Constraint-Driven Bus Matrix Synthesis for MPSoC

Sudeep Pasricha†, Nikil Dutt†, Mohamed Ben-Romdhane‡

†Center for Embedded Computer Systems
University of California, Irvine, CA
{sudeep, dutt}@cecs.uci.edu

‡Conexant Systems Inc.
Newport Beach, CA
m.benromdhane@conexant.com

Abstract – Modern multi-processor system-on-chip (MPSoC) designs have high bandwidth constraints which must be satisfied by the underlying communication architecture. Bus matrix based communication architectures consist of several parallel busses which provide a suitable backbone to support high bandwidth systems, but suffer from high cost overhead due to extensive bus wiring inside the matrix. Manual traversal of the vast exploration space to synthesize a minimal cost bus matrix that also satisfies performance constraints is practically infeasible. In this paper, we address this problem by proposing an automated approach for synthesizing a bus matrix communication architecture which satisfies all performance constraints in the design and minimizes wire congestion in the matrix. To validate our approach, we consider several industrial strength applications from the networking domain and show that our approach results in up to 9× component savings when compared to a full bus matrix and up to 3.2× savings when compared to a maximally connected reduced bus matrix.

I. Introduction

Multi-processor system-on-chip (MPSoC) designs are increasingly being used in today's high performance embedded systems. These systems are characterized by a high level of parallelism, due to the presence of multiple processors, and large bandwidth requirements, due to the massive scale of component integration. The choice of communication architecture in such systems is of vital importance because it supports the entire inter-component data traffic and has a significant impact on the overall system performance.

Traditionally used hierarchical shared bus communication architectures such as those proposed by AMBA [1], CoreConnect [2] and STbus [3] can cost effectively connect few tens of cores but are not scalable to cope with the demands of very high performance systems. Point-to-point communication connection between cores is practical for even fewer components. Network-on-Chip (NoC) based communication architectures [5] have recently emerged as a promising alternative to handle communication needs for the next generation of high performance designs. However, although basic concepts have been proposed, research on NoCs is still in its infancy, and few concrete implementations of complex NoCs exist to date [6].

In this paper we look at *bus matrix* (sometimes also called *crossbar switch*) based communication architectures [7] which are currently being considered by designers to meet the high bandwidth requirements of modern MPSoC systems. Fig. 1 shows an example of a three-master seven-slave AMBA bus matrix architecture for a dual ARM processor based networking subsystem application. A bus matrix consists of several busses in parallel which can support concurrent high bandwidth data streams. The *Input stage* is used to handle interrupted data bursts, and to register and hold incoming transfers if receiving slaves cannot accept them immediately. The *Decode* stage generates select signal for appropriate slaves. Unlike in traditional shared bus architectures, *arbitration* in a bus matrix is not centralized, but rather distributed so that every slave has its own arbitration.

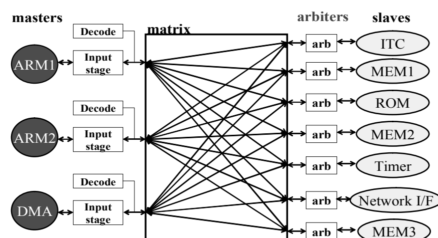

Fig. 1. Full bus matrix architecture

One drawback of the *full bus matrix* structure shown in Fig. 1 is that it connects every master to every slave in the system, resulting in a prohibitively large number of busses. The excessive wire congestion can make it practically impossible to route and achieve timing closure for the design [14]. To overcome this shortcoming, designers tailor a full matrix structure to the particular application at hand, creating a *partial bus matrix*, as shown in Fig. 2. This structure has fewer busses and consequently uses fewer components (arbiters, decoders, buffers), has a smaller area and also utilizes less power.

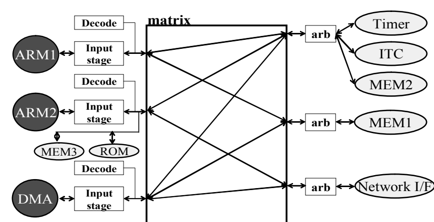

Fig. 2. Partial bus matrix architecture

The problem of synthesizing a minimal cost (i.e. having the least number of busses) bus matrix for a particular application is complicated by the large number of combinations of possible matrix topologies and bus architecture parameters such as bus widths, clock speeds, out-of-order (OO) buffer sizes and shared slave arbitration schemes. Previous research in the area of bus matrix/crossbar synthesis (discussed in the next section) has been inadequate in addressing the entire problem, and instead has been limited to exploring a small subset of the synthesis problem (such as topology synthesis [8]). Very often, designers end up evaluating the bus matrix design space by creating simulation models annotated with detail based on experience, and manually iterating through

0-7803-9451-8/06/$20.00 **©2006 IEEE.**

different combinations of topology and communication architecture parameters. Such an effort remains time consuming and produces bus matrix architectures which are generally overdesigned for the application at hand.

Our goal in this paper is to address this problem by presenting an automated approach for synthesizing a bus matrix communication architecture, which generates not only the matrix topology, but also communication parameter values for bus clock speeds, OO buffer sizes and arbitration strategies. Most importantly, our synthesis effort minimizes the number of busses in the matrix and satisfies all performance constraints in the design. To demonstrate the effectiveness of our approach we synthesize a bus matrix architecture for four industrial strength MPSoC case studies from the networking domain and show that our approach significantly reduces wire congestion in a matrix, resulting in up to 9× component savings when compared to a full bus matrix and up to 3.2× savings when compared to a maximally connected reduced bus matrix.

II. Related Work

The need for bus matrix (or crossbar switch) architectures has been emphasized in previous work in the area of communication architecture design. Lahtinen et al. [9] compared the shared bus and crossbar topologies to conclude that the crossbar is superior to a bus for high throughput systems. Ryu et al. [10] compared a full crossbar switch with other bus based topologies and found that the crossbar switch outperformed the other choices due to its superior parallel response. Loghi et al. [11] presented exploration studies with the AMBA and STBus shared bus, full crossbar and partial crossbar topologies, concluding that crossbar topologies are much better suited for high throughput systems requiring frequent parallel accesses. An interesting conclusion from their work is that partial crossbar schemes can perform just as well as the full crossbar scheme, if designed carefully. However, the emphasis of their work was not on the generation of such partial crossbar topologies.

Although a lot of work has been done in the area of hierarchical shared bus architecture synthesis [12-14] and NoC architecture synthesis [15-16], few efforts have focused on bus matrix synthesis. Ogawa et al. [17] proposed a transaction based simulation environment which allows designers to explore and design a bus matrix. But the designer needs to manually specify the communication topology, arbitration scheme and memory mapping, which is too time consuming for the complex systems of today. The automated synthesis approach for STBus crossbars proposed by Murali et al. in [8] is the only work that comes closest to our goal of automated bus matrix synthesis. However, their work primarily deals with automated crossbar topology synthesis – the communication parameters (arbitration schemes, OO buffer sizes, bus widths and speeds) which have considerable influence on system performance [22-23] are not explored or synthesized. Our synthesis effort overcomes this shortcoming and synthesizes both the topology and communication architecture parameters for the bus matrix. Additionally, [8] assumes that critical data streams cannot overlap on the same bus, places a static limit on the maximum number of components that can be attached to a bus and also requires the designer to specify hard-to-determine threshold values of traffic overlap as an input, based on which components are allocated to separate busses. These are conservative approaches which lead to an overdesigned, sub-optimal system. Our approach carefully selects appropriate arbitration schemes (e.g. TDMA based) that can allow multiple constraint streams to exist on the same bus, and also does not require the designer to specify data traffic threshold values or statically limit the number of components on a bus. Experimental comparison studies (described in Section IV) show that our scheme is more aggressive and obtains greater reduction in bus matrix connections, when compared to [8].

III. Bus Matrix Synthesis

This section describes our approach for automated bus matrix synthesis. First we formulate the problem and present our assumptions. Next, we describe our simulation engine and elaborate on communication parameter constraints, which guide the matrix synthesis process. Finally, we present our automated bus matrix synthesis approach in detail.

A. Problem Formulation

We are given an MPSoC design having several components (IPs) that need to communicate with each other. We assume that hardware/software partitioning has taken place and that the appropriate functionality has been mapped onto hardware and software IPs. These IPs are standard “black box” library components which cannot be modified during the synthesis process, except for the memory components. The target standard bus matrix communication architecture (e.g. AMBA bus matrix [1]) that determines the pins at the IP interface and for which the matrix must be synthesized, is also specified. Typically, all busses within a bus matrix have the same data bus width, which usually depends on the number of data interface pins of the IPs in the design. We assume that this matrix data bus width is specified by the designer, based on the knowledge of the IPs selected for the design.

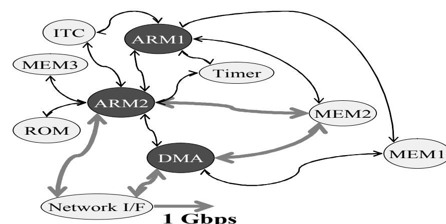

Fig. 3. Communication Throughput Graph (CTG)

Generally, MPSoC designs have performance constraints which are dependent on the nature of the application. The *throughput* of communication between components is a good measure of the performance of a system [12]. To represent performance constraints in our approach, we define a **Communication Throughput Graph** *CTG = G(V,A)* which is a directed graph, where each vertex *v* represents a component in the system, and an edge *a* connects components that need to communicate with each other. A **Throughput Constraint Path** (*TCP*) is a sub-graph of a *CTG*, consisting of a single master for which data throughput must be maintained and other masters, slaves and memories which are in the critical

path that impacts the maintenance of the throughput. Fig. 3 shows a *CTG* for a network subsystem, with a *TCP* involving the ARM2, MEM2, DMA and 'Network I/F' components, where the rate of data packets streaming out of the 'Network I/F' component must not fall below 1 Gbps.

Problem Definition *A bus* ***B*** *can be considered to be a partition of the set of components* ***V*** *in a CTG, where* $\boldsymbol{B} \subset \boldsymbol{V}$. *Then the problem is to determine an optimal component to bus assignment for a bus matrix architecture, such that the* ***V*** *is partitioned onto a minimal number of busses* ***N****, and satisfies all performance constraints in the design, represented by the TCPs in a CTG.*

B. Simulation Environment

Since communication behavior in a system is characterized by unpredictability due to dynamic bus requests from cores, contention for shared resources, buffer overflows etc., a simulation based approach is necessary for accurate performance estimation. For the simulation part of our flow, we capture behavioral models of components and bus architectures in SystemC [18][24], and keep them in an IP library database. Since we were concerned about the speed of simulation, we chose a fast transaction-based, bus cycle accurate modeling abstraction, which averaged simulation speeds of 150–200 Kcycles/sec [19], while running embedded software applications on processor ISS models. The communication model in this abstraction is extremely detailed, capturing delays arising due to frequency and data width adapters, bridge overheads, interface buffering and all the static and dynamic delays associated with the standard bus architecture protocol being used.

C. Communication Parameter Constraint Set

In the interest of generating a practically realizable system, we allow a designer to specify a discrete set of valid values (constraint set) for communication parameters such as bus clock speeds, OO buffer sizes and arbitration schemes. We allow the specification of two types of constraint sets for components – a global constraint set (Ψ_G) and a local constraint set (Ψ_L). For instance, a designer might set the allowable bus clock speeds for a set of busses locally in a subsystem to multiples of 33 MHz, with a maximum speed of 166 MHz, based on the operation frequency of the cores in the subsystem, while globally, the allowed bus clock speeds are multiples of 50 MHz, up to maximum of 400 MHz. The presence of a local constraint overrides the global constraint, while the absence of it results in the resource inheriting global constraints. This provides a convenient mechanism for the designer to bias the synthesis process based on knowledge of the design and the technology being targeted. Such knowledge about the design is not a prerequisite for using our synthesis framework, but informed decisions can help avoid the synthesis of unrealistic system configurations.

D. Synthesis Approach

We now describe our automated bus matrix synthesis approach. Fig. 4 gives a high level overview of the flow. The inputs to the flow include a Communication Throughput Graph (CTG), a library of behavioral IP models, a target bus matrix template (e.g. AMBA bus matrix [1]) and a communication parameter constraint set (Ψ) – which includes Ψ_G and Ψ_L. The general idea is to first perform a fast TLM level simulation of the system to get application-specific data traffic statistics. This information is used in a global optimization phase to reduce the full bus matrix architecture, by removing unused busses and local slave components from the matrix. We call the resulting matrix a **maximally connected reduced matrix**. The next step is to perform a static *branch and bound based hierarchical clustering* of slave components in the matrix which further reduces the number of busses in the matrix. We rank the results of the static clustering analysis, from the best case solution (least number of busses) to the worst (most number of busses) and save them in the database. We then use a fast bus cycle accurate simulation engine [19] to validate and select the best solution which meets all the performance constraints, determine slave arbitration schemes, optimize the design to minimize bus speeds and OO buffer sizes and then finally output the optimal synthesized bus matrix architecture.

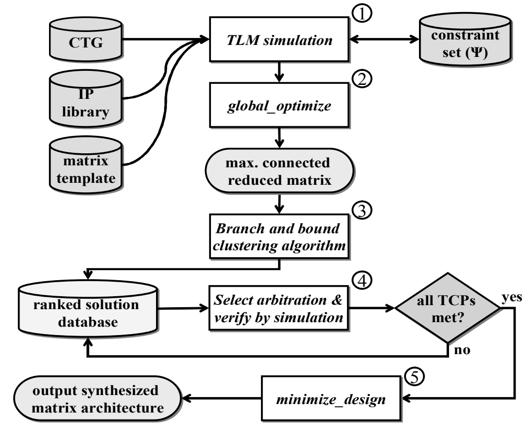

Fig. 4. Automated bus matrix synthesis flow

We now describe the synthesis flow in detail. In the first phase, the IP library is mapped onto a full bus matrix and simulated at the TLM level, with no arbitration contention overhead since there are no shared channels and also because we assume infinite ports at IP interfaces. We also set the OO buffer sizes to the maximum allowed in Ψ. The TLM simulation allows us to obtain application-specific data traffic statistics such as number of transactions on a bus, average transaction burst size on a bus and memory usage profiles. Knowing the bandwidth to be maintained on a channel from the Throughput Constraint Paths (TCPs) in the CTG, we can also estimate the minimum clock speed at which any bus in the matrix must operate, in order to meet its throughput constraint, as follows. The data throughput ($\Gamma_{TLM/B}$) from the TLM simulation, for any bus B in the matrix is given by

$$\Gamma_{TLM/B} = (numT_B \times sizeT_B \times width_B \times \Omega_B) / \sigma$$

where $numT$ is the number of data transactions on the bus, $sizeT$ is the average size of these data transactions, $width$ is the bus width, Ω is the clock speed, and σ is the total number of cycles of TLM simulation for the application. The values of

numT, *sizeT* and σ are obtained from the TLM simulation in phase 1. To meet the throughput constraint $\Gamma_{TCP/B}$ for bus *B*,

$$\Gamma_{TLM/B} \geq \Gamma_{TCP/B}$$

$$\therefore \quad \Omega_B \geq (\sigma \times \Gamma_{TCP/B}) / (numT_B \times sizeT_B \times width_B)$$

The minimum bus speed thus found is used to create (or update) the local bus speed constraint set $\Psi_{L(speed)}$ for the bus *B*.

In the next phase (phase 2 in Fig. 4), we perform global optimization (*global_optimize*) on the matrix by using information gathered from the TLM simulation in phase 1. In this phase we first remove all the busses that have no data traffic on them, from the full bus matrix. Next, we analyze the memory usage profile from the simulation run and attempt to split those memory nodes for which different masters access non-overlapping regions. Finally we cluster dedicated slave and memory components with their corresponding masters by migrating them from the matrix to the local busses of the masters, to reduce congestion in the bus matrix. Note that we perform memory splitting before local node clustering because it allows us to generate local memories which can then be clustered with their corresponding masters. After the *global_optimize* phase, the matrix structure obtained is termed as a **maximally connected reduced bus matrix**.

The next phase (phase 3 in Fig. 4) involves static analysis to determine the optimal reduced bus matrix for the given application. We make use of a *branch and bound based hierarchical clustering algorithm* to cluster slave components to reduce the number of busses in the matrix even further. Note that we do not consider merging masters because it adds two levels of contention (one at the master end and another at the slave end) in a data path, which can drastically degrade system performance. Before describing the algorithm, we present a few definitions. A slave cluster $SC = \{s_1 ... s_n\}$ refers to an aggregation of slaves that share a common arbiter. Let M_{SC} refer to the set of masters connected to a slave cluster *SC*. Next, let $\Pi_{SC1/SC2}$ be a superset of sets of busses which are merged when slave clusters *SC1* and *SC2* are merged. Finally, for a *merged bus set* $\beta = \{b_1 ... b_n\}$, where $\beta \subset \Pi_{SC1/SC2}$, let K_β refer to the set of allowed bus speeds for the newly created bus when the busses in set β are merged, and is given by

$$K_\beta = \Psi_{L(speed)}(b_1) \cap \Psi_{L(speed)}(b_2) \ldots \cap \Psi_{L(speed)}(b_n)$$

The branching algorithm starts out by clustering two slave clusters at a time, and evaluating the gain from this operation. Initially, each slave cluster has just one slave. The total number of clustering configurations possible for a bus matrix with *n* slaves is given by $(n! \times (n-1)!)/2^{(n-1)}$. This creates an extremely large exploration space, which cannot be traversed in a reasonable amount of time. In order to consider only valid clustering configurations and arrive at an optimal solution quickly, we make us of a bounding function. Fig. 5 shows the pseudo code for our bounding function which is called after every clustering operation of any two slave clusters *SC1* and *SC2*. In Step 1, we use a look up table to see if the clustering operation has already been considered previously, and if so, we discard the duplicate clustering. Otherwise we update the lookup table with the entry for the new clustering. In Step 2, we check to see if the clustering of SC1 and SC2 results in the merging of busses in the matrix, otherwise the clustering is not beneficial and the solution can be bounded. If the clustering results in bus mergers, we calculate the number of merged busses for the clustering and store the cumulative weight of the clustering operation in the branch solution node. In Step 3, we check to see if the allowed set of bus speeds for every merged bus is compatible or not. If the allowed speeds for any of the busses being merged are incompatible (i.e $K_\beta == \phi$ for any β), the clustering is not possible and we bound the solution. Additionally, we also calculate if the throughput requirement of each of the merged busses can be theoretically supported by the new merged channel. If this is not the case, we bound the solution. The bounding function thus enables a conservative pruning process which quickly eliminates invalid solutions and allows us to rapidly converge on the optimal solution.

Step 1: **if** (exists lookupTable(SC1,SC2)) **then** discard duplicate clustering
  else updatelookupTable(SC1, SC2)

Step 2: **if** ($M_{SC1} \cap M_{SC2} == \phi$) **then** bound clustering
  else $cum_weight = cum_weight + |M_{SC1} \cap M_{SC2}|$

Step 3: **for each** set $\beta \in \Pi_{SC1/SC2}$ **do**
  if $((K_\beta == \phi) \,||\, (\sum_{i=1}^{|\beta|} \Gamma_{TCP/i} > (width_B \times max_speed_B)))$ **then**
  bound clustering

Fig. 5. *bound* function

The solutions obtained from the static branch and bound clustering algorithm are ranked from best to worst and stored in a solution database. The next phase (phase 4 in Fig. 4) validates the solutions by simulation. We use a fast transaction-based bus cycle accurate simulation engine [19] to verify that the reduced matrix still satisfies all the constraints in the design. We perform arbitration strategy selection at this stage (from the allowed schemes in the constraint set Ψ). If a static priority based scheme for a shared slave (with priorities distributed among slave ports according to throughput requirements) results in TCP constraint violations, we make use of other arbitration schemes, in increasing order of implementation costs. So we would use a simpler arbitration scheme like round robin (RR) first, before resorting to the more elaborate TDMA/RR scheme like that used in [4]. It is possible that even after using these different arbitration conflict schemes, there are TCP constraint violations. In such a case we remove the solution from the solution database, and proceed to select the next best solution, continuing in this manner till we reach a solution which successfully passes the simulation based verification. This is the minimal cost solution, having the least number of busses in the matrix, while still satisfying all TCP constraints in the design. Once we arrive at such a solution, we call the *minimize_design* procedure (phase 5 in Fig. 4) where we attempt to minimize the bus clock speeds and prune OO buffer sizes. In this procedure, we iteratively select busses in the matrix and attempt to arrive at the lowest value of bus clock speeds (as allowed by Ψ) which does not violate any TCP constraint. We verify any changes made in bus speeds via simulation. After minimizing bus speeds, we prune the OO buffer sizes from the maximum values allowed to their peak traffic buffer count utilization values, obtained from simulation. Finally, we output the synthesized minimal cost bus matrix, with a well defined topology and parameter values.

IV. Case Studies

We applied our automated bus matrix synthesis approach on four MPSoC applications – VIPER, SIRIUS, ORION4 and HNET8 – from the networking domain. While VIPER and SIRIUS are variants of existing industrial strength applications, ORION4 and HNET8 are larger systems which have been derived from the next generation of MPSoC applications currently in development. Table 1 shows the number of components in each of these applications. Note that the *Masters* column includes the processors in the design.

Table 1. Number of cores in MPSoC applications

Applications	Processors	Masters	Slaves
VIPER	2	4	15
SIRIUS	3	5	19
ORION4	4	8	24
HNET8	8	13	29

Table 2. Throughput Constraint Paths (TCPs)

IP cores in Throughput Constraint Path (TCP)	TCP constraint
ARM1, MEM1, DMA, SDRAM1	640 Mbps
ARM1, MEM2, MEM6, DMA, Network I/F2	480 Mbps
ARM2, Network I/F1, MEM3	5.2 Gbps
ARM2, MEM4, DMA, Network I/F3	1.4 Gbps
ASIC1, ARM3, SDRAM1, Acc1, MEM5, Network I/F2	240 Mbps
ARM3, DMA , Network I/F3, MEM5	2.8 Gbps

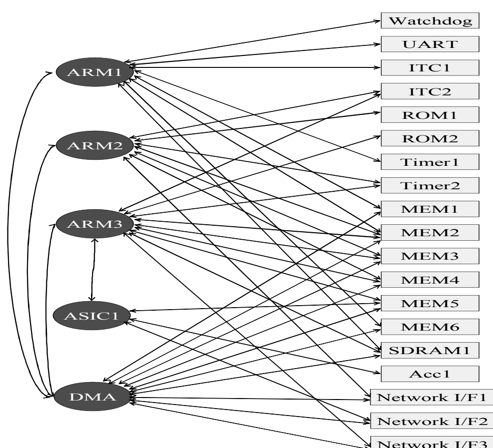

Fig. 6. CTG for SIRIUS application

Fig. 6 shows the CTG for the SIRIUS application. For clarity, the TCPs are presented separately in Table 2. ARM1 is a protocol processor (PP) while ARM2 and ARM3 are network processors (NP). The ARM1 PP is responsible for setting up and closing network connections, converting data from one protocol type to another, generating data frames for signaling, operating and maintenance and exchanging data with NP using shared memory. The ARM2 and ARM3 NPs directly interact with the network ports and are used for assembling incoming packets into frames for the network connections, network port packet/cell flow control, assembling incoming packets/cells into frames, segmenting outgoing frames into packets/cells, keeping track of errors and gathering statistics. The ASIC1 block performs hardware cryptography acceleration for DES, 3DES and AES. The DMA is used to handle fast memory to memory and network interface data transfers, freeing up the processors for more useful work. SIRIUS also has a number of memory blocks, network interfaces and peripherals such as interrupt controllers (ITC1, ITC2), timers (Watchdog, Timer1, Timer2), UART and a packet accelerator (Acc1).

Table 3. Customizable Parameter Constraint Set

Set	Values
bus speed	25, 50, 100, 200, 300, 400
arbitration strategy	static, RR, TDMA/RR
OO buffer size	1 – 8

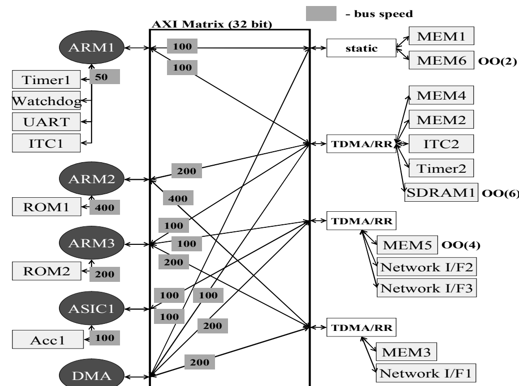

Fig. 7. Synthesized bus matrix for SIRIUS

Table 3 shows the global customizable parameter set Ψ_G. For the synthesis we target an AMBA3 AXI [21] based bus matrix structure. Fig. 7 shows the matrix structure output by our synthesis flow, which satisfies all six throughput constraints in the design (Table 2). The data bus width used in the matrix is 32 bits, and the slave-side arbitration strategies, operating speeds for the busses and OO buffer sizes (for components supporting OO transaction completion) are shown in the figure. While the full bus matrix used 95 busses, after the global optimization phase (Fig. 4) we were able to reduce this number to 34 for the maximally connected reduced matrix. The final synthesized matrix further reduces the number of busses to as few as 16 (this includes the local busses for the masters) which is almost a 6$\times$ saving in the number of busses used when compared to the original full matrix. The entire synthesis process took just a few hours to complete instead of the several days or even weeks it would have taken for a manual effort.

We now present two sets of experiments to prove the effectiveness of our approach - the first compares our synthesis results with previous work in the area of bus matrix synthesis, while the second compares the results of our synthesis approach for four MPSoC applications of varying complexity.

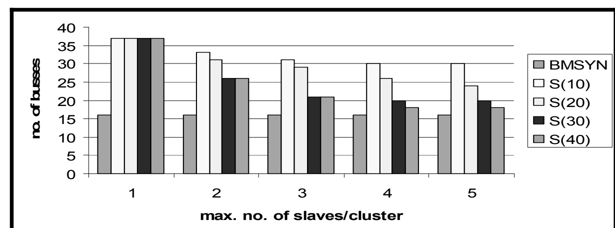

Fig. 8. Comparison with threshold based approach for SIRIUS

To compare the quality of our synthesis results, we chose the closest existing piece of work that deals with automated matrix synthesis with the aim of minimizing number of busses [8]. Since their approach only generates matrix topology (while we generate both topology and parameter values), we restricted our comparison to the number of busses in the final

synthesized design. The threshold based approach proposed in [8] requires the designer to statically specify (i) the maximum number of slaves per cluster and (ii) the traffic overlap threshold, which if exceeded prevents two slaves from being assigned to the same bus cluster. The results of our comparison study, performed on the SIRIUS application, are shown in Fig. 8. BMSYN is our bus matrix synthesis approach while the other comparison points are obtained from [8]. *S(x),* for x = 10, 20, 30, 40, represents the threshold based approach where no two slaves having a traffic overlap of greater than x% can be assigned to the same bus, and the X-axis in Fig. 8 varies the maximum number of slaves allowed in a bus cluster for these comparison points. The values of 10 – 40% for traffic overlap are chosen as per recommendations from [8]. It is clear from Fig. 8 that our synthesis approach produces a lower cost system (having lesser number of busses) than approaches which force the designer to statically approximate application characteristics.

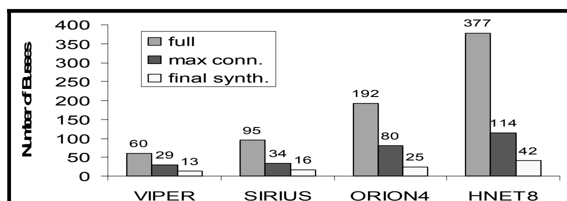

Fig. 9. Comparison of number of busses for MPSoC applications

The number of busses in a full bus matrix, a maximally connected reduced matrix and the final synthesized bus matrix using our approach, for the four applications we considered, are compared in Fig. 9. More detailed experimental results can be found in our technical report [20]. It can be seen that our bus matrix synthesis approach results in significant matrix component savings, ranging from 2.1× to 3.2× when compared to a maximally connected bus matrix, and from 4.6× to 9× when compared with a full bus matrix.

In the present and near future, we believe that the bus matrix communication architecture can efficiently support MPSoC systems with tens to hundreds of cores with several data throughput constraints in the multiple gigabits per second range. However, for very large MPSoC systems in the future, bus-based communication systems will suffer from unpredictable wire cross-coupling effects, significant clock skews on longer wires and serious routability issues for multiple wires crossing the chip in a non-regular manner. Network-on-chip (NoC) based communication architectures, with a regular wire layout and having all links of the same length, offer a predictable model for wire cross-talk and delay. This predictability will permit aggressive clock rates and support much larger data throughputs. Therefore we believe that for very large MPSoC systems in the future having several hundreds of cores, a packet-switched NoC communication backbone would be a more suitable choice.

V. Conclusion

In this paper, we presented an approach for the automated synthesis of a bus matrix communication architecture for MPSoC designs with high bandwidth requirements. Our synthesis approach satisfies all throughput performance constraints in the design, while generating an optimal bus matrix topology having a minimal number of busses, as well as values for parameters such as bus speeds, OO buffer sizes and arbitration strategies. Results from the synthesis of an AMBA3 AXI [21] based bus matrix for four MPSoC applications from the networking domain show a significant reduction in bus count in the synthesized matrix when compared with a full bus matrix (up to 9×) and a maximally connected reduced matrix (up to 3.2×). Our approach is not restricted to an AMBA3 [21] matrix based architecture and can be easily extended to synthesize CoreConnect [2] and STBus [3] crossbars as well.

Acknowledgements

This research was partially supported by grants from SRC Contract 1330, Conexant Systems, CPCC fellowship and UC Micro (03-029).

References

[1] ARM AMBA Specification (rev2.0), *www.arm.com, 2001*

[2] "IBM On-chip CoreConnect Bus Architecture", *www.chips.ibm.com*

[3] "STBus Communication System: Concepts and Definitions", *Reference Guide, STMicroelectronics, May 2003*

[4] "Sonics Integration Architecture, Sonics Inc", *www.sonicsinc.com*

[5] L.Benini, G.D.Micheli, "Networks on Chips: A New SoC Paradigm", *IEEE Computers, pp. 70-78, Jan. 2002*

[6] J. Henkel, et al, "On-chip networks: A scalable, communication-centric embedded system design paradigm", *VLSI Design, 2004*

[7] M. Nakajima et al. "A 400MHz 32b embedded microprocessor core AM34-1 with 4.0GB/s cross-bar bus switch for SoC", *ISSCC 2002*

[8] S. Murali, G. De Micheli, "An Application-Specific Design Methodology for STbus Crossbar Generation", *DATE 2005*

[9] V. Lahtinen, et al, "Comparison of synthesized bus and crossbar interconnection architectures", *ISCAS 2003*

[10] K.K Ryu, E. Shin, V.J. Mooney, "A Comparison of Five Different Multiprocessor SoC Bus Architectures", *DSS 2001*

[11] M. Loghi, et al "Analyzing On-Chip Communication in a MPSoC Environment", *DATE 2004*

[12] M. Gasteier, M. Glesner "Bus-based communication synthesis on system level", *ACM TODAES, January 1999*

[13] S. Pasricha, N. Dutt, M. Ben-Romdhane, "Automated Throughput-driven Synthesis of Bus-based Communication Architectures", *In Proc of ASPDAC 2005*

[14] S. Pasricha, N. Dutt, E. Bozorgzadeh, M. Ben-Romdhane, "Floorplan-aware Automated Synthesis of Bus-based Communication Architectures", *In Proc. of DAC 2005*

[15] K. Srinivasan, et al, "Linear Programming based Techniques for Synthesis of Network-on-Chip Architectures", *ICCD 2004*

[16] D. Bertozzi et al. "NoC synthesis flow for customized domain specific multiprocessor systems-on-chip", *IEEE TPDS, Feb 2005*

[17] O. Ogawa et al, "A Practical Approach for Bus Architecture Optimization at Transaction Level", *DATE 2003*

[18] SystemC initiative. *www.systemc.org*

[19] S. Pasricha, N. Dutt, M. Ben-Romdhane, "Fast Exploration of Bus-based On-chip Communication Architectures", *In Proc. of CODES+ISSS 2004*

[20] S. Pasricha, N. Dutt, M. Ben-Romdhane, "Bus Matrix Communication Architecture Synthesis*", CECS Technical Report 05-17, October 2005*

[21] ARM AMBA AXI Specification *www.arm.com/armtech/AXI*

[22] S. Pasricha, N. Dutt, M. Ben-Romdhane, "Extending the Transaction Level Modeling Approach for Fast Communication Architecture Exploration", *In Proc. of DAC 2004*

[23] K. Lahiri et al, "Efficient exploration of the SoC communication architecture design space", *ICCAD 2000*

[24] S. Pasricha, "Transaction Level Modeling of SoC with SystemC 2.0" Synopsys User Group Conference (SNUG 2002), Bangalore, May 2002