines. One possibility is to use the two memory banks to provide the temporary storage (Figure 4) for a virtual pipeline. Partial results can be stored into one memory bank, and they can be used later when the FPGA has been reconfigured to

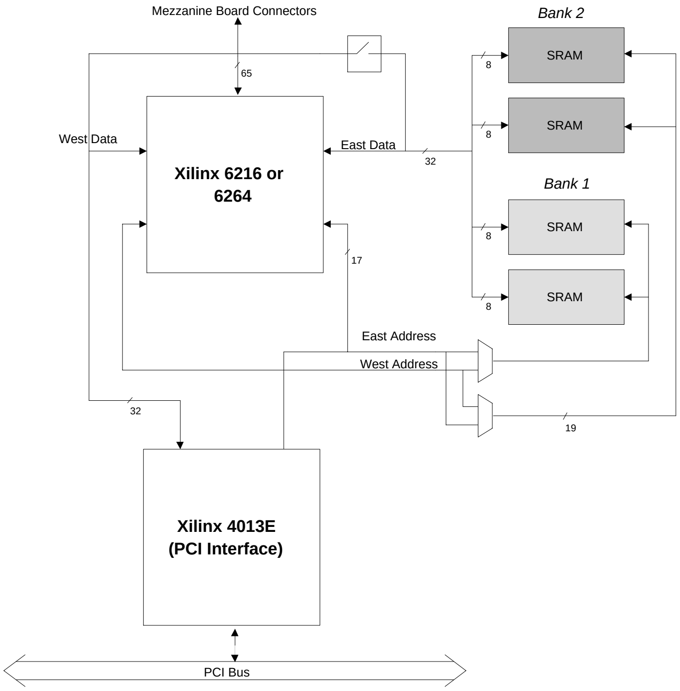

Fig. 6. Xilinx 6200 PCI system.

implement a different region of the virtual pipeline. Another possibility is to use the on-chip registers of the Xilinx 6200 FPGA to implement the temporary storage. If registers or global connections are at a premium, the folded structure (Figure 5) may prove to be an appealing alterative in implementing a physical pipeline.

Summary

This paper introduces morphing, an effective technique for reconfiguring pipelines. We explain how morphing can be applied to linear and mesh pipelines at word-level and bit-level, and how it can be implemented in Xilinx 6200 FPGA technology. We also describe the mapping of large virtual pipelines onto small physical pipelines, and how the resulting implementations can benefit from morphing.

Current and future work includes extending the scope of morphing to cover various architectural templates; this extension will enable us to morph between pipelines with different number of pipeline stages, or to morph a linear pipeline into a tree-shaped architecture. Frequently there are multiple ways of morphing between designs, and it will be important to evaluate the trade-offs involved.

The use of languages such as Ruby [2] and VHDL [9] for modelling hardware morphing is also being explored; we expect such work to result in techniques and tools for automating the implementation of morphing and virtual pipelines.

Acknowledgements

The authors are indebted to John Gray, Douglas Grant, Hamish Fallside, Tom Kean, Steve McKeever, Stuart Nisbet and Bill Wilkie for their constructive comments. The support of Xilinx Development Corporation, the UK Engineering and Physical Sciences Research Council (Grants GR/L24366 and GR/L54356) and the UK Overseas Research Student Award Scheme is gratefully acknowledged.

References

1. S. Churcher, T. Kean and B. Wilkie, "The XC6200 FastMap Processor Interface", in *Field Programmable Logic and Applications*, W. Moore and W. Luk (eds.), LNCS 975, Springer, 1995, pp. 36–43.
2. S. Guo and W. Luk, "Compiling Ruby into FPGAs", in *Field Programmable Logic and Applications*, W. Moore and W. Luk (eds.), LNCS 975, Springer, 1995, pp. 188–197.
3. J. Hadley and B. Hutchings, "Design Methodologies for Partially Reconfigured Systems", in *Proc. FCCM95*, P. Athanas and K.L. Pocek (eds.), IEEE Computer Society Press, 1995, pp. 78–84.
4. B. Hutchings and M.J. Wirthlin, "Implementation Approaches for Reconfigurable Logic Applications", in *Field Programmable Logic and Applications*, W. Moore and W. Luk (eds.), LNCS 975, Springer, 1995, pp. 419–428.
5. W. Luk, "A Declarative Approach to Incremental Custom Computing", *Proc. FCCM95*, P. Athanas and K.L. Pocek (eds.), IEEE Computer Society Press, 1995, pp. 164–172.
6. W. Luk, S. Guo, N. Shirazi and N. Zhuang, "A Framework for Developing Parametrised FPGA Libraries", in *Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, LNCS 1142, Springer, 1996, pages 24–33.
7. W. Luk, N. Shirazi and P. Y. K. Cheung, "Modelling and Optimising Run-Time Reconfigurable Systems", in *Proc. FCCM96*, K.L. Pocek and J. Arnold (eds.), IEEE Computer Society Press, 1996, pp. 167–176.
8. W. Luk, N. Shirazi and P. Y. K. Cheung, "Compilation Tools for Run-Time Reconfigurable Designs", in *Proc. FCCM97*, K.L. Pocek and J. Arnold (eds.), IEEE Computer Society Press, 1997.
9. P. Lysaght and J. Stockwood, "A Simulation Tool for Dynamically Reconfigurable Field Programmable Gate Arrays", *IEEE Transactions on VLSI*, Vol. 4, No. 3, September 1996.
10. H. Schmit, "Incremental Reconfiguration for Pipelined Applications", in *Proc. FCCM97*, K.L. Pocek and J. Arnold (eds.), IEEE Computer Society Press, 1997.
11. S. Trimberger, D. Carberry, A. Johnson and J. Wong, "A Time-Multiplexed FPGA", in *Proc. FCCM97*, K.L. Pocek and J. Arnold (eds.), IEEE Computer Society Press, 1997.