

# Software Managed Distributed Memories in MPPAs

Robin Panda, Jimmy Xu, Scott Hauck  
Dept. of Electrical Engineering  
University of Washington  
Seattle, WA 98195

Email: {robin, jimmyxu, hauck}@ee.washington.edu

**Abstract**—When utilizing reconfigurable hardware there are many applications that will require more memory than is available in a single hardware block. While FPGAs have tools and mechanisms for building logically larger memories, it often requires developer intervention on word-oriented devices like Massively Parallel Processor Arrays (MPPAs). We examine building larger memories on the Ambric MPPA. Building an efficient structure requires low-level development and analysis of latency and bandwidth effects of network and protocol choices. We build a network that only requires only five instructions per transaction after optimization. The resource use and performance suggests architectural enhancements that should be considered for future devices.

**Keywords**-MPPA; memory; pipelining;

## I. INTRODUCTION

As reconfigurable hardware targets a wide range of applications, it is infeasible to provide memory blocks that will have sufficient capability for all tasks and not be underutilized in the majority of cases. FPGA vendors include mechanisms in the development environments and hardware for building larger memories and accessing them as a single logical memory. Word-oriented accelerators like Massively Parallel Processor Arrays (MPPAs) and Coarse Grained Reconfigurable Arrays (CGRAs) promise increased efficiency [1], but users will still require these mechanisms.

For example, in our lab we have implemented a portion of the processing pipeline for PET scanners on a variety of platforms [2, 3]. This algorithm uses lookup tables for implementing several functions. In developing for the Ambric MPPA, the need for large memories proved to be one of the largest challenges [2], adding to the already difficult task of partitioning up a task into separate blocks.

In this paper, we will discuss how to build large, distributed memories on the Ambric architecture. We aim to match the ease of use and low hardware and cycle time cost that FPGAs currently enjoy. We discuss the instruction level optimizations that are required and the protocol for communicating with these memories. Keeping memory transactions synchronized will limit network options, and pipelining must be maintained throughout to ensure throughput. Some architectural features unique to Ambric will aid our development and some present unique challenges to be overcome.

## II. AMBRIC MPPA

The Ambric AM2045 MPPA is an array of 336 32-bit RISC processors and 336 memory banks [4, 5]. Each processor can execute a single Java object and can only communicate with other Java objects through asynchronous FIFO channels. While the processors are synchronous in their programming and registers, the clocks are not necessarily shared with other processors, yielding a Globally Asynchronous, Locally Synchronous (GALS) architecture. Processor operations are synchronized solely through data arrival.

Ambric is broken up into 42 “brics”; half of a bric is shown in Fig. 1. The other half is identical except for being vertically mirrored. The RAM Units (RUs) on the bottom are made up of 4 2KB RAM banks, RAM access engines, and interconnect to connect the banks to the engines. The engines are specialized processors responsible for interfacing between the memories and the processors used for computation.

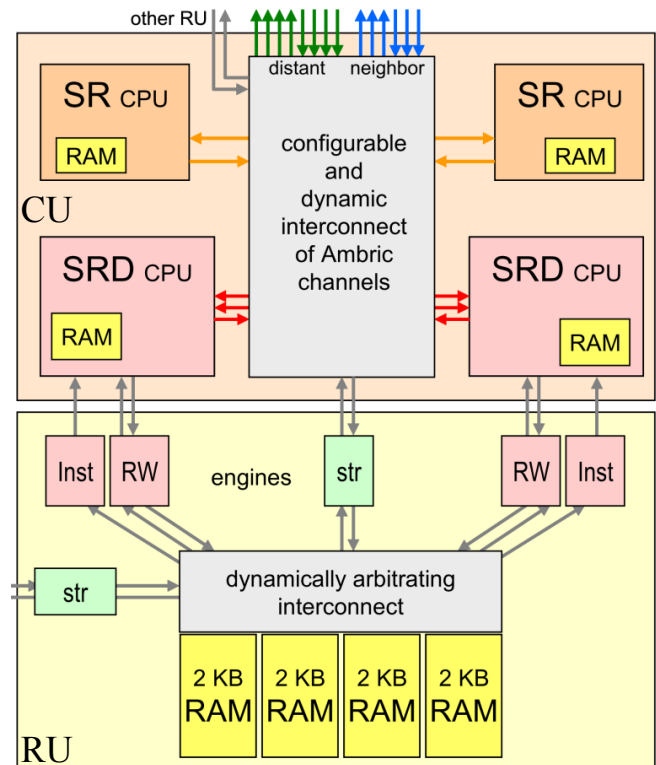


Figure 1. Ambric AM2000 half-bric [6]

Two of the engines (Inst) are dedicated to providing overflow instruction memory space, and two (RW) provide data access for the main processors within the bric. The other two engines (str) are stream engines, one connected to the internal bric switchbox and the other to the network outside the bric. The data access engines can emulate a fixed set of memory types such as FIFOs, regular random access, and combinations of the two where reads use one mode and writes use another. Additionally the stream engines can accept a request for a block of memory and reply with a stream.

The Compute Units (CUs) on the top are made of 2 simple Streaming RISC (SR) processors and 2 Streaming RISC processors with DSP extensions (SRD). Each SRD has a 256-word memory for data and the word-sized instructions. The SRDs are significantly more capable than the SRs, with three execution units. Each SRD has instruction and data memory engines dedicated to it within the RU. The SR processors are intended for simple tasks in support of the SRDs. Each SR has a small 64-word memory for data and half-word sized instructions. The smaller instruction size limits which registers and streams are allowed for several instructions.

The most distinguishing feature in the Ambric architecture is its interconnect. Ambric channels are distributed FIFOs communicating between clock domains in the Globally Asynchronous, Locally Synchronous (GALS) architecture of Ambric. Flow control on these channels is implemented with a *valid* control signal that accompanies the data words between registers, and an *accept* signal routed backwards.

When a register contains data, it will assert *valid* to its successor in the channel to tell it that the data can be captured. If the next register does not already have data, it asserts its *accept* signal. When both *valid* and *accept* signals are high, the receiving register loads itself with the data and de-asserts *accept* as it is currently full. The sending register de-asserts *valid* until it has new data available. Normally, a processor operating on a channel will stall while writing unless *accept* is asserted, and while reading unless *valid* is asserted. This allows the registers of the channel and, by extension, the processing elements to operate asynchronously with respect to one another.

The processors are typically programmed by writing in a subset of Java. Each Java object occupies a single processor, cannot directly affect any other objects, and is automatically compiled into assembly. The Java code communicates by calling specific Java functions to read or write to a channel. The interconnect is defined in a language called aStruct. This instantiates Java or other aStruct objects and defines the connections between them. Breaking a task down into these objects is one of the greatest challenges to developing an application.

### III. LARGE MEMORIES IN FPGAS

Before we discuss how to construct large memories in Ambric, it is useful to consider how FPGA tools handle these needs. FPGA development environments already include the capability to aggregate memories in the architecture. We

tested this with the structure shown in Fig. 2. The clock is not shown for clarity. A counter sends addresses to the memory. The data from the memory is added with the address and the result is fed into an accumulator. The high bit of the accumulator is output. This circuit is designed to keep IO from affecting timing, and to remain constant in logic use and speed across memory sizes. The grey pipelining registers may be placed at locations A, B, and C to assist logic and data movement across the FPGA.

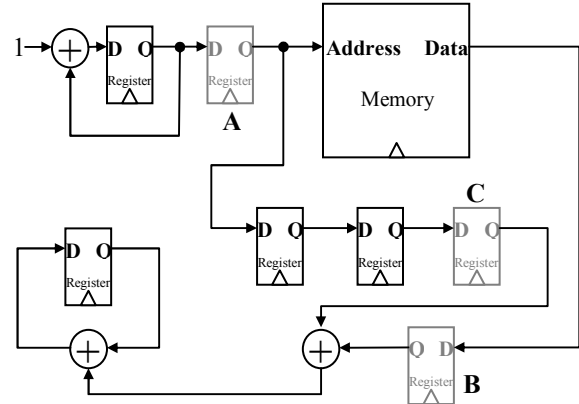


Figure 2. FPGA memory test logic

To characterize the tool's ability to aggregate small memories into larger ones, we targeted an EP3C80 Cyclone III in Quartus [7]. This FPGA has 81,264 LEs and 2,745 Kbits of memory, divided into 305 M9K memory blocks. While this FPGA has only half as much memory as the AM2045, it does occupy a similar price point. We varied memory size from one to 304 M9K blocks, as we ran into a placement error at 305. We tracked logic usage and speed as reported by Quartus. We assumed that using one M9K required no extra logic blocks and therefore all blocks used were from the test circuit. We subtracted this from the remainder of the data points and the results are shown in Fig. 3.

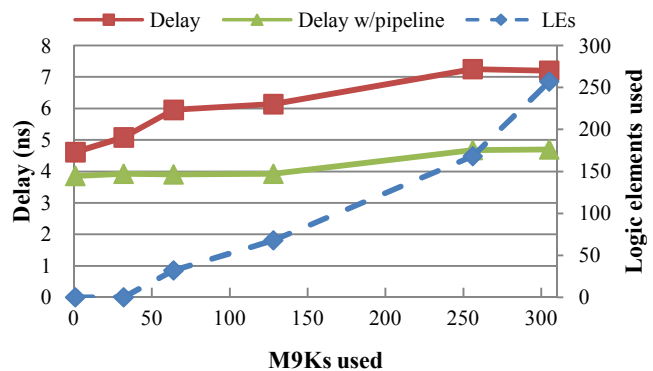


Figure 3. Circuit delay and FPGA resources used

At 32 memory blocks used, no additional combinational logic was used. As our datapath was 32 bits wide, the memory was most likely being divided into individual bits, so no computation or multiplexing was required. After that, there is a small amount of additional logic to select the

memory and route the signal to the output. With increasing memory size, speed decreased as signals needed to traverse the entire FPGA. With the additional pipelining registers, using all the memories was as fast as using only one memory without the pipelining and only 20% slower than using one memory with the additional pipelining. Additionally, far less than 1% of the total logic available on the 80K gate device is used for even the largest memories. This shows that implementing large memories is inexpensive in terms of logic and speed.

#### IV. BUILDING LARGE MEMORIES ON AMBRIC

Developing large memories on Ambric has all of the challenges that developing on Ambric normally entails. For example, tasks must be broken down into smaller blocks to be placed upon individual processors [2]. Efficiency is dependent on balancing the size of these blocks, and pipeline throughput is only as high as the slowest block can maintain. Given that we are processing on streams of data, we will optimize for pipeline throughput (bandwidth) as our first priority.

With the goal of developing a generic memory interface, very little is defined for us already. Given the different engines for accessing a memory, we needed to choose the access mechanism. We also needed to come up with a network that would bring our request to the proper memory and deliver the data. In order for bandwidth to be greater than 1 / latency, the network needs to allow multiple outstanding requests, while ensuring transactions occurred in the proper order.

##### A. Java vs Assembly

An application developer new to hardware design may find Ambric’s Java environment friendlier than the more traditional HDLs on FPGAs. However, design effort savings derived from high level programming usually comes at a cost in implementation efficiency. The stream processing paradigm used in Ambric exacerbates this effect because it is poorly understood by developers and the compilers. Even when writing at a high level, it is useful to keep in mind low-level capabilities.

An example of Ambric Java code is in Fig. 4. This code from our implementation reads in data from four channels and sends the bitwise OR out. Line 3 reads in data from the first channel. Lines 4 and 5 read in data from the next channels and OR them with the current value. Line 6 reads in from the last channel, ORs with the running total, and sends the result out.

```

1 int val;
2 while(true){
3     val = down0.readInt();
4     val = val | down1.readInt();
5     val = val | down2.readInt();
6     up.writeDE0(val | down3.readInt());
7 }

```

Figure 4. Ambric Java example

Compiling this code targeting the simpler SR processors, we end up with the assembly in Fig. 5. Ambric instructions can have multiple destinations; *sink* is a destination that discards all data. Lines 1, 2, 4, and 6 read in data from the incoming channels. Lines 3, 5, and 7 OR the data from the channels together. Line 9 sends the result out. Line 8 jumps back to the start. The jump is not the last instruction because jumps and branches take a cycle to take effect. Therefore, there is a delay slot afterwards that is executed under all conditions; the compiler is smart enough to take advantage of this. This appears to be a reasonable compilation for a RISC processor and should take 9 cycles to execute a single iteration.

```

_WT_23:
1 mov down0, r2, sink
2 mov down1, r1, sink
3 or r2, r1, r2, sink
4 mov down2, r1, sink
5 or r2, r1, r2, sink
6 mov down3, r1, sink
7 or r2, r1, r1, sink
8 jump code_ptr(_WT_23)
9 mov r1, sink, up, de0 //delay slot

```

Figure 5. Compiler assembly example

However, even the SR processors are richer than simple RISC processors so there is an opportunity for optimization. Channels can be accessed almost as easily as the register file, multiple destinations are allowed, and there are advanced looping constructs. Optimized assembly is shown in Fig. 6. Lines {2, 3} and {4, 5} from the compiler output have been compressed into lines 3 and 4, respectively in our hand-tuned code. Similarly, compiler lines 6, 7, and 9 have been compressed into a single line, 5, which reads in a channel, ORs it, and sends out the result. Finally, we have eliminated the need for the jump instruction by using the zero-overhead loop construct in line 1, resulting in 4 cycles per iteration.

```

1 loopalways END
2 mov down0, r1, sink
3 or down1, r1, sink
4 or down2, r1, sink
END:
5 or down3, r1, sink, up, deflag

```

Figure 6. Manual assembly example

If this was the limiting step in our pipeline, hand tuning resulted in a 2.25x throughput increase. Alternatively, we could have written Java that processes only two incoming streams. We would need two processors to watch all four streams, and then a third processor to combine the results of the first two. Each one of these stages would require 5 cycles per iteration, thus we would need three processors instead of one and we would still only be 1.8x as fast as the original implementation. The key lesson is that the compiler does not optimize sufficiently in some cases and the critical step should be written directly in assembly language.

### B. Ambric branches

The zero overhead loop mechanism is designed for implementing FOR loops and cannot be used to process our packets. As discussed previously, branches have a delay slot following them. Additionally, the traditional ALU flags like *zero* and *overflow* must be sent to the IP/branching mechanism manually through an Ambric channel. This channel has a depth of 2, so any branch that is dependent on a test in the previous instruction will be forced to stall for an additional cycle for the test results to arrive. This means that, including the conditional branch and its delay slot, 4 cycles will be used.

For our purposes, there are two decisions that need to be made for each memory transaction: (1) Is this packet a read or a write? (2) Is this packet targeting this memory block, or some other block? If we handled both of these decisions as branches on a single processor, the best we were able to achieve was 8 cycles per memory transaction. To achieve better performance, we must reduce branching in each processor, or replace some operations with non-branching alternatives.

### C. Ambric memory protocol

Ambric has a memory access protocol called TMS for accessing the streaming memory engines and external DDR2 memory. As our larger memory is an intermediate step between off-chip memory and the current 8KB maximum that is accessible through a stream engine, let us start with it. A TMS data packet is shown in Fig. 7. The Ambric channels contain a special 33<sup>rd</sup> bit called *de* that accompanies the data word. This bit is intended to help with multi-word packets. The final word of any packet, which is the only word in the case of a single-word packet, will have *de* = 0; all other words in the packet will be accompanied by a *de* bit set to 1.

de	31	28	27	16	15	0
<b>Raw data: single word</b>						
0	data[31:0]					
<b>Raw data: multiple words</b>						
1	data[31:0]					
	...					
1	data[31:0]					
0	data[31:0]					

Figure 7. TMS data [6]

Read request packets are shown in Fig. 8 and write request packets are shown in Fig. 9. Both functions have a “long” addressing mode for when 16-bit addressing is insufficient. The top nibble of the first word specifies the type of packet and the request type. Bits 16-27 specify the requested length. For reads, the memory controller will reply with a stream of *length* words starting with the data at *address*. For writes, the controller will start writing at *address* and write *length* words.

de	31	28	27	16	15	0
<b>Memory read: 16-bit address</b>						
0	0xA	length[11:0]	address[15:0]			
<b>Memory read long: 48-bit address</b>						
1	0xB	length[11:0]	address[47:32]			
0	address[31:0]					

Figure 8. TMS read packets [6]

de	31	28	27	16	15	0
<b>Memory write: 16-bit address</b>						
1	0x8	length[11:0]	address[15:0]			
1	data[31:0]					
	...					
1	data[31:0]					
0	data[31:0]					
<b>Memory write long: 48-bit address</b>						
1	0x9	length[11:0]	address[47:32]			
1	address[31:0]					
1	data[31:0]					
	...					
1	data[31:0]					
0	data[31:0]					

Figure 9. TMS write packets [6]

While we would like to remain compatible with TMS packets, a small modification can increase performance greatly. A read request for multiple words may be larger than a single memory. Supporting these operations would require significant computation. If we limit *length* to 1, then we can reuse the *length* bits as part of the address. This allows us to address all of the on-chip memory of the AM2045 in a flat address space without needing to support long reads or writes.

The resulting packets are shown in Fig. 10. The 11 bits in the low portion of the address can be sent directly to the memories. The high portion of the address will be used to select the memory.

Notice in Fig. 10 that memory reads are now 1 word long, and memory writes are two words long. This means that the *de* for reads is 0, and for writes *de* is 1. This is particularly helpful for our system, since the Ambric processors can branch on the *de* bit without a test instruction, and without the 1-cycle latency between test and branch operation. Thus, a branch on *de* after a word read costs 2 cycles (branch instruction and subsequent delay slot) instead of 4.

de	31	28	27	11	10	0
<b>Memory read: 28-bit address</b>						
0	0xA	address high[16:0]		address low[10:0]		
<b>Memory write: 28-bit address</b>						
1	0x8	address high[16:0]		address low[10:0]		
0	data[31:0]					

Figure 10. Our read and write packets

### D. Memory access

There are two ways of accessing the memories for data. Each RU has two stream engines wired to the interconnect and RW engines dedicated to each of the SRDs in the RU’s partner CU. Using the SRD is more flexible; however, there is a latency of 4 cycles for a SRD to send the read address to its RW engine and to read the response. For our initial examination, let us ignore the network that brings requests and delivers data. Reads will be sent to the processor for each memory block that is part of our memory and if that particular memory does not service the requested block the request will be suppressed.

If the SRD had to do this processing on the packet, it would be beneficial to be able to do it in parallel with the memory access; this leads to the need to do non-blocking

reads. Non-blocking reads require branching dependent on whether or not the read was successful, increasing the processing load. Therefore, to maintain pipeline speed, we will need to move any processing off to another processor. In either scenario, the SRD is not even doing any work for the application and is simply providing access to the memory. Even in a barely used memory network if performance was unimportant, it would be difficult to reuse the memory processors for application tasks in a generic manner. Thus, a stream engine is just as useful, and will save the SRDs for application code.

When dealing with the memory stream engines, we must still pay the memory access latency for each memory transaction. However, the stream engines have separate input and response streams so the processor sending the address can be different from the processor receiving the data. This allows us to build a feed-forward network that uses pipelining to hide the memory latency. We can also split the processing decisions between these two processors.

Using the network shown in Fig. 11, the address processor passes all reads to the memory after converting it to a legitimate TMS packet and sends the data processor the result of the test on the memory select. This will tell the downstream data processor if the read targeted this memory block and the data should be forwarded or if the read targeted a different block and the data should be suppressed. Pseudo-code for a read-only implementation is shown in Fig. 12. Each line number indicates a separate assembly instruction; this code only requires 4 cycles to execute.

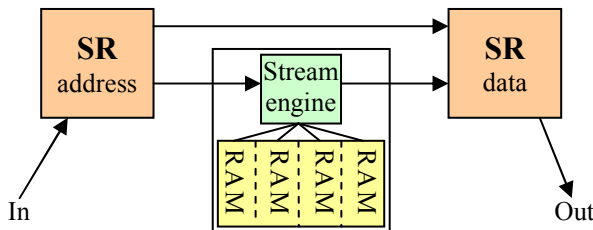


Figure 11. Address processor, stream engine, and data processor network

```

1 Read the input channel.
2 XOR the high address bits with our address.
3 Mask off [27:11] high address portion and
  send the read command to the stream engine.
4 Zero out everything from the XOR result but
  the high address bits. Send result to data
  processor.
```

Figure 12. Address processor read pseudocode

Fig. 13 shows code for the address processor for a read/write memory. We add a branch on *de* to either execute the read or write code, based on the top bit of the incoming word. We fill the delay slot with the XOR to check if this is the correct memory block, since this is used in both the read and write case. The rest of the read code is the same as the read-only version. The write code checks the address, and only writes when this block is the proper target. We need to consume the word to be written even if not actually writing, so this can go into the delay slot (line 9). Note that, when writing, nothing is sent on to the data processor, since the

memories will not respond and the data processors do not need to generate a return value.

```

1 Set up zero-overhead loop to cycle from
  instructions 2-6.
2 Read the input channel.
3 Branch to write code (line 7) if de is set.
4 XOR high addr bits w/our addr. //Delay slot
5 Mask off [27:11] high address portion and
  send the read command to the stream engine.
6 Zero out everything from the XOR result but
  the high address bits. Send result to data
  processor. The zero overhead jumps back up.
Write code:
7 Zero out everything but the high address
  portion. Send results to the flag channel.
8 Jump back into our read loop (line 2) if
  flag indicates this was not our packet.
9 Read input to consume data. //Delay slot
10 Mask off [27:11] from original packet and
  send the write command to the stream
  engine.
11 Jump back into read loop (line 2).
12 Send write data to stream engine. //Delay
  slot
```

Figure 13. Address processor read/write pseudocode

The data processor reads the address test from the address processor. It then passes on the data response if the address matched and a null synchronization packet otherwise (Fig. 14). While doable with a branch, line 2 shows how a mux instruction can be used instead. The mux replaces the test and the branch, so the resulting code only requires 3 cycles. We will explain synchronization packets in the next section.

```

1 Read from the address processor.
2 If the address processor sent 0, load 1's into
  the mask, otherwise load 0's.
3 Read the channel from the stream engine. AND
  the mask with the result. Send the result out.
```

Figure 14. Data processor pseudocode

### E. Trees and synchronization

Large memories must be composed of the smaller memories spread across the chip. Since each processing element can handle a limited number of channels, this implies some sort of network. Another requirement is that transactions are completed in order. A request for memory address *X* returns the data at address *Y* or our computation will be incorrect. Alternately, a write followed by a read to the same address must return the recently written value.

Keeping transactions in order is easier if the latency to each of the memories is equal. This will also minimize the worst-case latency. The best structure for this is a tree (Fig. 15) which also minimizes the number of processors required for a given fanout. However, without the ability for multiple requests to be in the tree at once, bandwidth will be as bad as  $1/\text{latency}$ .

To solve this problem, our tree must be pipelined. Doing this on a tree processing both requests and resulting data means each node cannot stall after a read request before the data arrives. It is possible to do a non-blocking channel read,

but the processors would have to add extra processing to determine whether or not a packet was actually received. It is easier to duplicate the tree into separate fanout and fanin portions as shown in Fig. 16.

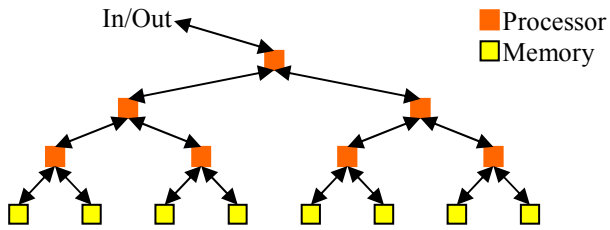


Figure 15. Bi-directional tree

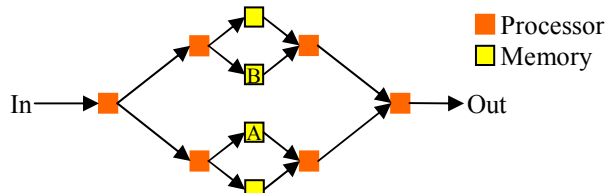


Figure 16. Unidirectional tree

With two separate trees as in Fig. 16, requests to different memories may be serviced out of order. If there are several writes to memory A followed by a read, that read will have to wait for the writes to complete first. Since these operations all follow the same path, they will remain in the proper order. However, a subsequent read request to memory B could return before the read to A because there are no other requests pending on that path, and the fanin tree will simply pass the response up. Even if requests were perfectly distributed, there is no guarantee that requests are returned in the proper order through the GALS network of Ambric.

A potential solution to this problem is shown in Fig. 17. As the fanout tree makes routing decisions, these are communicated to the fanin network through the cross channels between from the fanout tree to the fanin tree. The fanin processor would know which of its inputs to relay first. The processor then waits for a response from that input. Subsequent requests on the cross channel, or read results on other inputs, will just queue up on the appropriate channel. Since the cross channel keeps a sequential list of reads, and each input channel keeps read results in order, this guarantees that the results will be merged together in the proper order.

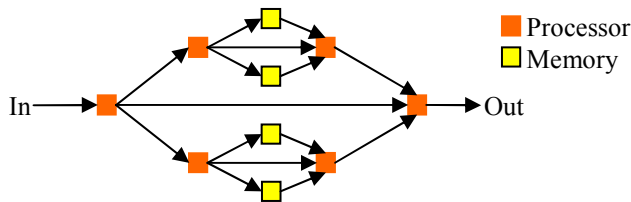


Figure 17. Unidirectional tree with crosslinks

Unfortunately, the routing decisions required in the fanout tree are at least as complex as those of the address processors in the previous section. As the fanout processors

need to make the routing decisions, they must do the entire address compare themselves. This will add several cycles of delay.

Since we do not want to make decisions within the fanout tree, we need another method of maintaining transaction order. If all transactions go to all memories, we do not need to worry about the requests getting out of order, as the FIFO channels will keep their data in order. All that the fanout processors will be doing is copying their inputs to each of their outputs. This particular situation was considered in the design of Ambric and special-purpose hardware was placed into the interconnect to duplicate the data in a stream into two streams, allowing us to save on processors used by building a tree of duplicators instead.

There is still a chance for the data replies from the memories to get out of order. The solution to this problem is to ensure that there is always a response, even when the read is for a different memory. Then the FIFOs will keep the data ordered for reads as well. This requires the fanin tree to read in an input word from each input and combine the results. If all inputs are sync packets, the fanin node will produce a sync packet; it will send the data response if one of the inputs is a data response.

We would like to do this without branches if possible. A good way to implement this is to use 0's as synchronization packets. The fanin processor can then just OR its inputs together to generate the proper output. This means that it only requires one cycle to process each input packet. Since the throughput is limited by our address processor, which takes 5 cycles in a read-write implementation, we can use 4:1 compressors which only take 4 cycles (assembly in Fig. 6) in our fanin tree without reducing bandwidth as shown in Fig. 18.

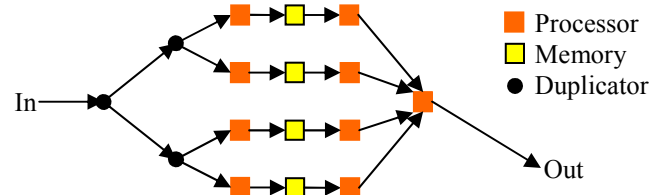


Figure 18. Broadcast tree with hardware stream duplicators

#### F. Latency and hardware assignment

Most of our processor usage will be in the leaves of our tree, with two SRs hooked to a single stream engine. The logical placement is to place all of these elements within the same half-bric. However, this creates an unintended bottleneck. Two processors in the same half-bric that communicate with each other have a channel with no extra registers on it. This means that when the data SR is waiting for a read's data to emerge from the stream engine it is not reading another memory select test from the address SR and the address SR cannot begin a second read. Therefore, although logically this structure should be pipelined, in practice the two SRs are in lock step, and cannot hide the latency of the memory access. To fix this we deliberately force the two SRs to go to different half-brics by placing a minimum FIFO length on that SR->SR link of 1.

For large memories, where the fanin and fanout trees may need to span much of the chip, imbalances in different path lengths might cause similar problems, with fast paths that go through fewer registers causing stalls on previous processors when there are multiple requests outstanding. However, since our system takes 5 cycles per memory transaction, there is only one word every 5 cycles. Each register on a given path can delay for up to 4 more cycles, so a 4-stage mismatch in path length would cause no problems as long as at least 1 register is on the path. Thus, this possible bottleneck did not turn out to be a concern in practice for our memory structures.

## V. USAGE

Using this architecture, we built memories ranging from 1 to 32 RUs. When only using one memory, which does not use our network, the stream engine can process one word per cycle. However, for our network, read bandwidth is constant for all other sizes at one word every five cycles. For comparison, in our PET application example [2], the stage that requires large memories has a throughput of 11 cycles per input, which is only half our network’s bandwidth.

We tracked read latency and the number of processors used and plotted the result in Fig. 19. The number of processors used is approximately linear, as one address and one data processor is required per memory and these make up the majority of the processors required. The actual network does not grow as fast and uses a relatively small number of processors. There is a fairly large latency cost when initially implementing the tree, and slower growth afterwards. Tree depth only grows logarithmically with memory count, but the registering in the interconnect as the memory spreads out forces latency to grow faster than logarithmically.

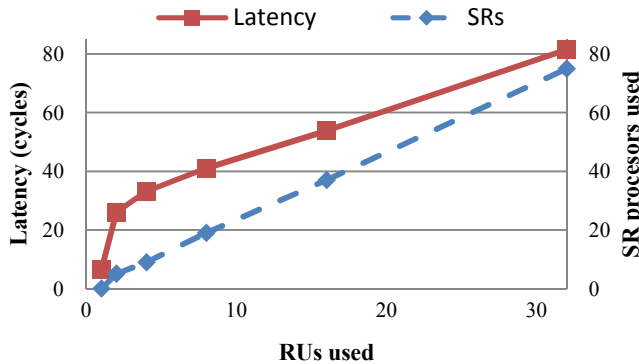


Figure 19. Processor utilization and latency vs memory size

It is important to observe that the AM2045 uses a higher percentage of its computational resources than the FPGA uses for a similar percentage of memory. Using all the memories on the FPGA used less than 1% of the logic, but on Ambric, using all the memories would require over half of the processors. In fact, some processing would need to be moved to SRDs. FPGAs, with their bit-oriented computation, are far more efficient at the match and compare tasks. Single-bit signals can be defined where required, eliminating

the need to multiplex control and data. It is also much easier to maintain synchronization with a global clock on the FPGA.

Similarly, the latency on Ambric is also quite high. On the FPGA, only two cycles of additional latency were added in the pipelined version to remedy all of the speed loss of building the memories. On Ambric, in addition to processing, the distributed FIFOs and routes also require several cycles so latency grows much more. FPGA cycles are quite different from an MPPA though. On a FPGA, computation steps that do not require a full clock can have several executed in one cycle. Long distance signals will only slightly slow the maximum clock instead of adding additional cycles. However, it is worth noting that the 130nm AM2045 runs at 350MHz, while our FPGA test circuit only ran at a maximum of 250MHz on a 65nm FPGA and accessed less memory overall.

## VI. ARCHITECTURAL ENHANCEMENT

The primary limit on performance of these larger memories is packet processing, particularly the top address bits and the Read/Write signal. FPGAs have the decided advantage by offering efficient bit operations. Additionally, the raw number of processors required may make it difficult for some applications to coexist with the memories. Indeed, since there are two SR processors per RU on the chip, it is not possible to access all the memory on the chip without using a portion of the larger SRD processors. Given these limitations, it is important to consider what future architectures could do to help.

Our processor count has already been reduced by using the hardware duplicators built into the interconnect as our fanout network. A logical extension would be to add a similar merge capability and eliminate the fanin network too. The remaining address and data processors could be replaced by adding some FPGA-like bitwise capabilities. This was suggested for the CGRA space in [9], but synchronization of the separate channel within the GALs framework may prove difficult. This could be remedied using a purpose-built memory network, but it is likely this resource fragmentation would reduce the overall efficiency of this option.

Already, the memory stream engine processors have helped us pipeline the packet processing. As they already have several access modes, they could be enhanced to support a per-memory ID so they can be accessed over a broadcast bus like our existing network. The stream engine would ignore writes that did not have the same address bits as its ID. For reads, the stream engine could put out a sync packet for IDs that did not match. In this way, all of the packet processing that now consumes SRs could be migrated to the stream engine itself. The stream engines could be modified in other ways, but due to the potentially large number of memories, simple FPGA-like techniques are likely to be better than processor cache or virtual memory techniques. Future work is required to determine which of these mechanisms is worthwhile.

## VII. CONCLUSION

In this paper, we provided support for memories larger than the hardware blocks in Ambric. This required several major steps:

- Direct use of assembly language and elimination of branch instructions where possible.
- Splitting the fanin and fanout network to produce a pipelined, feed-forward network that also can leverage Ambric's built-in duplication hardware.
- Broadcasting reads and writes to all memories and the use of synchronization packets during reads to maintain memory transaction ordering.
- Splitting decisions between multiple leaf processors to reduce per-stage delay, and overlap packet processing with memory access latencies.

When combined this provides scalable, efficient large-scale memory support within the Ambric array. It provides a 4-cycle ROM, and 5-cycle RAM memory transaction issue time.

In comparison to FPGAs, we demonstrated that, while Ambric can support large memories, word-oriented processors are a poor match to the bit-oriented address processing required by this task. However, we proposed simple architectural changes to the stream engine processors and the addition of a reduction network to handle merges in the fanin network that could largely eliminate these overheads. These, or similar enhancements, should be considered for future word-oriented accelerators as composing memories together will be necessary for many applications.

Beyond just supporting large memories, these large memories demonstrate the use of the Ambric hardware itself. Like an FPGA, the Ambric system allows for the flexible creation of complex, interconnected processing systems. However, the processor nature of the individual elements

means that care must be taken to best split the processing across multiple processors, and to find the best implementation of that computation within the Ambric processor's instruction set.

## ACKNOWLEDGMENT

This work was supported by Department of Energy grant #DE-FG52-06NA27507 and NSF grant #CCF0702621.

## REFERENCES

- [1] Ebeling, C.; Fisher, C.; Guanbin Xing; Manyuan Shen; Hui Liu; , "Implementing an OFDM receiver on the RaPiD reconfigurable architecture," *Computers, IEEE Transactions on*, vol.53, no.11, pp. 1436- 1448, Nov. 2004
- [2] Haselman, M.; Johnson-Williams, N.; Jerde, C.; Kim, M.; Hauck, S.; Lewellen, T.K.; Miyaoka, R.; , "FPGA vs. MPPA for Positron Emission Tomography pulse processing," *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pp.231-238, 9-11 Dec. 2009
- [3] J. Xu, N. Subramanian, S. Hauck, A. Alessio, "Impulse C vs. VHDL for Accelerating Tomographic Reconstruction", *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2010, in press.
- [4] M. Butts, A.M. Jones, P. Wasson, "A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing," *International Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 2007, pp. 55-64.
- [5] Mike Butts, "Synchronization through Communication in a Massively Parallel Processor Array," *IEEE Micro*, pp. 32-40, September/October, 2007
- [6] Ambric, Inc., "Am2000 Family Architecture Reference," July 2008
- [7] Altera, Inc, "Quartus II Subscription Edition Software," Mar 2010; <http://www.altera.com/products/software/quartus-ii/subscription-edition/>.
- [8] Ambric, Inc., "Am2000 Family Instruction Set Reference," January 2008
- [9] Mirsky, E.; DeHon, A.; , "MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources," *FPGAs for Custom Computing Machines*, 1996. Proceedings. IEEE Symposium on, pp.157-166, 17-19 Apr 1996