SUPPLEMENTARY MATERIAL

We devote the supplementary material section to a more detailed description of the arbiter, which was summarized in Section 3.D.

Given a list of 16 memory access requests generated by the cores, the role of the arbiter is to determine whether multiple cores request data from different addresses of the same bank. There are a few properties that we want the arbiter to satisfy:

- **Fairness:** The arbiter does not always prefer granting the memory access request of one core over another when collisions happen between them so that all FSMI cores can be equally active.
- **One-cycle:** The arbiter must complete all collision checking and memory association in one clock cycle. This constraint ensures that the arbiter does not introduce additional delay for memory access, which reduces the memory bandwidth from the occupancy grid map to the FSMI cores.

Maximizing memory sharing between two read ports for each memory bank complicates the design of the arbiter protocol, introduces significant increase in the critical path of the digital logic, and does not lead to much bandwidth improvement. Thus, we break the arbiter into two identical smaller arbiters called Port 0 and Port 1 Arbiter as shown in Figure 1. Port 0 Arbiter handles memory requests from cores 0 to 7 and service these requests using port 0 of every memory bank. Similarly, Port 1 Arbiter handles memory requests from cores 8 to 15 and service these requests using port 1 of every memory bank. Hence, the complexity of the overall arbiter decreases because each arbiter only needs to handle memory requests from half of the cores and uses only one read port from every memory bank.

The arbiter uses a round robin scheme by keeping track of a priority pointer to ensure fairness among cores which helps to balance the cycles per core and the resulting overall throughput. Figure 2 illustrates this with 3 stages. In stage 1 and 2, the arbiter is constructing a continuous window containing groups of memory requests (from consecutive cores) that can be granted at every cycle. This window always starts at the core with the priority index (core 1). If a memory access collision occurs (core 6 and core 2 try to access the same memory bank with different addresses), memory access priority is given to the core that is closest to the priority index

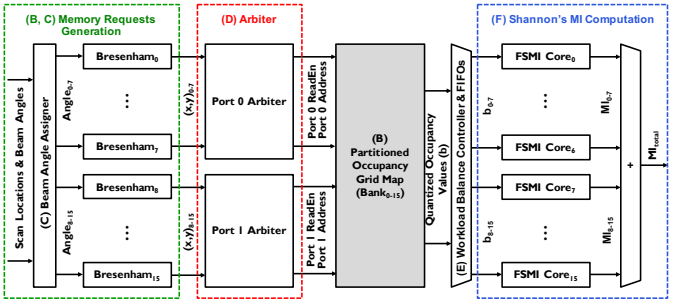


Fig. 1: Overview of the top-level architecture.

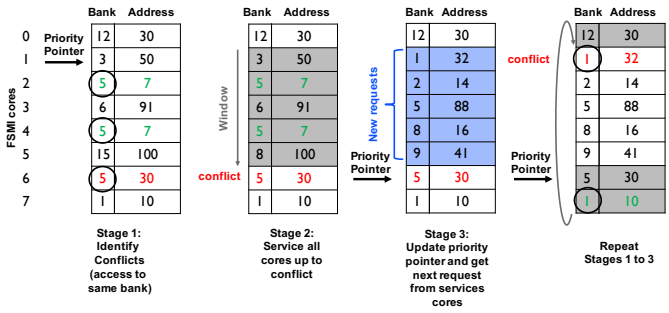


Fig. 2: Overview of the arbiter operation

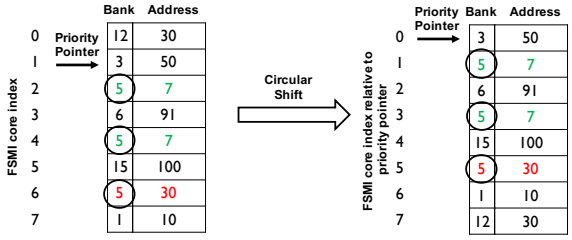


Fig. 3: Step 1 of the arbiter

(core 2). Thus, the window ends at the core before the collision (core 5). Since all memory requests within the window is serviced, the next set of memory requests from all cores within the window replaces the current ones, and the priority index is updated to the core where collision happens (core 6). In all, the process from stage 1 to 3 is repeated every clock cycle until request from all cores are serviced.

One naive approach for determining the location of the memory collision in stage 1 is to check the memory requests one core at a time starting from the priority index. Since stage 1 to 3 has to be completed within a single clock cycle, the critical path of the entire arbiter design scales linearly with the number of cores which lowers the maximum operating clock frequency for the hardware and in turn reduces memory bandwidth to FSMI cores. Thus, an alternative 4-step arbiter architecture is proposed to achieve the same functionality as the arbiter in Figure 2 but with a critical path that logarithmically scales with the number of cores and the number of memory banks.

In step 1 (Figure 3), the arbiter performs a circular shift so that the core with the priority index always occurs at the top of the array. Using the resulting array from step 1, the arbiter then reorganizes the memory access requests by banks in step 2 (Figure 4).

Since each arbiter processes the memory requests from 8 cores and service them using 1 port from every bank, it organizes the memory request into 16 groups. Since each group only contains memory access information for its corresponding banks, a tree architecture used to determine the window size for each bank is shown in step 3 (Figure 6). Each element of the tree consists of five fields: valid (V), address (A), window start (ST), window end (END), and lowest index (L). The field V indicates whether the memory request and

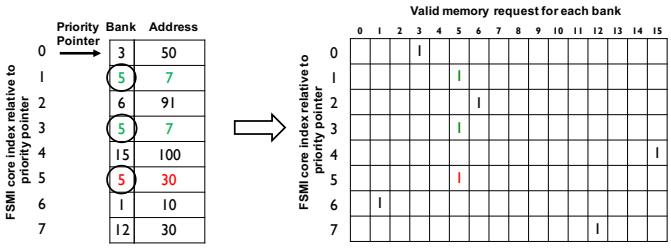


Fig. 4: Step 2 of the arbiter

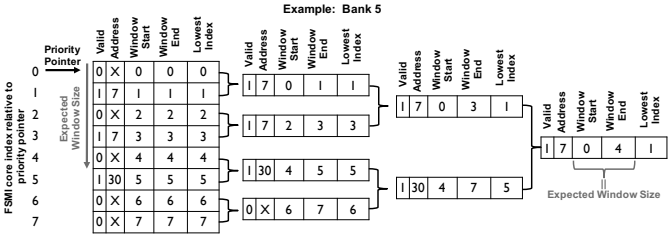


Fig. 5: Tree structure to determine start and end of service window in arbiter for each bank.

address (A) of its corresponding core is valid for a specific bank. The ST and END fields indicate the start and end boundaries of the window. The L field tracks the location of the valid request with the lowest index in its corresponding window. At leafs of the tree, the window size of each element is 1 (ST = END). As the tree converges, the elements and their corresponding windows merge using the the hardware architecture in Figure 6. The final element of tree contains the window size for each bank. An example of the step 3 tree operation for bank 5 is shown in Figure 5.

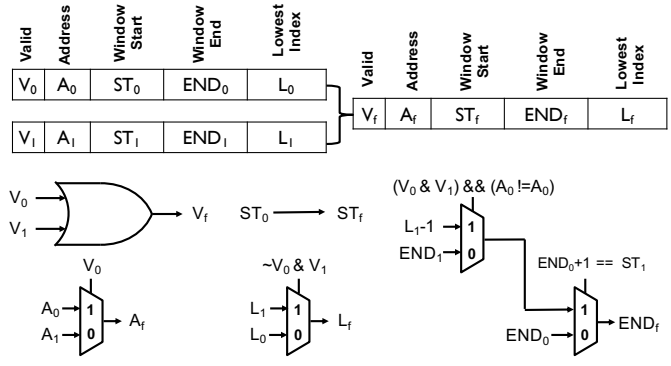
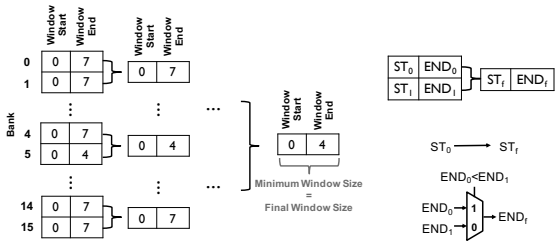


Fig. 6: VLSI architecture for step 3 of the arbiter

Due to the circular shift, the final window size of each bank all starts at the priority index (0 offset) but end at different core index. In step 4, another tree structure is used to determine the end the of the overall window over all banks by finding the smallest offset index. This tree structure with its hardware architecture is shown in Figure 7(a) and Figure 7(b), respectively. In all, the final window size of the proposed arbiter architecture shown in Figure 7(a) matches the expected window size of the naive arbiter in Figure 2 and contains a



(a) Step 4 of the arbiter (b) architecture

Fig. 7: Operation and hardware architecture for step 4 of the arbiter

critical path that scales logarithmically with the number of banks and cores.