DX-Gt: Memory Management and Crossbar Switch Generator for Multiprocessor System-on-a-Chip

Mohamed A. Shalan, Eung S. Shin and Vincent J. Mooney III

Georgia Institute of Technology
School of Electrical and Computer Engineering
777 Atlantic Dr.
Atlanta, Georgia 30332, USA.
{shalan,eung,mooney}@ece.gatech.edu

Abstract: As the number of transistors on a single chip increases rapidly, there is a productivity gap between the increasing number of available transistors and the design time. One solution to reduce this productivity gap is to increase the use of Intellectual Property (IP) cores. However, an IP core should be customized/configured before being used in a system different than the one for which it was designed. Thus, to reconfigure the IP core, either an engineer must spend significant effort altering the core by hand or else an enhanced CAD tool (IP generator) can automatically configure and customize the core according to the customer specifications.
In this paper, we present an SoCDMMU-crossbar (Xbar) switch Generator (DX-Gt) tool that automatically configures the memory and bus subsystems of a multiprocessor SoC to meet design constraints. Specifically, we show the area results of configurations for a crossbar switch plus SoCDMMU for 2x2 up to 12x12. The first contribution of our paper is to provide a method and tool for automatic generation of a Dynamic Memory Management Unit for an SoC. The second contribution is the automatic generation of a crossbar switch. Both hardware units are generated as synthesizable Verilog HDL code at the RTL level.

I. Introduction

In a year or so integrated circuits will have close to one billion transistors on a single chip [1]. Such chips give designers the opportunity to integrate many functionalities, each of which used to be implemented on different chips, into the same chip. In other words, a complete system that used to be implemented on a printed circuit board will be integrated into a single chip; i.e., System-on-a-Chip (SoC). One opportunity for such chips is building a multiprocessor SoC that has multiple processors of different types, large memory, custom digital logic and interfaces.

We predict that in next five years multiprocessor SoCs will be dominated by designs with four to eight processors and on-chip DRAM of 16Mbytes to 128Mbytes. In such multiprocessor SoCs, multiple buses may be desired to provide multiple communication channels to each processor so that communication among processors does not become a system bottleneck.

Designers of a multiprocessor SoC with multiple processors and large on-chip memory must decide whether the allocation of the on-chip memory among the on-chip processors will be dynamic or static. Obviously, dynamic allocation is a desirable feature; however, software implementation of dynamic memory management is not usually deterministic and typically consumes thousands of the processor clock cycles per allocation. A hardware approach in the form of a hardware *intellectual property* core was introduced to provide a multiprocessor SoC with dynamic yet deterministic memory management capabilities [2].

As the number of transistors on a single chip increases rapidly, there is a productivity gap between the increasing number of available transistors and the design time. One solution to reduce this productivity gap is to increase the reusability of Intellectual Property (IP) cores. However, an IP core should be customized/configured before being used in a system different than the one for which it was designed. Thus, to reconfigure the IP core, either an engineer must spend significant effort altering the core by hand or else an enhanced CAD tool (IP generator) can automatically configure and customize the core according to the customer specifications. For example, memory and I/O generators by Artisian [10] and processor generators by Tensilica [11] and ARC [12] supply application specific IP cores that can be highly tuned for specific applications.

To the best of the authors' knowledge, this paper presents the first published work on automatic generation of a crossbar (Xbar) switch coupled with a memory management unit. More specifically, reconfiguring an Xbar is more than reconfiguring bus parameters such as address bus width and data bus width. An MxN Xbar must be configured to support an exact number of masters (processors), M, and an exact number of slaves (memory), N. Thus, to reconfigure an MxN Xbar, one must generate (i) an arbiter able to handle the exact number of requests in an Xbar and (ii) wires (address bus, data bus and some control lines) between masters and slaves. One evidence of the need for multiple buses can be found in CoreConnect [21] which is an on-chip bus from IBM. CoreConnect provides synthesizable Verilog for eight masters; the CoreConnect bus presumably shows good performance with up to eight masters. In other words, the bus itself can be the bottleneck if the number of masters exceeds eight. Thus, a bus, an Xbar and/or arbiters must be generated or reconfigured to support an exact number of masters and an exact number of slaves specified by the user. CoreConnect requires interfaces for masters and slaves.

Our paper is focused on the design of a CAD tool for the generation of a memory management unit and an MxN Xbar

switch; we name this tool the **D**ynamic memory management unit-**X**bar **G**enera**t**or (DX-Gt). The first contribution of our paper is to provide an automatic generation of SoC Dynamic Memory Management Unit (SoCDMMU). The second contribution is the automatic generation of a crossbar (Xbar) switch. The Xbar generation is also integrated with an arbiter generation tool [4]. Both SoCDMMU and Xbar are generated in Verilog Hardware Description Language (HDL) code synthesizable at the RTL level. Our generated IPs (SoCDMMU and Xbar) require interfaces for masters and slaves. We assume that the effort to connect a new IP core to the custom IP generated by DX-Gt is almost the same as that of IP based design such as adding other IP cores to CoreConnect.

The paper is organized as follows. First, Section 2 gives an overview of the related work. Section 3 explains the target architecture for DX-Gt. Section 4 shows how DX-Gt works. Section 5 gives synthesis results of our generated SoCDMMU and Xbar. Finally, we conclude our paper in Section 6.

II. Related Work

A. *Crossbar switch design*

There are few approaches to reduce design time for an SoC crossbar switch. Mai *et al*. propose reconfigurable crossbar switch and memory blocks [7]. In [7], two integer clusters and one floating point cluster are connected to sixteen 8Kbyte SRAMs via the crossbar switch with the re-configurability feature. However, the authors do not give details about their crossbar switch design. Compared with [7], which appears to be designed by hand, our MxN crossbar (Xbar) switch is automatically generated with bus parameters specified by a user to support exact number of masters and slaves. Also, our generated Xbar is synthesizable at the RTL level resulting in a reduction in design time. Thus, from the above discussion, DX-Gt provides the first automated approach to Xbar switch generation.

B. *Hardware based memory management synthesis*

Most previous research in memory management for embedded systems has focused either on static allocation and how to synthesize memory hierarchies for an SoC [22] or on designing hardware that accelerates the memory management function execution.

The literature shows that a hardware implementation of a simple buddy allocator was first proposed by Knowlton [14] [17]. It is a simple and fast buddy allocator that can allocate memory blocks whose sizes are a power of two; hence, the allocator suffers from internal and external fragmentation. Puttkamer introduced a hardware buddy allocator that does not suffer from internal fragmentation [13]. Chang and Gehringer propose a modified hardware-based buddy system which eliminates internal fragmentation and has a constant execution time [16]. Chang *et al*. have implemented the *malloc()*, *realloc()* and *free()* C-Language functions in hardware [15][16]. Also, they propose a hardware extension to be a part of future microprocessors to accelerate dynamic memory management [15]. Although Chang's modified binary buddy allocator eliminates internal fragmentation, the allocator can only detect a free memory block chunk if it starts at an address which is power of two. This problem is called the *blind spot* problem. To overcome the *blind spot* problem, Cam *et al.* propose a hardware buddy allocator that detects any available free block of requested size and eliminates internal fragmentation [14].

The previous research focuses only on the hardware implementation of specific functionality (e.g., allocation or de-allocation) but never discusses in detail how to integrate these functionalities into a system nor how to synthesize/configure the hardware for a specific system. Moreover, the use of these hardware allocators for multiprocessor systems has not been addressed. Also, all of the hardware allocators introduced are not suitable for small memory allocations which make them impractical for most real world applications[1].

Our SoCDMMU is synthesizable and has been integrated into a practical system including porting SoCDMMU functionality to an RTOS (so that the user can access SoCDMMU functionality using standard software memory management APIs) [5]. Also, DX-Gt can configure and optimize the SoCDMMU to suit a specific system. In this way, DX-Gt automates the customization and the generation of the hardware memory management functionalities.

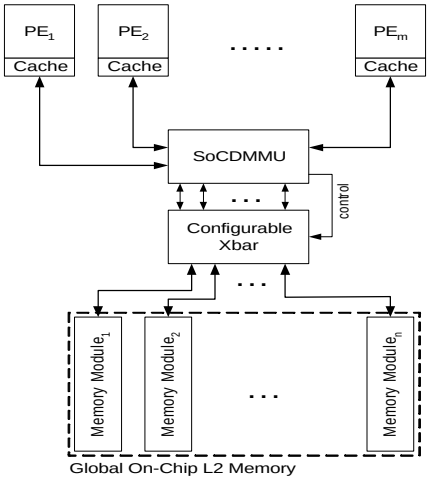

Figure 1. The SoC Target Architecture

III. Target Architecture

As shown in Figure 1, our target SoC architecture consists of multiple Processing Elements (PEs) of various types (i.e.,

[1] See pp. 7-8 of [30] for more details.

general purpose processors, domain-specific CPUs such as DSPs, and custom hardware), large configurable global on-chip memory blocks and the SoC Dynamic Memory Management Unit (SoCDMMU) to manage the memory allocation and deallocation among the PEs. An SoCDMMU remaps processor addresses (virtual addresses) to physical addresses which are passed to Level 2 (L2) memory via an MxN Xbar. To achieve the maximum concurrent transfers between processors and memory blocks, N should be equal to M. The memory coherency problem due to concurrent accesses of global memory blocks is reduced by using the SoCDMMU [2]. The combination of the SoCDMMU and the Xbar for a multiprocessor SoC showed an overall speedup of 4.4X during application transition time when compared to a fully shared memory system with the same memory organization and number of processors [5].

Our SoC configuration tool can generate an architecture like that of Figure 1 with any number of processors and any number of memory modules of different types (e.g., SRAM and DRAM) and different numbers of ports. DX-Gt automatically configures both the SoCDMMU and the MxN Xbar. Currently, DX-Gt only supports two kinds of processors: MPC750 and ARM9TDMI. However, DX-Gt can be easily extended to additional processors. The SoCDMMU and the Xbar have a generic bus interface. Each PE in our target architecture has a wrapper that converts the PE's bus interface signals into the SoCDMMU and Xbar generic bus interface signals. Supporting a new PE is just a matter of developing a new wrapper for the PE's bus interface (which is an easy task).

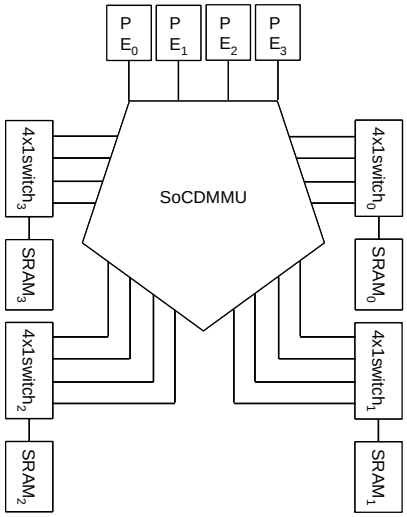

Figure 2. The target architecture of four processors and four memory blocks each with a single port

Figure 2 shows a system that is generated by DX-Gt. Note that four 4x1 switches in Figure 2 are grouped together into the configurable Xbar (4x4 Xbar in this case) in Figure 1. A PE is connected to four 4x1 switches via an SoCDMMU (1-to-4 connections) and sends an address and data and controls (read, write and byte selection) to all four 4x1 switches at the same time. Each 4x1 switch translates each address to check if the address belongs to the corresponding address space of the attached SRAM. The system consists of four processors and four single port SRAM modules. Each processor block in Figure 2 can be either MPC750 or ARM9TDMI depending on the user input. In Figure 2, the generated 4x4 Xbar switch, which consists of four 4x1 switch blocks, provides four concurrent accesses to four SRAM modules by four processors. The generated SoCDMMU manages the dynamic allocation/deallocation of memory in the four SRAM modules.

The target architecture runs the Atlanta Real-Time Operating System (RTOS) which is an open source RTOS developed at the Georgia Institute of Technology for a shared memory multiprocessor SoC [3]. DX-Gt can configure Atalanta to support the SoCDMMU – if used – and tune its different modules to reflect the user settings.

A. The SoCDMMU

The SoC Dynamic Memory Management Unit (SoCDMMU) is a hardware unit, to be a part of the SoC, that allows a fast and deterministic dynamic way to allocate/de-allocate global memory between PEs [2]. The PEs are connected to the on-chip memory via the SoCDMMU as shown in Figure 1 which allows the SoCDMMU to control all of the global memory accesses. This feature enables the SoCDMMU to convert the PE address (virtual address) to a physical address. The SoCDMMU is mapped into a location in the I/O space of each PE. This memory mapped address or I/O port to which the SoCDMMU is mapped is used to send commands to the SoCDMMU (writing data to the port or memory-mapped location) and to receive the status of the command execution (reading from the port or memory-mapped location).

The SoCDMMU assumes that the global on-chip memory is divided into small global memory blocks called *G_block*s. The SoCDMMU can allocate a page of one or more *G_block*s to a PE upon request. Each *G_block* has one physical address and one or more virtual addresses (PE addresses). The base virtual address a particular PE assigns to a *G_block* may differ from one PE to another.

The introduction of the SoCDMMU introduces a new memory management hierarchy we call Two-Level Memory Management. Level Two, in Two-Level Memory Management, is the management of the global on-chip memory blocks among the on-chip Processing Elements, while Level One is the management of memory allocated to a particular on-chip Processing Element, e.g., an operating system's management of memory allocated to a particular processor.

The SoCDMMU allocation/deallocation of the memory *G_block*s is completely deterministic, which makes the SoCDMMU suitable for real-time SoC applications. Using the SoCDMMU speeds up the system; in [5], we showed an example where our approach gives a 4.4X overall speedup in

memory management during the application transition time when compared to a fully shared memory system with the same memory organization and number of processors.

B. The Xbar

Our generated MxN Xbar switch consists of N Mx1 switches. M is equal to the number of PEs and N equals the number of memory blocks. An Mx1 switch chooses one processor out of M processors to which to grant access to the attached memory block [4]. Figure 3 shows the internal structure of a 4x1 switch; in Figure 3, prev_ indicates wires from an SoCDMMU. Also, a number appended to the signal name identifies a signal from the corresponding processor. The operation of the switch block is as follows. First, a comparator (comp) compares the addresses from the SoCDMMU (prev_addresses) only if prev_req signals are asserted. If a prev_address input belongs to the address space of the attached memory block, mem_req is asserted. The mem_req signal asks an arbiter to grant a single bus attached to the corresponding memory block. An arbiter handles M requests from M processors and grants one request in round-robin order by asserting the appropriate mem_on signal [4]. A mem_on signal turns on switch blocks (addr bus switch, data bus switch, wire switches for read and write signals and wire_ta switch for memory transfer acknowledgement) in a 4x1 switch.

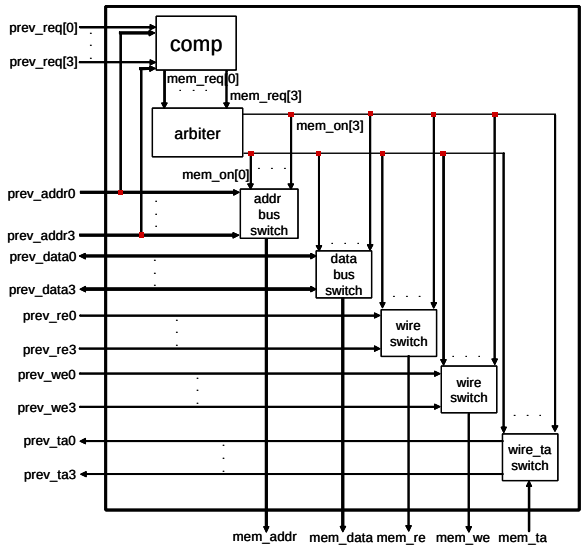

Figure 3. Internal Structure of a 4x1 Switch

Example 1: Suppose PE_0 and PE_3 both try to access $SRAM_0$, PE_1 tries to access $SRAM_2$, and PE_2 tries to access $SRAM_1$ in Figure 2. Then, prev_addr0 through prev_addr3 from the SoCDMMU are compared in the comparator of 4x1 $switch_0$. Consider Figure 3 to describe 4x1 $switch_0$. Then, in this case as described so far, only prev_addr0 and prev_addr3 are matched to the address space of $SRAM_0$ resulting in the assertion of mem_req[0] and mem_req[3]. Likewise, only mem_req[1] and mem_req[2] are asserted in 4x1 $switch_2$ and 4x1 $switch_1$, respectively. In this case, the arbiter of 4x1$switch_0$ grants the request mem_req[0]; mem_req[3] will be next in round-robin order. Thus, PE_0's request is granted. Then, mem_on[0] turns on the corresponding switch blocks so that prev_addr0 is connected to mem_addr, prev_data0 to mem_data, prev_re0 to mem_re, prev_we0 to mem_we, and prev_ta0 to mem_ta.

At the same time, only mem_req[2] is asserted for 4x1 $switch_1$ since only prev_addr2 is matched to $SRAM_1$. Likewise, only mem_req[1] is asserted for 4x1 $switch_2$ since only prev_addr1 is matched to $SRAM_2$. Thus, PE_2's and PE_1's memory access requests to $SRAM_1$ and $SRAM_2$ are both granted. Thus, in this example, three concurrent memory transfers are supported. □

IV. Methodology

Figure 4 gives an overview of the flow of our configuration tool. A Graphical User Interface (GUI), which consists of a set of HTML forms, captures the user's inputs and passes them to the Dynamic memory management unit and crossbar (Xbar) switch Generator (DX-Gt) application (developed in C-Language). DX-Gt processes the user inputs, validates them and generates the SoC hardware files in RTLVerilog. Moreover, DX-Gt generates Synopsys DC™ [8] synthesis scripts and a Mentor Graphics Seamless CVE™ [18] configuration file for simulation of the resulting SoC design.

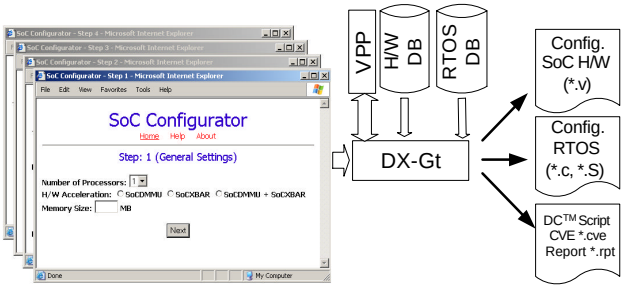

Figure 4. The SoC Configuration Tool Flow

The following is a partial list of the user specified parameters:

- System wide parameters
  - The number of PEs which determines M in an MxN Xbar and the number of the SoCDMMU ports
  - The number of the global on-chip memory *G_block*s which determines the size of the SoCDMMU memory allocator
  - The sizes of the global on-chip memory *G_block*s which determines the address bus widths between Mx1 switches and memory blocks
  - The number of memory ports which determines N in an MxN Xbar
  - The PE types which determines processor interfaces to SoCDMMU chosen from a hardware database

- The memory type which determines the memory controller chosen from a hardware database
- The choice of use of SoCDMMU, Xbar, both or none

- SoCDMMU related parameters
  - The scheduling scheme to resolve concurrent memory requests from different PEs (first come first served scheme or priority scheme)
  - Memory *G_block*s initially assigned to the PEs (initial memory assignment for the PEs)
- Xbar related parameters
  - The data bus width of each PE (determined by the PE type)
  - The address bus width connected to each PE (determined by the global memory size)
  - The number of memory modules determined by the *number of G_blocks* parameter

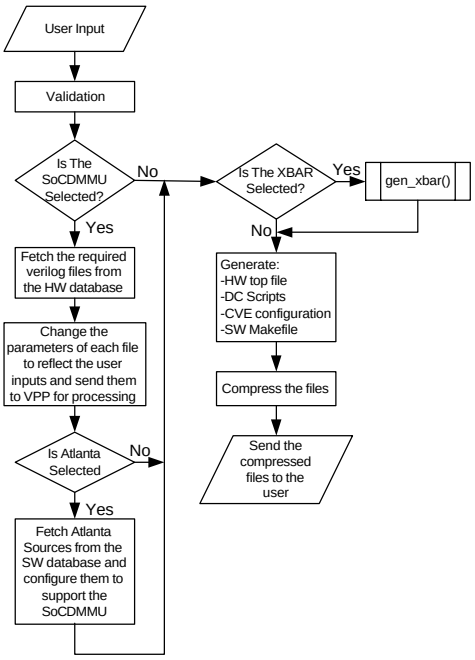

Figure 5. Flowchart of DX-Gt

In order to generate the hardware files, a "hardware database (HW DB)" of parameterized Verilog files of each system component – SoCDMMU sub-modules, processor bus wrappers, memory controller and Xbar switches and comparators in Figure 3 – is being used. The Xbar arbiter in Figure 3 is generated in a way that described in [4]. The Verilog files in the database are written in such a way that a custom version of the file can be generated using a Verilog preprocessor.

Once the user configurations and settings are captured using a set of HTML forms, DX-Gt selects from the database the hardware components that satisfy the user specified configurations. Next, DX-Gt sets the parameters of each component to reflect the user input. The hardware components (preprocessed Verilog files) are passed to the Verilog PreProcessor (VPP) [6] which processes them and generates new customized Verilog files. Figure 5 shows the flowchart of DX-Gt.

As shown in Figure 5, the Xbar hardware is generated by calling the function *gen_xbar()*. The function *gen_xbar(bus parameters)* (as shown in Figure 6) generates N (number of memory blocks) Mx1 switches where M is equal to the number of processors.

Example 2: Consider an SoC with four ARM9TDMI processors and global on-chip memory of 16Mbytes. Now suppose that the user chooses to use the SoCDMMU. Also, suppose that the user requests the on-chip memory to be divided into 128 *G_block*s and the use of first come first come first served (FCFS) scheduler to resolve concurrent accesses to the SoCDMMU. To generate a custom SoCDMMU that reflects the user's input, DX-Gt fetches from the HW DB an ARM9TDMI bus wrapper for the SoCDMMU and the required SoCDMMU sub-modules (the Allocation Unit, Allocation Table, Address Converter and FCFS Scheduler [30]) preprocessed Verilog files. Then, Dx-Gt sets the variables on the top of the preprocessed Verilog files to reflect the user's configurations. For example in Figure 6 Dx-Gt sets the variables n to 128 (number of G_blocks), p to 4 (number of PEs) and sch to 1 (FCFS scheduler) in the SoCDMMU top preprocessed Verilog file (socdmmu.vpp). The modified socdmmu.vpp is then processed by VPP to generate the customized socdmmu.v file as shown in Figure 6. Please note that because the variable sch is set to "1" in the socdmmu.vpp file, as shown in Figure 6, the customized SoCDMMU (socdmmu.v in Figure 6) uses the FCFS scheduler. □

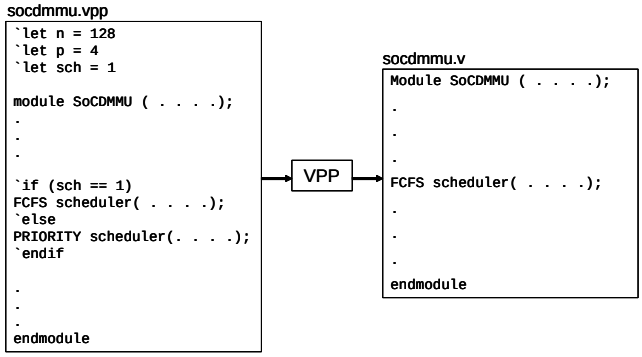

Figure 6. sub-module customization for Example 2

The function *gen_xbar()* makes use of a linked-list to store the wire names for processors, memory blocks and bus lines. To generate an MxN Xbar, first the function *gen_xbar()* calls the function *gen_Mx1()* N times to generate N Mx1 switches; then, *gen_xbar()* integrates these N Mx1 switches (submodules in Verilog) into an Xbar by generating a top file.

To generate Mx1 switches, the function *gen_Mx1()* first fills the wire names linked-list with the processors' wire names, the memory blocks' wire names and the bus lines'

wire names by calling the functions: *gen_proc_wires()*, *gen_mem_wires()* and *gen_bus_wires()*, respectively. Then, *gen_Mx1()* invokes a set of functions to generate address bus switches, data bus switches, wire switches and wire_ta switches (defined in Section III.B) in Figure 3. These switches are hand-coded beforehand. Finally, *gen_Mx1()* calls *gen_arbiter(M)* to generate the appropriate arbiter using the algorithm described in [20]. Note that specific user input for the arbiter is not required because the arbiter type is set to a bus arbiter and the number of masters is set to M [20].

```
gen_xbar(bus parameters) {
   n=0;
   while (n<N)
      gen_Mx1(bus parameters);
   integrate();
}
gen_Mx1(bus parameters) {
         gen_proc_wires(M) ;
         gen_mem_wires(N);
         gen_bus_wires(M);
         gen_addr_bus_switch(M);
         gen_data_bus_switch(M);
         gen_wire_switch(M);
         gen_wire_ta_switch(M);
         gen_arbiter(M);
         gen_comp(M);
}

 gen_proc_wires(M) {
            m=0;
            while (m<M) { /* M = number of processors */
              fill in data structure for processor wire names;
            }
 }
gen_mem_wires(N) {
             n=0;
             while (n<N) { /* N = number of memory blocks */
               fill in data structure for memory wire names;
             }
}
 gen_bus_wires(M) {
           m=0;
            while (m<M) { /* M = number of processors */
              fill in data structure for processor wire names;
 }
```

Figure 7. Pseudo Code for the MxN Xbar generation.

Example 3: Suppose a user wants to build a system with two MPC750s and two ARM9TDMIs and four memory blocks with sizes 2Mbytes, 2Mbytes, 4Mbytes and 8Mbytes. DX-Gt, shown in Figure 5, first determines the data bus widths to be 64 bits for MPC750 and 32 bits for ARM9TDMI. Also, the address bus widths are set to be 32 bits for both processors. The memory address bus widths of the four memory blocks are set to 21 bits, 21 bits, 22 bits and 23 bits. The order of memory blocks attached to the generated Xbar is clock-wise as shown in Figure 2. Next, DX-Gt, as indicated by the rightmost box in Figure 5, calls *gen_xbar()*. *gen_xbar()* calls *gen_Mx1()* which, as shown in Figure 7, fills the data structure for wire names by *gen_proc_wires()*, *gen_mem_wires()* and *gen_bus_wires()*. Then, *gen_Mx1()* generates the switch blocks shown in Figure 3. *gen_Mx1(),* as seen in Figure 6, also invokes RAG [20] to generate an arbiter handling 4 requests and generating a comparator comparing 4 addresses. The arbiter parameters (the number of requestors and the arbiter type) are passed to RAG. All generated submodules are connected together by wire names in *gen_Mx1()*. For example, prev_req signals are input signals to an Xar, the top module, from an SoCDMMU. Also, prev_req signals are input signals to four 4x1 switches and are input signals to the comparator as well, the submodule of 4x1 switches as shown in Figure 3. Thus, prev_req signals are connected to all four 4x1 switches (submodules of an Xbar) in the top module and are connected to a comparator (the submodule of a 4x1 switch) in each 4x1 switch. In this way, the same wire names are connected. After M=4 iterations, four 4x1 switches are generated with corresponding bus parameters. Finally, the function *integrate()* creates the top file so that the generated four 4x1 switches are wired together by bus wire names. □

For the software part, the d framework does the required modification to the Atalanta RTOS to support the generated hardware according to the user inputs[28][29]. Then the d framework generates the modified source files (in C and assembly), also the d framework generates the makefile required to compile the RTOS.

V. Synthesis Results

This section presents the synthesis results of the SoCDMMU and the Xbar. We use the Synopsys Design Compiler [8] with a .25µm TSMC technology library [19] from LEDA Systems [9].

A. Xbar

We use the TSMCWIRE model (lec25dscc25_FF) for a wire load to provide more accurate area results. Figure 7 shows the synthesis results of the area of Mx1 switches for increasing number of processors. The total area of an MxN Xbar can be easily calculated by N times the area of an Mx1 switch, where N is the number of memory ports.

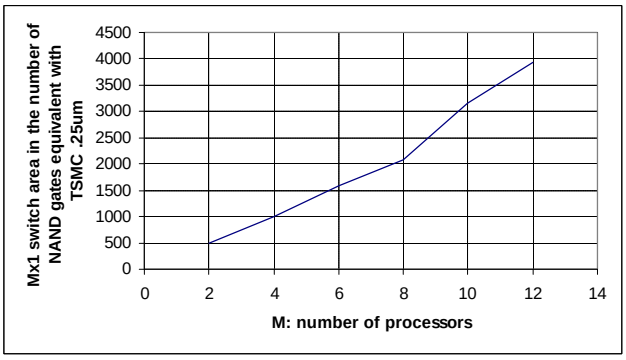

Figure 8. Area of Mx1 Switches.

As shown in Figure 8, the area of an Mx1 switch increases almost linearly with the number of PEs. However, as shown in Figure 9, the area of Xbar increases sub-quadratically as the number of PEs and the number of memory ports increases for an MxN Xbar configuration.

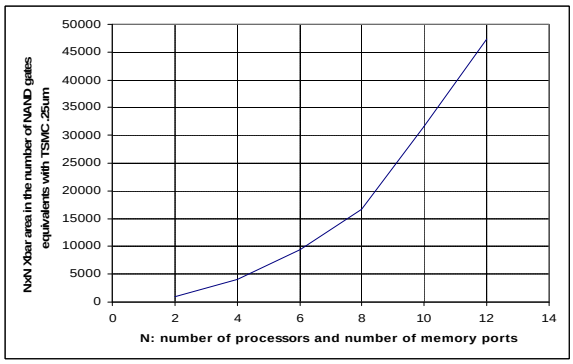

Figure 9. Area of MxM Xbar Switches

B. SoCDMMU

Figure 10 shows how the SoCDMMU area scales with the number of PEs (2, 4, 8 and 12) and *G_blocks* (128, 256, 512 and 1024). The area (represented by number of equivalent NAND gates) scales linearly with the number of processors. Please note that the results were obtained using a clock frequency of 100MHz.

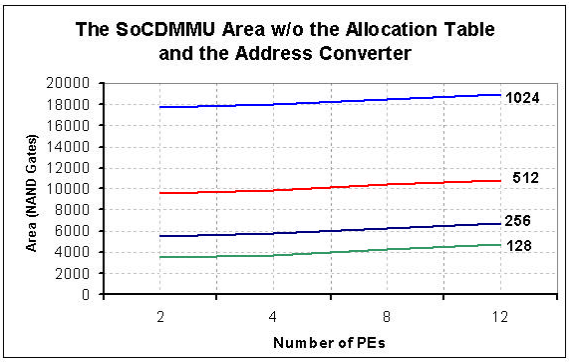

Figure 10. The Area of the SoCDMMU (w/o the Allocation Table and the Address Converter) for different number of PEs and *G_block*s

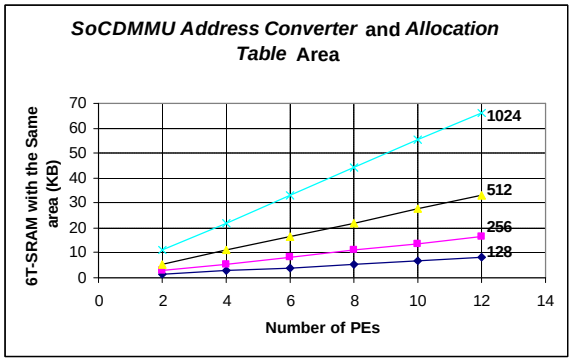

Figure 11. The Area of the *Address Converter* and the *Allocation Table* for different number of PEs and *G_blocks*

Figure 11 shows the area of the *Address Converter* and the *Allocation Table*, which are mainly memory elements and are both part of the SoCDMMU, for different numbers of processors (2, 4, 8 and 12) and *G_blocks* (128, 256, 512 and 1024). The area is represented in equivalent 6T-Static Random Access Memory (SRAM) area.

VI. SoCDMMU and Xbar Layout

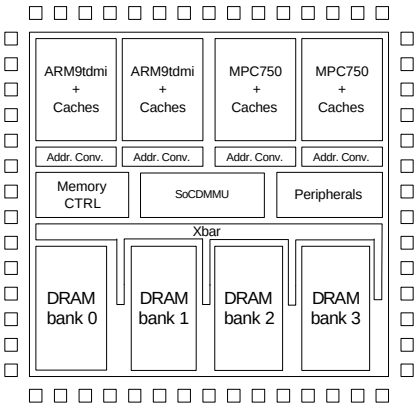

Figure 12. The floorplan of an SoC that utilizes the SoCDMMU and the Xbar

Creating a high clock rate working chip containing millions of logic gates, whose functionality used to be implemented by multiple chips, is a time consuming task for Integrated Circuits (IC) designers. Usually designers start with floorplanning the chip by partitioning it into portions each of which presents a functional entity in the SoC (e.g., CPU, DRAM, caches or custom peripheral logic). Also, this step involves pin assignment, global routing and global clock tree generation. Tools such as Synopsys Chip Architect [24] can help the SoC designer to accomplish such tasks. The floorplanning and global routing provides useful timing information required for the RTL logical and physical synthesis of each functional entity using tools such as Synopsys Design/Physical Compiler [25][26] and Cadence Silicon Ensemble (SE) [27]. Finally the layout of each unit is integrated into the SoC floorplan.

Dx-Gt generates the RTL Verilog of the SoCDMMU and the Xbar which are parts of the SoC presented in Figure 1. These RTL models with the RTL models of the other system components can be used to generate the layout of the SoC IC using the design flow described in the previous paragraph. As a sample effort to show part of the methodology described in the previous paragraph, we synthesized the RTL Verilog of the SoCDMMU (customized for 256 *G_block*s and 4 PEs) and the Xbar and then placed and routed the layout using a 0.25µm TSMC technology library. Figure 12 shows the floorplan of the system with four processors and four memory blocks of Example 3 including the layouts of the SoCDMMU and the Xbar. Since DRAM cores are not available, Figure 12 presumably describes how 4x4 Xbar is interleaved into DRAM modules. From our layout of a subset of Figure 12 (i.e., SoCDMMU plus Xbar), the area of 4x4 Xbar is 0.23 mm^2 and the area of SoCDMMU is 1.43 mm^2.

VII. Conclusion

In this paper, we described a System-on-a-Chip IP generation tool that enables an SoC designer to design a multiprocessor SoC and configure its memory and bus subsystems to meet the design constraints with ease.

Our paper is focused on the provision of a CAD tool for a memory management unit and an MxN Xbar switch which we named **D**ynamic memory management unit and **X**bar **G**enera**t**or (DX-Gt). The first contribution of our paper is to provide an automatic generation of SoC Dynamic Memory Management Unit (SoCDMMU). The second contribution is the automatic generation of crossbar (Xbar) switch and its arbiter as well. Both hardware units are synthesizable at the RTL level.

Also, we showed the integration of the SoCDMMU and the Xbar into one system. The combination of the SoCDMMU and the Xbar for a multiprocessor SoC has been shown to have an overall speedup of 4.4X during the application transition time when compared to a fully shared memory system with the same memory organization and number of processors [5]. DX-Gt automatically generates in RTL Verilog hardware files required to build the system. Also, DX-Gt generates the files required to enable standard EDA tools to implement the system.

Acknowledgments

This research is funded by NSF under INT-9973120, CCR-9984808 and CCR-0082164. We also acknowledge donations received from Denali, Hewlett-Packard, Intel, LEDA, Mentor Graphics, Sun and Synopsys.

References

[1] The International Technology Roadmap for Semiconductors, edited by SIA Semiconductor Industry Association, 2002.

[2] M. Shalan and V. Mooney, "A Dynamic Memory Management Unit for Embedded Real-Time System-on-a-Chip," *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'00)*, November 2000, pp. 180-186.

[3] D. Sun, D. Blough, and V. Mooney, "Atalanta: A New Multiprocessor RTOS Kernel for System-on-a-Chip Applications," Georgia Institute of Technology, Atlanta, Georgia, Technical Report GIT-CC-02-19, 2002, http://www.cc.gatech.edu/tech_reports/.

[4] E. S. Shin, V. J. Mooney and G. F. Riley, "Round-robin Arbiter Design and Generation," *Proceedings of the International Symposium on System Synthesis (ISSS'02)*, October 2002, pp. 243-248.

[5] M. Shalan and V. Mooney, "Hardware Support for Real-Time Embedded Multiprocessor System-on-a-Chip Memory Management," *Proceedings of the Tenth International Symposium on Hardware/Software Codesign (CODES'02)*, May 2002, pp. 79-84.

[6] VPP, Available HTTP: http://www.surefirev.com/vpp/

[7] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally and M. Horowitz, "Smart Memories: A Modular Reconfigurable Architecture," *Proceedings of International Symposium on Computer Architecture (ISCA'00)*, June 2000, pp. 161-171.

[8] Synopsys Design Compiler. Available HTTP: http://www.synopsys.com/products/logic/.

[9] LEDA Systems. Available HTTP: http://ledasys.com.

[10] Artisan Components, Inc. http://www.artisan.com/.

[11] Tensilica, Inc. Available HTTP: http://www.tensilica.com.

[12] ARC Inc. Available HTTP: http://www.arc.com/.

[13] E. V. Puttkamer, "A simple hardware buddy system memory allocator," *IEEE Transaction on Computers*, vol. 24, no. 10, October 1975, pp. 953-957.

[14] H. Cam et al., "A high-performance hardware-efficient memory allocation technique and design," *Proceedings of International Conference on Computer Design (ICCD'99)*, October 1999, pp. 274-276.

[15] J. M. Chang et al., "Introduction to DMMX (Dynamic Memory Management Extension)," *Proceedings of ICCD Workshop on Hardware Support for Objects and Micro architectures for Java*, October 1999, pp. 11-14.

[16] J. M. Chang and E. F. Gehringer, "A High-Performance Memory Allocator for Object-Oriented Systems," *IEEE Transactions on Computers*, vol. 45, no. 3, March 1996, pp. 357-366.

[17] K.C. Knowlton, "A Fast Storage Allocator," *Communications ACM*, Vol. 8, October 1965, pp. 623-625.

[18] Mentor Graphics Seamless. Available HTTP: http://www.mentor.com/seamless.

[19] TSMC IP Services. Available HTTP: http://www.tsmc.com/design/ip.html.

[20] E. S. Shin, V. J. Mooney and G. F. Riley, "Round-robin Arbiter Design and Generation," Georgia Institute of Technology, Atlanta, GA, Technical Report GIT-CC-02-38, 2002, http://www.cc.gatech.edu/tech_reports.

[21] IBM CoreConnect Bus Architecture, Available HTTP: http://www-3.ibm.com/chips/products/coreconnect.

[22] S. Wuytack et al., "Memory Management for Embedded Network Applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 5, May 1999, pp. 533-544.

[23] Y. Li and W.H. Wolf, "Hardware/Software Co-Synthesis with Memory Hierarchies," *IEEE Transaction Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 10, October 1999, pp. 1405-1417.

[24] Synopsys Chip Architect, Available HTTP: http://www.synopsys.com/products/designplanning/designplanning.html.

[25] Synopsys Design Compiler, Available HTTP: http://www.synopsys.com/products/logic/design_compiler.html

[26] Synopsys Physical Compiler, Available HTTP: http://www.synopsys.com/products/unified_synthesis/unified_synthesis.html

[27] Cadence Silicon Ensemble, Available HTTP: http://www.cadence.com/products/sepks.html

[28] V. Mooney and D. Blough,"A Hardware-Software Real-Time Operating System Framework for SOCs," IEEE Design and Test of Computers, November-December 2002, pp. 44-51.

[29] J. Lee, K. Ryu and V. Mooney, "A Framework for Automatic Generation of Configuration Files for a Custom Hardware/Software RTOS," *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'02)*, pp. 31-37, June 2002.

[30] M. Shalan and V. Mooney, "Hardware Support for Real-Time Embedded Multiprocessor System-on-a-Chip Memory Management," Georgia Institute of Technology, Atlanta, Georgia, Technical Report GIT-CC-03-02, 2003, http://www.cc.gatech.edu/tech_reports/.