

Dynamic Communication in a Coarse Grained Reconfigurable Array

Robin Panda, Scott Hauck
Dept. of Electrical Engineering
University of Washington
Seattle, WA 98195
{robin, hauck}@ee.washington.edu

Abstract - Coarse Grained Reconfigurable Arrays (CGRAs) are typically very efficient for a single task. However all functional units are required to perform in lock step, wasting resources and making complex programming flows difficult. Massively Parallel Processor Arrays (MPPAs) excel at executing unrelated tasks simultaneously, but limit the amount of resources dedicated to a single task. We propose an architecture with an MPPA's design flexibility and a CGRA's throughput, capable of processing and transferring data in a pre-compiled schedule, with dynamic transfers between components. Alternative interconnect strategies are compared for silicon area cost and power utilization.

Keywords - CGRA; MPPA; flow control; interconnect

I. INTRODUCTION

Field programmable gate arrays (FPGAs) have long been used for accelerating compute intensive applications without the cost and difficulty of custom ASICs. However, their ability to be programmed for any logic function at the granularity of individual bits is unnecessary for many applications. If we limit primarily to standard logic operations performed on entire words of data, logic units can be used instead of LUTs. If the routing also operates on a word granularity, area can be saved by not storing or processing redundant configuration information.

Coarse Grained Reconfigurable Arrays (CGRAs) attempt these word optimizations with a sea of ALUs connected with a FPGA-like interconnect that operates on a word's worth of data at a time. The reduction in configuration information allows for time-multiplexing several configurations onto the same hardware, increasing hardware utilization. An alternate method of wiring computational elements together is a Massively Parallel Processor Array (MPPA). This is a network of more independent processors and their memory which communicate by passing messages.

CGRAs are good at using many processors for a single task, but the need for the whole device to operate in lock step is inefficient for control and handling multiple tasks in an application. MPPAs are great for control and a large number of tasks or applications, but are less efficient for pipelined tasks and cannot automatically spread operations across hardware. Our goal is to design an architecture that combines the benefits of both.

II. WHAT IS A CGRA

Our base CGRA [1] is composed of word-width functional units, including ALUs, shifters, or other special-purpose processing elements, connected with a word-oriented interconnect. It also includes LUTs and single bit communication channels to form a basic FPGA within the architecture for control and bitwise logic. Memory is local like in an FPGA with no native coherency mechanisms for shared memory. Block memories are managed by the application code, while registers required for timing and synchronization are managed by the CAD tools.

Each cycle, the configuration of the functional units' opcodes and addresses to be requested from register banks is sent to the functional units. These opcodes are scheduled by the compiler using modulo scheduling [2] for automatically pipelining user code. The ALUs in a CGRA cannot change contexts independently. In addition to sharing the context selection mechanism with other hardware, they must all follow the same schedule to ensure the proper synchronization of operations. This limits independence of separate threads and therefore the thread level parallelism that can be used. It is still a very good architecture for the parallelism and pipelining extracted by the compiler.

The interconnect, is handled similarly. Switchboxes (Fig. 1) connect 32-bit buses to other switchboxes and processing elements. An incoming bus (A) will fan out to multiplexers (B, C) in all other directions. Configuration memories (D) cycle through the configurations to select the appropriate inputs to the multiplexer B. After passing through the multiplexer, the bus is registered before being driven across the long wires to the next switchbox.

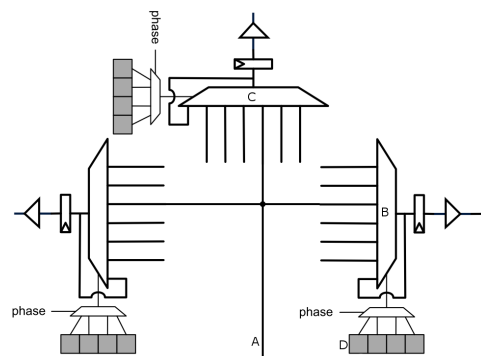


Figure 1. Switchbox schematic

A CGRA's cyclic schedules waste hardware because they must use predication instead of branching with data dependent instructions. When executing multiple applications, all tasks are locked to the same execution rate, slowing down some tasks and wasting more hardware on others. Some tasks may process data at significantly different rates [3], requiring careful optimization by the programmer.

III. MPPAS

In the Ambric MPPA [4], the ALUs from the CGRA are replaced with small processors with full, independent branching. This makes it well suited for small control tasks. However, since the processors and network are no longer executing in a lock-step manner, the rest of the architecture is more complicated. The processors must be individually programmed. The network routing can still be configured at compile time, but special sets of registers (Fig. 2) are needed to coordinate operations between different processors. The interconnect is flow-controlled, which allows processors to know when to operate on the data in the channel and can retain data in flight if the processor or network downstream is not ready to receive data.

Fig. 2 represents one stage of the communication network. In the absence of congestion, data is sent from the upstream stage via *Data In/Valid In*, held in register **B**, and sent out via *Data Out/Valid Out*. If the downstream stage is unable to receive data, it will deassert *Ready In*, and this stage will maintain the data in **B**. It will alert the upstream stage via *Ready Out*. However, since the upstream stage may have already forwarded a new value, this stage includes a second register **C** to hold this value.

Many applications do not easily break down into the hundreds of sub-tasks required for an MPPA in a human-comprehensible manner. Increasing throughput without these sub-tasks requires either heavyweight processors or dedicated hard blocks. Even when broken down, communication overhead can be a problem [6] when executing the tiny subtasks required for high throughput. Since hard blocks go unutilized in many applications and large, out-of-order processors are not efficient uses of die area and power, another solution is required.

IV. ARCHITECTURES

Ideally, a CGRA could be divided into a few regions that operate on different schedules called control domains. If at least some of the control domains have the ability to conditionally branch like MPPAs, control tasks could be implemented in a compact manner and eliminate costly predication within the main computation blocks. Multiple tasks, from the same application or not, could operate independently of one another, and each task would be spread over its maximum utilizable area. This spreading can be done automatically via CGRA-style tools, allowing the computation to be written in a logical, integrated manner that is easier for programmers to understand and compose.

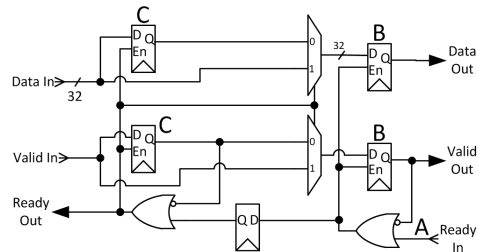


Figure 2. Ambric register set for dynamic flow control (adapted from [5])

This hybrid will operate on a fixed schedule in large processing blocks, but also have smaller control flow oriented tasks that communicate dynamically with each other and the computation blocks. Because the processing blocks take up more area, most of the architecture will be operating in the scheduled mode. To be most efficient, the system will need to be able to allocate resources to large, CGRA-style blocks or small, MPPA-style tasks on an application-by-application basis.

For such a system, independent scheduled and flow-controlled networks would be inefficient. Instead, there should be flow controlled communications on underlying scheduled resources, or scheduled communications on underlying flow controlled resources. Since much of the array is likely to be dedicated to CGRA-like computations, having the underlying hardware be scheduled is likely to be the most efficient. Because of this, this paper will explore adapting a scheduled network for dynamic communication, using the existing single bit resources for implementing the data valid and backpressure signals.

A. Dedicated and borrowed storage

The most straightforward way to shoehorn the handshake circuit of Fig. 2 into the switchbox of Fig. 1 is by adding dedicated shadow registers in parallel with each existing register like gray register **A** in Fig. 3. However, there are already more registers conveniently located in the same switchbox that might be borrowed. Each wire entering the switchbox fans out to multiplexers headed in every direction. For example, Fig. 3 shows a track from the West fanning out to the North and East. Each multiplexer is followed by a register, which can store the extra data during a stall if its mux is programmed properly. By adding the gray wire from **B**'s register to **A**, the data captured a **B** during a stall can be recovered during normal operation.

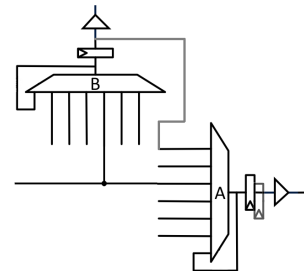


Figure 3. Fanout inside a switchbox

This design should be more efficient than putting in special purpose shadow registers. When dedicating registers to the task, the register overhead of supporting the handshaking will be paid by every channel with shadow registers of this capability, even if not used. This is very inefficient when the handshaking is used little. When borrowing registers from another channel, the primary overhead when the handshaking capability is unused comes from adding the additional input to each multiplexer and the logic gates for the handshake. However, each channel actually using handshaking will be wasting a set of perpendicular driver buffers that are quite large. Therefore, it is inefficient when handshaking is heavily used.

B. Half-bandwidth

The extra registers are required because it takes a cycle to notify the upstream register after the downstream register stops; since the upstream register may have already sent a value, storage is needed to save the incoming value. An alternative is to require a register to hold its value until the receiving register is guaranteed empty. Thus, if a stall occurs any value will have an empty register ahead of it already.

While this means we do not need to add storage to the interconnect, it only allows a data word to be sent every other clock cycle. Fortunately, it is rare to have an iteration interval of 1 in a CGRA [2]. Similarly, meaningful loops on an MPPA processor generally contain more than one instruction, like in [3] and in [6]. Therefore, in many cases it is impossible to send or receive data on every clock cycle anyway, and a half-bandwidth channel is sufficient.

The handshake must be modified as follows: If a register set holds *valid* data, then it must deassert *ready* to prevent upstream data from being lost in case the set downstream from it becomes no longer *ready*. If a set does not hold valid data, then it is *ready* to receive data. Thus, the *ready* signal output is simply the inverse of its own *valid* bit. When used for dynamic flow control, a half-bandwidth channel only uses up one 32-bit bus and two single bit routes, the minimum for implementing a handshake.

C. Interleaved channels

If a sending task can produce data every cycle and its receiving task can receive every cycle, they may benefit from a full bandwidth channel. If either is slower, eventually the storage available in the channel will be exhausted and the faster task will have to periodically stall until its data rate matches its partner's. Thus, since full-rate channels are rare, we would like to avoid adding hardware to the architecture specifically for these cases.

This need for a full-rate channel can be satisfied by alternating between two half-rate channels within the processing control domain. The sequence of words **A**, **B**, **C**, **D** would be sent with **A** and **C** on one channel and **B** and **D** on the other. Note that interleaving two half-bandwidth channels is roughly the same cost as borrowing registers,

only requiring two extra single-bit channels. This cost is only borne by channels that actually require this full bandwidth, since half-bandwidth and interleaved full-bandwidth channels can coexist in the same mapping

D. Minimal changes

The interconnect itself does not need to perform the handshake; it can be performed by the proper programming of the regular compute elements. To avoid using flow control and buffering within the channel, we instead pipeline the channel without stalls, and have a large enough FIFO at the end of the channel to catch all sent values when the receiving task stalls. If the FIFO begins to fill, a *ready/stall* signal is sent via a separate one-bit channel to the sender to throttle the data in the stream.

Ready must be predicted many clock cycles in the future to account for the longer propagation delay between where *ready* is generated and where the signal takes effect. For a channel of length N , the FIFO is no longer ready when there are less than $2N$ words: It will take N clock cycles for *ready* to propagate to the sending device and, at that time, there may be N more words in the channel.

In practice, an entire memory block would be dedicated to the receiver FIFO. The memory cannot be shared with any other functions because the full memory bandwidth is potentially required. Note that special provision must be made to ensure the FIFO logic itself does not stall. Otherwise, data already in flight in the channel will be lost. In fact, for all of the flow-controlled networks in this paper the network itself cannot stall with the surrounding logic; since these channels may traverse completely independent tasks, allowing those tasks to stall the channels can easily produce deadlock.

E. Tokens

For custom handshake processors, the memory required can be reduced even further by calculating when to stall on the sending side. A local register stores the count of available words in the FIFO. Every time a new word is sent by the sender, the count of storage remaining is decremented. Whenever the FIFO is read by the receiver, it will send a single bit *token* back to the counter to update it; when the counter receives the *token* it will increment the counter. When this counter reaches zero, the sending task must stall or its data may arrive at a full FIFO. Because of the reduced latency between where the stall signal is generated and where it stops the sender, this design can operate with fewer than $2N$ entries in the FIFO by throttling the send rate so no more data is in flight than can be handled. Therefore, it is more versatile when the FIFO memory is limited.

V. RESULTS

To evaluate these designs, they are compared for gate area and power consumption. In our base architecture, delay in the interconnect is dominated by the long wires between switchboxes and is not significantly different between the designs. The area of the logic gates added to a single

scheduled channel from one switchbox to another to add the dynamic flow control capability is calculated. To account for the existing logic used by the design we add the area of the scheduled resources for each dynamic channel actually used. For power, we calculate the energy used when an individual bit on the bus toggles. This is added to the energy used by the handshake logic for those channels using the handshake.

The areas and powers are calculated using the models developed in a 65nm process for the Mosaic project [1]. The Mosaic CGRA is used for wire length and driver requirements. Because both the built FIFO and the token counter are only required once per channel, their expenses are amortized over the average channel length from the Mosaic benchmarks suite. To put the calculated costs of each design in context, the average area and energy breakdowns of the various components in Mosaic are used to estimate the final effect on full chip area and power. The full chip power and area is graphed versus the percentage of dynamic channels in dynamic mode. This is repeated varying the percentage of scheduled channels capable of handshaking.

In Fig. 4, the lines labeled *built* represent a FIFO composed of existing processing elements as described in section D. $\frac{1}{2}$ *BW* is the design that operates at half data rate. *Borrow* uses registers from perpendicular channels. *Dual* interleaves two half-rate channels for full rate. *Extra* uses dedicated shadow registers. *Token* builds dedicated FIFO hardware with token counters.

As not all channels need be able to operate dynamically, we only add this capability to 25% of the channels; we then plot the full chip area for each design as the percentage of the dynamic channels used increases. For area, the $\frac{1}{2}$ -bandwidth design is the best in almost all situations. Full bandwidth communication will still require another method. If less than 20% of the available dynamic channels are in use, *borrowing* registers from an adjacent channel is the most efficient, with the *dual* interleaved channels a close runner-up that will be superior if there are channels that only need half bandwidth. However, as the usage increases, such as when everything is operating like an MPPA, the *token* method becomes the most area-efficient. If we increase the percentage of handshake-capable channels

beyond 25, the absolute areas increase, but the comparison between the designs remains roughly the same.

The energy graph tells a different story. The $\frac{1}{2}$ -bandwidth design is one of the worst power-wise for large channel utilization. The *token* counting FIFO has the best energy performance for all utilizations. We can combine the two graphs into the area-energy product. For this balance, we see that the $\frac{1}{2}$ -bandwidth and token versions provide the best results.

VI. CONCLUSION

The best implementation of a hybrid interconnect is dependent on usage. For full bandwidth signaling with most channels used, dedicated FIFO hardware with tokens is ideal. However, in real applications this high utilization and bandwidth is unlikely. Therefore, the best design is likely to be the half-bandwidth channels with interleaving when full bandwidth is required. To verify, future work is required to develop benchmarks for this sort of hybrid architecture to fully characterize the channel requirements for length, bandwidth, and density. With appropriate CAD tool support, the effects of stealing resources from a control domain can be determined along with more precise cost estimates.

REFERENCES

- [1] Brian Van Essen, *Improving the Energy Efficiency of Coarse-Grained Reconfigurable Arrays*, Ph.D. Thesis, University of Washington, Dept. of CSE, 2010.
- [2] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, S. Hauck, "SPR: An Architecture-Adaptive CGRA Mapping Tool", *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, pp. 191-200, 2009.
- [3] M. Haselman, N. Johnson-Williams, C. Jerde, M. Kim, S. Hauck, T. K. Lewellen, R. Miyaoka, "FPGA vs. MPPA for Positron Emission Tomography Pulse Processing", *International Conference on Field-Programmable Technology*, 2009.
- [4] M. Butts, A.M. Jones, P. Wasson, "A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing," *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp.55-64, 23-25 April 2007
- [5] A. M. Jones, "Asynchronous communication among hardware object nodes in IC with receive and send ports protocol registers using temporary register bypass select for validity information," U.S. Patent 7409533, Aug. 5, 2008.
- [6] R. Panda, J. Xu, S. Hauck, "Software Managed Distributed Memories in MPPAs", *International Conference on Field Programmable Logic and Applications*, 2010.

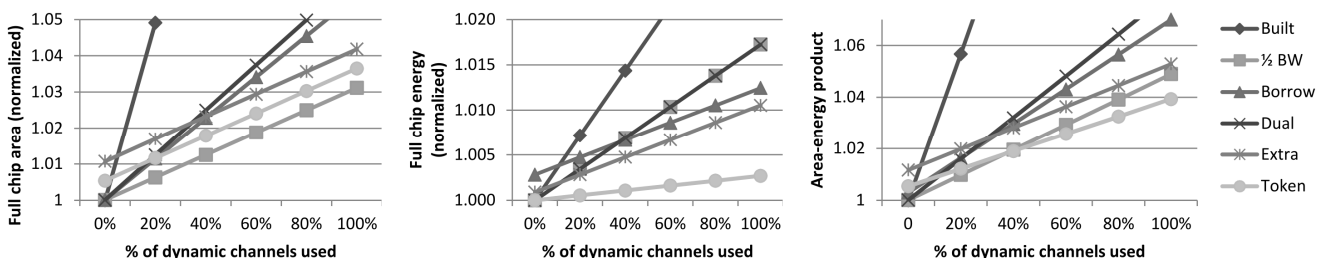


Figure 4. Normalized full chip area, energy, and area-energy product