ADDING DATAFLOW-DRIVEN EXECUTION CONTROL TO A COARSE-GRAINED RECONFIGURABLE ARRAY

Robin Panda, Scott Hauck

Dept. of Electrical Engineering
University of Washington
Seattle, WA 98195
email: robin@ee.washington.edu, hauck@ee.washington.edu

ABSTRACT

Coarse Grained Reconfigurable Arrays (CGRAs) are a promising class of architectures for accelerating applications using a large number of parallel execution units for high throughput. While they are typically good at utilizing many processing elements for a single task with automatic parallelization, all processing elements are required to perform in lock step; this makes applications that involve multiple data streams, multiple tasks, or unpredictable schedules more difficult to program and use their resources inefficiently. Other architectures like Massively Parallel Processor Arrays (MPPAs) are better suited for these applications and excel at executing unrelated tasks simultaneously, but the amount of resources easily utilized for a single task is limited.

We are developing a new architecture with the multi-task flexibility of an MPPA and the automatic parallelization of a CGRA. A key to the flexibility of MPPAs is the ability for subtasks to execute independently instead of in lock step with all other subtasks on the array. In this paper, we develop the special network and control circuitry to add support for this execution style in a CGRA with less than 2% area overhead. Additionally, we also describe the CAD tool modifications and application developer guidelines for utilizing the resulting hybrid CGRA/MPPA architecture.

1. INTRODUCTION

Field programmable gate arrays (FPGAs) have long been used for accelerating compute intensive applications. They do not have the development and validation difficulty or other costs of a custom ASIC, but are far faster than a general purpose CPU for most parallel or pipelined applications. The FPGA's programmability comes at a cost, though. Look up tables (LUTs) must store each output per input combination in a logical operation. Signals between operations require large and complex routing switches and common arithmetic operations are programmed down to each individual bit instead of a word at a time. These inefficiencies result in lower speed and higher power consumption compared to an ASIC implementation.

Many common calculations involve multi-bit words of data. If the individual bits share a configuration, this mitigates some of the inefficiencies of a FPGA. A more word-oriented FPGA could even use ALUs instead of LUTs. This further reduces area, power, and delay at the cost of flexibility in implementing non-datapath logic.

There are two main ways of designing a word-based array. Coarse Grained Reconfigurable Arrays (CGRAs) like MorphoSys [1], ADRES [2], VEAL [3], and Mosaic [4] have a sea of ALUs connected with an word-based FPGA-like interconnect. For better hardware usage, multiple configurations are time-multiplexed; these are generated using FPGA-like placement and routing like SPR [5] for automatic parallelization. Massively Parallel Processor Arrays (MPPAs) like ASAP2 [6] Ambric [7], and RAW [8] contain independent processors that communicate by passing messages. Each is programmed individually, with a traditional instruction set, but using memory local to each processor and explicit communication over the network instead of the large shared memory of a multicore CPU.

The speedup of a completely-parallel algorithm, according to Amdahl's law, reduces to the number of processing elements that are utilized [9]. The FPGA-like tools and configuration of CGRAs can use the parallelism and pipelining in the algorithm to map a single task to several processing elements automatically. However, the design is completely scheduled at compile time so they are poor at handling control and require predictability from their workflow. The traditional processors of MPPAs are great for control and variable workloads, but the programmer is required to manually split a computation into 100's of CPU-sized programs.

Our aim is to combine the benefits of each architecture to produce a hybrid with the control and workload flexibility of MPPAs, but with tasks automatically parallelized over multiple processing elements like in a CGRA. One of the key contributors to the flexibility of MPPAs is that the operation of the otherwise independent processors is synchronized solely by their communication. CGRA operation is synchronized implicitly by all

Supported by NSF grant #CCF-1116248 and DOE grant #DE-FG02-08ER64676

processing elements and communication resources executing to a fixed schedule produced at compile time.
To create the hybrid architecture, an MPPA-like flow controlled communication was added to the Mosaic CGRA's interconnect [10] for little additional area and energy in [11]. In this paper, we describe the new hardware and software modifications for the dataflow to actually control execution in our hybrid CGRA/MPPA architecture.

This paper is organized as follows: Section 2 describes the existing architecture classes. Section 3 proposes the hybrid architecture and discusses some of the requirements for implementation. Sections 4 and 5 present the new modifications to a CGRA developed in this research to add MPPA-like, data controlled, execution. Section 6 discusses how the data controlled execution is complicated by the CGRA and how this affects applications and CAD tools. Finally, section 7 summarizes the best resulting solution.

2. EXISTING ARCHITECTURES

A generalized CGRA is composed of various word-width functional units, which can include ALUs, shifters, or other special-purpose processing elements, connected with a programmable, word-width interconnect. It is often useful to include some LUTs and single bit communication channels to form a basic FPGA within the architecture for control and bitwise logic [10]. All memory is local, like in an FPGA, with no native coherency mechanisms for shared memory. Block memories are typically explicitly managed by the application code, while registers required for timing and synchronization are managed by the CAD tools as necessary.

The configuration, consisting of the functional units' opcodes and addresses requested from register banks, is sent to the functional units each cycle. The interconnect is controlled in a similar manner. Each word entering a switchbox fans out to multiplexers in all the other directions. A phase counter cycles through the different routes in configuration memory, synchronized with the incoming data, to time-multiplex the interconnect.

There are two main ways to think about these configurations. The most straightforward is as a set of predefined contexts, cycling after each clock cycle to simulate additional hardware, similar to a word-width version of a Tabula 3PLD [12]. The other is as an instruction word for the core of a clustered VLIW, but without the complex instruction processing components required for conditional branches. After passing through the multiplexer, the bus is registered before being driven across the long wires to the next switchbox. Resources like the configuration memory, decode, and clock gating are shared by all the wires in a bus.

The computing model of CGRAs is promising because tools such as SPR, can automatically spread a single computation across a large array of computation units from only a single program. However, many common styles of computation run into problems with this model:

- *Multiple tasks sharing the hardware share a single static schedule*. Because CGRA tools generally take only a single computation and spread it across the entire array, we must combine all tasks into one integrated computation. Thus, multiple independent tasks (such as processing on different streaming inputs), or multiple tasks for a single computation (such as the stages in an image-processing pipeline) must be combined into one loop. This is time-consuming, inefficient, and hard to support. On the other hand, this lockstep operation is what allows the architecture and CAD tools to be as efficient as they are.
- *They use predication for data-dependent execution*. Individual tasks usually have data-dependent operation, such as the choices in an IF-THEN-ELSE construct, or different modes of processing at different times in a computation (such as the phases in K-Means clustering). Since a CGRA requires every operation to occur at exactly the same time and place in each iteration, CGRAs use predication to handle data-dependent operation. This means that a large fraction of the issue slots in each iteration are consumed by operations that are simply predicated away.
- *All schedules run at once must be the same length.* Computation pipelines often have some tasks that are more complex, and therefore have a longer recurrence loop that limits their natural computation rate. In a CGRA, this is the Initiation Interval [13], or "II". Every task has a natural II, but a CGRA generally forces all tasks to use the same II, which is the maximum II of any task. If communication rates were identical, this is not a big problem. For computations with long tasks that are executed sporadically (such as PET [14]), or long tasks on lower-bandwidth paths in the computation, this imposes a significant performance penalty on the entire computation.

In the Massively Parallel Processor Array (MPPA), the hundreds of ALUs from the CGRA are replaced with small processors with full branching capability independent of other functional units. This makes it relatively inexpensive to handle small control tasks on chip, because predication is not required. The processors are individually programmed, often in a traditional language. However, since the processors and network are no longer executing in a lock-step manner, this complicates the coordination of the architecture. The interconnect multiplexers can no longer select based simply on clock cycle, and all memory blocks are coupled tightly with an individual processor or have a dedicated processor to sequence data.

MPPAs are dynamically synchronized by using communication channels with flow control between the processors. This flow control identifies when a valid data word is on the channel downstream and provides backpressure upstream. It is straightforward to understand that processors should stall until they see valid data arrive. However, if the process transmitting data can transmit faster than the receiver can receive, signals from full buffers prevent the sender from sending when the receiver is not ready. In this manner, the data synchronizes processing instead of a global program counter.

While some MPPA architectures such as RAW have included full dynamic communication routing, RAW required additional networks to avoid deadlock. More recent architectures, such as ASAP2 and Ambric, configure all their routing statically at compile time. Because the processors themselves are configured at compile time this does not result in a significant loss in flexibility. In an architecture with hundreds of processors, some of them can be programmed by the user to act as soft-routers for the remaining cases [15].

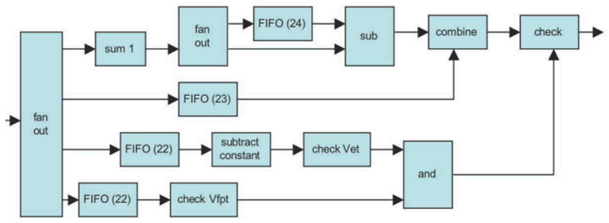


Fig. 1. Block diagram for running sum on an MPPA

Because an MPPA has multiple, independent processors loosely synchronized through communication channels, they avoid most of the previously mentioned problems with a CGRA. Each processor can have its own schedule, so different computations can have different schedule lengths, and independent tasks do not need to be combined into a single program. In addition, since the processors have true program counters, they can use branching for IF-THEN-ELSE constructs, and looping for supporting different modes. However, MPPAs have their own challenges:

- *MPPAs require the programmer to split computations into processor-sized chunk manually*. CGRAs leverage their system wide-synchronous behavior to provide tools that can automatically spread a computation across numerous processors. Thus, tools like SPR can take a single task and efficiently spread it across tens to hundreds of CPUs. MPPAs, with their more loosely coupled CPUs, do not provide the same functionality or tools, and instead force the application developer to write programs for each individual processor in the system. This is a huge task. For example, in [14], mapping a simple running sum threshold test to the Ambric MPPA required manually breaking the short loop into 8 processors and 4 FIFOs, all manually specified as shown in Fig. 1. This still took 6 clock cycles per input where a CGRA only needs one or two.
- *MPPAs generally keep most computations and results local to a single processor*. Although there are abundant resources connecting the individual processors together, communication between two processors in an MPPA is still noticeably more expensive than between CGRA ALUs operating in lockstep. This limits the achievable pipeline parallelism for a given task; thus many processors are lightly loaded while the processor with the most complicated task runs constantly [16].

3. MULTIKERNEL HYBRID

CGRAs prove to be quite good at utilizing many processing elements for a single kernel of execution. They are inefficient for control and handling multiple tasks in an application. MPPAs are great for control and a large number of tasks and/or applications, but are less efficient for individual pipelined tasks that are more difficult to spread across MPPA hardware. Our hybrid breaks a CGRA into a few regions, called control domains, each executing different tasks on different schedules, but with each task still spread over its maximum utilizable area for high throughput. Broken down in a logical manner, each task is easier to understand than in its single kernel counterpart and can be composed by different programmers. Data is only routed to relevant control domains, so the amount of hardware wasted by predication is significantly reduced over the CGRA.

A CGRA requires architectural and tool modifications to execute different, yet related tasks simultaneously. Within a control domain, routing and computation should still operate according to compiler-determined schedules for efficiency. Communication between control domains must be dynamically flow-controlled like in an MPPA and not scheduled. Details on how resources in a control domain are used to implement flow-controlled communication between unrelated control domains are described in [11].

We assume that the dynamic communication itself is possible and that there are two bits in the interface; one tells the sender the receiver is ready to receive and one tells the receiver that the sender has data. When read data is required from this network and none is available, or there is no room to write data when a send is scheduled, control domains must respond appropriately. In most cases, no useful computation can be done and the best reaction is to simply stall and wait for the situation to resolve. In the remainder of this paper, we develop novel methods to support stalling individual control domains effectively on an architecture like Mosaic.

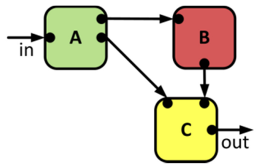


Fig. 2. Control domain communication pattern

As an example, CGRA regions communicate as shown in Fig. 2: Control domain **A** sends data to domains **B** and **C**, and **B** also sends to **C**. **B** cannot work until **A** provides the data to it. Similarly, anything executed in **C** before **A** and **B** have provided appropriate inputs is useless at best and often results in improper functionality. Therefore, tasks must be aware of the data dependency.

The device is configurably split into CGRA regions of varying sizes for different applications, each of which stalls independently. While our goals could be accomplished by designing an array of CGRAs that communicate using dataflow communication, this would fall prey to resource fragmentation. Two floorplans from example applications written in the Multi-Kernel Macah language [17] are shown in Fig. 3 from [18]; each color is a separate CGRA region. Within a region, SPR is used to handle data routing and instruction scheduling and placement just like a normal CGRA.

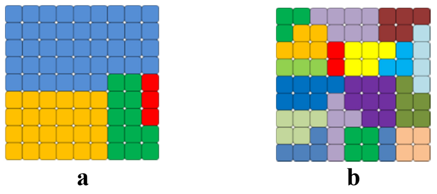


Fig. 3. Floorplans for **a**: Bayer filter and **b**: discrete wavelet transform

The development of this novel hybrid device gives rise to a research question that, as far as we know, has not been addressed before: How do we effectively stall an entire CGRA region of such a hybrid device? Unlike an MPPA, which stalls a single processor at a time, we must support coordinating the stalling of many ALUs simultaneously so all schedules remain synchronized. To support stalls in a hybrid CGRA/MPPA, we need to answer the following open research questions:

1. What is the most efficient mechanism for stalling an individual processor and its interconnect resources, without disrupting inter-task communications?
2. How do we coordinate the simultaneous stalling of multiple processors within the chip, particularly when the task size and shape is configurable?
3. How do we tolerate the potentially long stall latencies from when a communication channel initiates a stall to when the entire task actually does halt operation?

In the rest of this paper, we address each of these open research questions, developing a novel mix of hardware, software, and application approaches to provide stall support for hybrid MPPA/CGRA systems. As such, we provide an efficient and effective mechanism to aid in harnessing the best of both computation styles.

4. STALL MECHANISM

The first question our research must answer is how to stall execution on a single processing element, including a processor and its associated interconnect resources. For this, we assume that each end of a dynamic communication channel has a streaming communication port (black circles in Fig. 2) to interface with the control domain and buffer data to or from the channel. Fig. 4, zooms to the top right corner of control domain C to show a read port on top and a write port on the bottom.

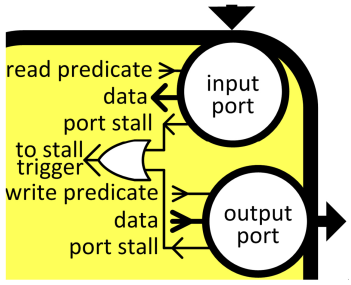


Fig. 4. Stream port interfaces

Each read or write port takes a predicate from within the control domain that indicates whether the kernel actually wishes to perform the scheduled communication that cycle. Stalls occur when an active port's predicate is true, and there is no data in the buffer for reads, or there is a full buffer for writes. In these cases, the stream port indicates to the control domain that it should stall and not proceed to the next instruction. If any stream port within a control domain signals a stall, the entire control domain halts at the exact same place in the schedule. Therefore, we also need a network to aggregate all stall signals from the ports and deliver it to the mechanism that stalls the control domain.

Because the entire state of a clock domain is held in registers in the interconnect and processing elements, and the domain's own memories, disabling the clock appears to be an excellent way to stall. Handshaking within the control domain would require each register in the interconnect to be replaced with FIFOs and complicate instruction sequencing. By hooking into the clock tree itself, we support stalls for control domains of various sizes by stopping the clock on different levels of the clock tree. Fig. 5 shows a flattened clock distribution tree to four processing elements, with some stall trigger network (discussed in section 5) wired into the different levels. When we activate the stall trigger and it stops the clock on the lowest level, this halts the clock for a single processing element. Moving up a level, we stop the clock for two elements, while if we activate at the highest level, all four processing elements are frozen. In an actual H-tree implementation, this stops a two by two region of processors. While this implementation is convenient, it restricts control domains to powers-of-two dimensions.

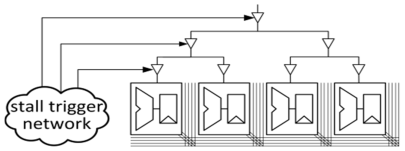


Fig. 5. Triggering stalls using the clock tree

Unfortunately, we cannot just stop all clocks in a region. The stream ports and the logic required for buffering and generating the stall signals within the communication channel interface itself cannot stall; if they stall, they are not able to detect the conditions to unstall. Running an additional ungated global clock tree and keeping it synchronized with the *configurable*, gateable control domain tree is troublesome to implement. Note that the flow controlled interconnect implementations discussed in [11] potentially steal registers, multiplexers and drivers from control domains that are unrelated to the sender and receiver. These resources should not stall when the unrelated domain stalls and so now we need a programmable clock select to choose between gated and free running clocks. With all of the duplication of clock gating and clock trees, this will be an expensive proposition. Using the models, benchmarks, and architecture from and assuming our original clock tree was as in, [19], this would result in an increase in area by 1.6% and used as much as 11% of the total execution energy if we never actually have to stall.

The other solution is to use the time multiplexing capability to generate no-ops for all functional units and routes. In the Mosaic CGRA, the interconnect contains feedback paths that automatically sustain a value and gate the clocks in the corresponding pipelining registers for reducing activity (and power) when a bus is unused. Similarly, there are configuration bits specifically for enabling the clocking of the registers within the functional units themselves [10] for temporary storage. Therefore, a configuration already exists that freezes all register data, disables memory writes and disables the stream ports, everything we need for a stall.

As previously mentioned, each processing element has a counter that cycles through all of the time-multiplexed configurations. We hijack these during a stall to select our own special stall configuration that was not part of the regular schedule; we could freeze the control domain without directly interfacing with all the components like in the configurable clock gating. The stall signal only has to control some logic associated with the phase counters. When creating this configuration at compile time, any resources that operate disregarding stalls simply receive the same configuration they have for every other phase so they execute normally while stalled.

To build this mechanism, we need stall signaling to freeze the counters at the existing value so execution can resume from the right instruction after the stall. They then output a special phase number (a reserved phase ID of all 1's) to select our stall configuration from the existing configuration memories as shown by the black configuration on the components in Fig. 6. This configuration gates the relevant clocks for each individual register, without adding new, configurable clock trees, so it is more energy efficient than the clock gating methodology. There are at most a few counters per processing element. Therefore, the main modification is practically free in terms of logic area, compared to building a stall network and configurable clock tree. However, configuration memory is not a trivial resource. Assuming the architecture in [4] supports a maximum II of 8, the total area overhead is 0.86% and it could increase our full-chip energy by a maximum of 3.7% (stalling once per II). However, we would hope a well-developed application stalls rarely, so this is our preferred stall mechanism because it uses only 52% of the area and 33% of the energy of the clock gating method.

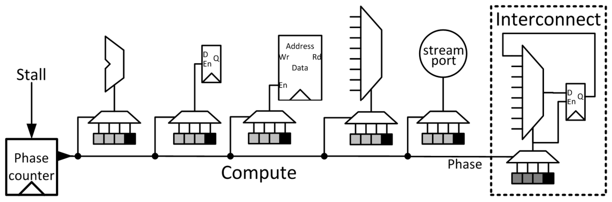


Fig. 6. Phase counter stalling (clock & data not shown)

5. TRIGGER NETWORK

As discussed in the previous section, we can stall an individual processor in the system efficiently via our no-op generation technique. Now we must develop a mechanism to coordinate the stall of an entire region simultaneously. We need a network to combine the stall signals from all stream ports in a control domain, and deliver them to all the phase counters and stream ports at the same time so that the control domain remains synchronized.

The stall trigger network faces several challenges:

- Since an individual CGRA region may be large, we cannot expect the stall signal to reach every processor in one clock cycle, but must instead pipeline the stall signal so that it occurs at the correct time.
- Multiple stall sources exist, one for each read and write port for this task.
- Since region size and shape is configurable, the stall trigger network must be configurable.
- Stall latency can be a major performance limiter, so stalls should happen as quickly as possible. Specifically, small tasks should not be forced to have as long a stall latency as larger tasks.

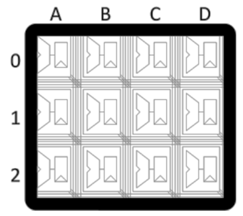


Fig. 7. Single control domain

To demonstrate the impact of multiple stall sources, consider the zoomed in view of a single control domain shown in Fig. 7. Coordinates for the 12 processors have been labeled across the top and left side. While a dedicated stall network could be faster, assume it takes one clock cycle for a trigger to get from one processor to its neighbors, like regular data. If the stream port at **A2** has no data and tries to read, the stall controller at **A2** is aware of the need to stall before the controller at **D0** and waits for the stall signal to propagate to **D0** before stalling. This ensures all processing elements stall simultaneously. This delay can be configured and coordinated in a couple ways.

In this paper, we present two new mechanisms we have developed for stall triggering. First, we can have a peer-to-peer mechanism, where nearest-neighbor communication is used to distribute out the stall signal, something like the ripples on a pond. The second mechanism chooses one of the processor's stall controllers as master, which reduces the amount of communication needed, and thus reduces power.

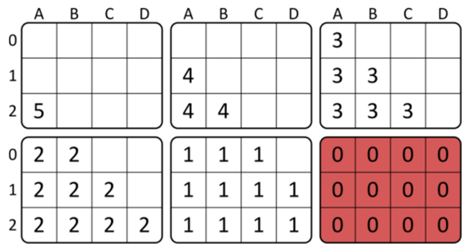
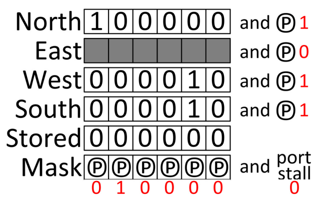


Fig. 8. Peer-to-peer stall trigger propagation

One way to send stall signals is to determine the latency dynamically and communicate this value along with the trigger. Fig. 8 shows the earliest stall for each processor in our control domain for a stall triggered at a port in **A2** in 6 time snapshots from left to right, and top to bottom. **A2** has a stall latency of 6 so it is configured to wait 5 cycles for stalls it triggers, as shown in the upper left. It sends the trigger to **B2** telling **B2** to wait 4 cycles then both count down. **B2** then tells **C2** to wait 3 cycles. This proceeds through **D2** and **D1** similarly, reaching 0 cycles upon arriving at **D0**. At this time, *all* processing elements are at 0 and begin the stall.



North 1 0 0 0 0 0 and Ⓟ 1
East and Ⓟ 0
West 0 0 0 0 1 0 and Ⓟ 1
South 0 0 0 0 1 0 and Ⓟ 1
Stored 0 0 0 0 0 0
Mask Ⓟ Ⓟ Ⓟ Ⓟ Ⓟ Ⓟ and port stall
0 1 0 0 0 0
0

$new_{0..n} = mask * port\ stall$

$stored_{i-1} = N_i N_{Ⓟ} + E_i E_{Ⓟ} + W_i W_{Ⓟ} + S_i S_{Ⓟ} + new_i + stored_i$

$clock\ gate = stored_0$

$to\ neighbors = stored$

Fig. 9. Stall vector handling. Ⓟ represents a program bit

While the nearest-neighbor countdown is simple to understand, we must handle stalls from multiple sources and their associated un-stalls also. To handle this general case, we use a bit-vector. An example for **D1,** in the 5th snapshot above, is shown in Fig. 9. Instead of counting down, this bit-vector is shifted to the right and the processor stalls if the LSB in the stored value is a 1. Assume our stalls are only to take 1 cycle. **D1** receives 000010, representing stall-in-1 and unstall-in-2, from the west and south (**C1** and **D2**). If a port in **D0** begins to initiate a stall at the same time, it sends 100000, representing stall-in-5 (from the north). **D1** ANDs these with programming bits that indicate which neighbors are in the same control domain so that the data from the East is ignored. The results all get ORed with the previously stored value shifted and the locally generated stall signal (which has its latency programmed by the mask). Here, **D1** stores 100010 for this pattern and sends that to its neighbors. If the LSB is a 1, the processor stalls; so in this case, it stalls on the following cycle.

A less expensive way is to designate one controller as the master when configuring the array. This central controller should be centrally located for minimum latency, such as **C1** in our example. Single bit stall triggers pass through neighboring processors in a pattern configured at compile time, are ORed together, and sent to the next processor on the way to **C1**. **C1** sends the coordinated stall back out in the opposite of the incoming pattern on a single bit return network. Because this path is known, each processor simply waits a configured amount of time before stalling or unstalling. This is similar to the bitmapped peer-to-peer method, but bitmaps are computed within each stall controller; the communication is serial.

All trigger controllers are statically configured to delay a fixed amount because the final trigger always propagates along the same path. Since **C1** has the first controller to know about any stalls, it always waits 3 cycles for the remainder of the trigger propagation. One cycle later, **B1**, **C0**, **C2** and **D1** are notified and wait 2 cycles. On the next cycle, the remainder, except for **A0** and **A2,** are notified and wait 1 cycle for the final two controllers to be notified. If the initial trigger came from **A2**, this propagation is **A2** > **B2** > **C2** > **C1** > **C2** > **B2** > **A2** for 6 cycles of latency from this network.

The stall networks use different amounts of resources. The bitmapped peer-to-peer stall trigger controller requires a number of communication bits equal to the maximum supported stall latency, to each neighboring controller. The centralized network only requires one bit in each direction. Using the models from [11], we calculated the percent increase in full-chip area from adding these networks for a maximum supported latency of 9 cycles. The central network has a full-chip area overhead of only 0.75% while the peer-to-peer network increased area by 2.20%. The internal logic is quite similar; the main difference is the long wires and large drivers between neighboring processors so the peer-to-peer method could have up to 4.5X the energy overhead of the central network.

6. LATENCY TOLERANCE

The new mechanisms for stalling individual processors, and coordinating the stalling of an entire CGRA region within a hybrid device, cope with regions of very different sizes and shapes. Each has a latency between detecting the need for a stall and the individual processors actually halting execution that must be tolerated somehow. Execution must be stopped before data loss occurs. If we attempt a send or receive and the resulting stall can only happen some number of cycles in the future, the stall signal must be triggered before the read or write actually fails. In this section, we will explore what happens within the stream ports to handle this latency and what this means to the CAD tools like SPR and the application developer.

A send FIFO with infinite capacity never needs to stall because data cannot be lost. With a more realistic buffer, we must stall before the buffer is full. To accommodate a stall latency of L clock cycles, the stall signal must be triggered L cycles before the buffer becomes full. This requires a prediction of the FIFO state L cycles from now. If the current number of words available in the FIFO is C, then the lower bound on the FIFO status L cycles from now, with no other information, is $C - L$. If $C - L <= 0$. In this situation, the FIFO could fill within L cycles so the stream port triggers a stall in advance, to be safe. This is straightforward to do; we simply add the logic shown in Fig. 10 **a**. While this prediction is acceptable for low values of L, it forces L words of memory to be empty at all times, except possibly during a stall.

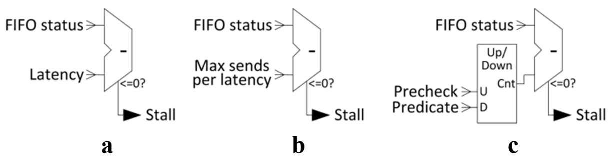
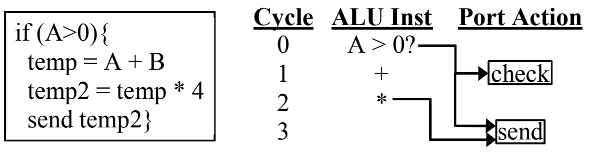


Fig. 10. Latency offset calculators for, **a**: latency only, **b**: CAD estimate, and **c**: precheck methods

A more accurate prediction is more efficient for larger control domains with high latencies. It is unlikely for a send to occur every clock cycle. To execute a send, the kernel sends the data value to the port including a predicate bit that specifies the input should be sent. The CAD tool schedules these operations. Therefore, it can count the possible sends per II. This ratio is used to determine the maximum number of sends that could occur in L cycles for this particular stream port in this specific program, which is programmed into the stream port as shown in Fig. 10 **b**.

However, some of these sends are predicated away, wasting a spot in the FIFO. Utilizing the entire FIFO is preferable for overall area and energy efficiency, but requires an actual count of the pending sends and not just an upper bound. Fig. 11 shows the schedule on the right for the pseudocode on the left. The CAD tools can deliver the send predicate to the stream port L cycles (2 in the example) in advance as a pre-send check. When the stream port receives this pre-check, it increments a counter to account for the pending send (Fig. 10 **c**). When the send actually occurs (predicate is true), the counter decrements. In this way, the counter always knows how many spaces must be available in the FIFO for the sends that will *actually* be done in the next L cycles.

This is usually easy for the CAD tools. If the predicate is not available until the same time as the data, CAD tools must use programmable registers or memory to delay the data for L cycles until it is appropriate to deliver to the stream port. In the worst-case for sends, this uses only as much extra storage as would have been wasted with the "sends per II" estimate. Modulo Scheduling such as that in SPR can generally hide this latency since the actual stream writes should not be in a loop carry dependency.



```
if (A>0){
  temp = A + B
  temp2 = temp * 4
  send temp2}
```

Fig. 11. An early buffer check for the code on the left

At first glance, a receive is the exact opposite of a send. The kernel requests data from a FIFO with a predicate and then uses the data present on the interface from the stream port to compute. Instead of consuming a memory location and creating data like in a send, it creates a newly empty memory location and consumes data. However, there is a crucial difference. To guarantee operation when sending, we ensure that the FIFO is sufficiently empty; it is quite easy to over-provision the FIFO. For receives, the equivalent is to use our precheck to ensure that the FIFO is sufficiently full, but there is no way to overprovision a FIFO to have extra data.

Receives classify into three different styles of communication, each with a somewhat different method of tolerating the stall latency:

1. *High-bandwidth communication:* The common case is a high rate send, like sending image sequences for processing. In this case, we can require a buffer to have enough data present to equal the estimate of reads in the stall latency. Although there is a short initial delay to fill the buffer, these transfers work the same way as sends after that. The end of the stream can be padded to allow the computation to finish.
2. *Unpredictable low-bandwidth communication (interrupts):* It makes no sense to stall when there is no input because continued operation is desired. The programmer should utilize a non-blocking type of read that never stalls the machine to poll these inputs and handle the presence or lack of data within their code.
3. *Regularly scheduled, low-bandwidth communications:* An example of this is a stream that sends a decryption key to a decryption kernel, which uses it to process a regular high-rate stream. These often have dependency cycles (explicit in the coding, or implicit due to hardware limitations) that are exacerbated by our latency needs and suffers deadlock if not handled properly.

Styles 1 and 2 are easily supported; style 3 causes problems. In Fig. 2, **A, B** and **C** can form this troublesome network. **A** does some basic processing and passes some of the results to **B,** and the remainder to **C**. **B** takes its data and uses it to recover a decryption key, which it sends to **C** to begin decrypting the data from **A**. If the stall latency of **C** is 10 cycles, the basic estimation method requires 10 keys in the buffer before execution can proceed. **A** sends data to **C** until its buffer fills, while **C** is stalled waiting for 9 more keys. This stalls **A** so no more data is sent to **B,** which stalls due to a lack of data, and no new keys will be made. Notice that the channels' backpressure has formed a cycle to cause deadlock though no such cycle appears in Fig. 2.

The CAD tools will always estimate at least one read per II because that is the extent of the schedule. This means that the amount of buffering required is only reduced by a factor of II. This could still be too high to allow for proper operation. Therefore, the best way to handle this is by having the CAD tools route an early copy of the read predicate as a "precheck" to only check for input when it will be necessary soon as shown in Fig. 10 **c**. This will eliminate deadlocks that could be caused by the stall latency, though it may result in a few wasted clock cycles. Adding these is inexpensive, with an area overhead of only 0.025%.

7. CONCLUSIONS

A hybrid CGRA/MPPA system combines the automatic parallelization and compilation support of CGRAs, with the multi-task support of MPPAs, to deliver high performance and low power implementations of streaming computations. This requires the ability to stall a configurably sized region of statically-scheduled processors simultaneously, so that they can remain in lockstep, yet stay synchronized with unpredictable external dataflow and events. To solve this problem we split the problem into three components: How do we stall one processor without disrupting inter-task communication? How do we coordinate the simultaneous stalling of multiple processors in the chip? How do we tolerate the resulting long stall latencies?

Stalling individual processors using traditional clock gating methods would increase our area by 1.6% and increase energy use by 11%. We developed a mechanism that adds only 0.86% area and only increases energy by 3.7% at most. This mechanism uses the existing configurability in the CGRA with only small changes to the phase counter and an additional configuration. Stall coordination must handle stall and unstall inputs from multiple stream ports everywhere in the control domain to trigger simultaneously. By pre-programming these routes and stall latencies, coordination only requires two new single-bit signals. The resulting coordination hardware only adds 0.75% area to the architecture.

Finally, we developed mechanisms and strategies to handle potentially long stall delays. Modifications to SPR during scheduling allow cheap hardware stall predictors (0.025% full-chip area) to start the stall triggers early enough to ensure proper operation and efficient storage usage. In most cases, the latency is completely hidden to the application. Combined, these techniques provide a simple but effective mechanism for stalling arbitrarily sized CGRA regions within an overall MPPA system with hardware overhead of only 1.6%.

8. REFERENCES

[1] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh and E. Chaves Filho, "MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications," *Computers, IEEE Transactions on,* vol. 49, no. 5, pp. 465 -481, May 2000.

[2] B. Mei, S. Vernalde, D. Verkest, H. De Man and R. Lauwereins, "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, 2003.

[3] N. Clark, A. Hormati and S. Mahlke, "VEAL: Virtualized Execution Accelerator for Loops," in *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, Washington, DC, USA, 2008.

[4] B. Van Essen, R. Panda, A. Wood, C. Ebeling and S. Hauck, "Managing Short-Lived and Long-Lived Values in Coarse-Grained Reconfigurable Arrays," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, 2010.

[5] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling and S. Hauck, "SPR: An Architecture-Adaptive CGRA Mapping Tool," in *FPGA '09: Proceeding of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, New York, NY, USA, 2009.

[6] D. Truong, W. Cheng, T. Mohsenin, Z. Yu, A. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, A. Tran, Z. Xiao, E. Work, J. Webb, P. Mejia and B. Baas, "A 167-Processor Computational Platform in 65 nm CMOS," *Solid-State Circuits, IEEE Journal of,* vol. 44, no. 4, pp. 1130 -1144, April 2009.

[7] Ambric, Inc, Am2000 Family Architecture Reference, 2008.

[8] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe and A. Agarwal, "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs," *Micro, IEEE,* vol. 22, no. 2, pp. 25 - 35, Mar/Apr 2002.

[9] M. Hill and M. Marty, "Amdahl's Law in the Multicore Era," *Computer,* vol. 41, no. 7, pp. 33 -38, July 2008.

[10] B. Van Essen, A. Wood, A. Carroll, S. Friedman, R. Panda, B. Ylvisaker, C. Ebeling and S. Hauck, "Static versus scheduled interconnect in Coarse-Grained Reconfigurable Arrays," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, 2009.

[11] R. Panda and S. Hauck, "Dynamic Communication in a Coarse Grained Reconfigurable Array," in Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on, 2011.

[12] T. R. Halfhill, "Tabula's Time Machine," *Microprocessor Report,* Mar. 29 2010.

[13] B. Ramakrishna Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," in *In Proceedings of the 27th Annual International Symposium on Microarchitecture*, 1994.

[14] M. Haselman, N. Johnson-Williams, C. Jerde, M. Kim, S. Hauck, T. Lewellen and R. Miyaoka, "FPGA vs. MPPA for Positron Emission Tomography pulse processing," in *Field-Programmable Technology, 2009. (FPT 2009) International Conference on*, 2009.

[15] R. Panda, J. Xu and S. Hauck, "Software Managed Distributed Memories in MPPAs," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, 2010.

[16] Z. Yu and B. Baas, "A Low-Area Multi-Link Interconnect Architecture for GALS Chip Multiprocessors," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on,* vol. 18, no. 5, pp. 750 -762, May 2010.

[17] A. Knight, "Multi-Kernel Macah Support and Applications," M.S. thesis, Dept. Elect. Eng., University of Washington, Seattle, 2010.

[18] A. Wood, B. Ylvisaker, A. Knight and S. Hauck, "Multi-Kernel Floorplanning for Enhanced CGRAs," Dept. Elect. Eng., Univ. of Washington, Seattle, WA, Tech. Rep., 2011.

[19] J. Oh and M. Pedram, "Gated Clock Routing Minimizing the Switched Capacitance," in *Design, Automation and Test in Europe, 1998., Proceedings*, 1998.