Communication Architecture Synthesis of Cascaded Bus Matrix

Junhee Yoo
Dongwook Lee
Seoul National University,
Seoul, Korea
ihavnoid@poppy.snu.ac.kr
peropero@poppy.snu.ac.kr

Sungjoo Yoo
Samsung Electronics,
Yongin, Korea
sungjoo.yoo@samsung.com

Kiyoung Choi
Seoul National University,
Seoul, Korea
kchoi@azalea.snu.ac.kr

Abstract – For high frequency on-chip communication architecture design, we propose cascaded bus matrix-based solutions. Due to the huge design space in cascaded bus matrix design, it is crucial to perform an efficient design space exploration. In our work, we present a simulated annealing-based design space exploration method. For an efficient representation of bus topology, we propose an encoding method called traffic group encoding and apply it to AMBA3 AXI-based bus system design. In addition, we propose a method of two-step simulated annealing to improve the quality of results. Experimental results show that the proposed methods allow designing complex communication architectures (ones with up to 31 masters and 71 slaves) with high frequency constraints to which existing methods could not give solutions.

I. Introduction

It is commonly predicted that a single SoC will have hundreds of IPs (CPUs, DSPs, ASIPs, coprocessors, etc.) and its operating frequency will continue to increase [1]. The increasing number of IPs and operating frequency impose a significant problem on on-chip communication architecture design. Conventional designs based on shared bus have limitations in accommodating a large number of IPs with high operating frequency.

Recently, bus matrix based (or crossbar based) designs are getting increasingly popular for on-chip communication, due to its high throughput and support for various popular bus protocols, such as for ARM's PL300/301 [2], Synopsys DesignWare AMBA 3 fabric,[3] and Sonics' SonicsMX [4].

There have been several studies on bus matrix-based communication architecture design [5]. However, their approach is based on the usage of one central bus matrix, along with many local shared buses. Although such a configuration is practical in that it is a natural migration from the multi-layer bus design, we predict that it will start to suffer from new problems as the number of IP's increases, due to the following reasons:

- A bus matrix's size is proportional to the number of masters times the number of slaves. Although it is possible to build a sparse matrix to remove unnecessary connections, and group several masters and/or slaves to a local bus as in [5], a local bus's size cannot increase indefinitely, due to many problems such as bandwidth and clock frequency. Therefore, the central bus matrix may become too large when used in a large system with many IP's.
- If the bus matrix's size increases, the added logic delay will lower the clock frequency of the bus matrix. This is because as the number of ports connected to the bus matrix increases, the logic depth of the bus increases. Based on our experience, the logic depth in the arbiter and the decoder of the bus matrix increases as the number of IP's increases.
- Although higher clock frequency may be achievable by pipelining the bus matrix's internal architecture, existing solutions have limitations in the pipelining. For instance, ARM's PL301 gives address decode as the only internal pipeline option while pipeline points in Sonics' SMX affect only the interface timing between the bus matrix (SMX) and the IP.

In order to resolve the problem of designing on-chip

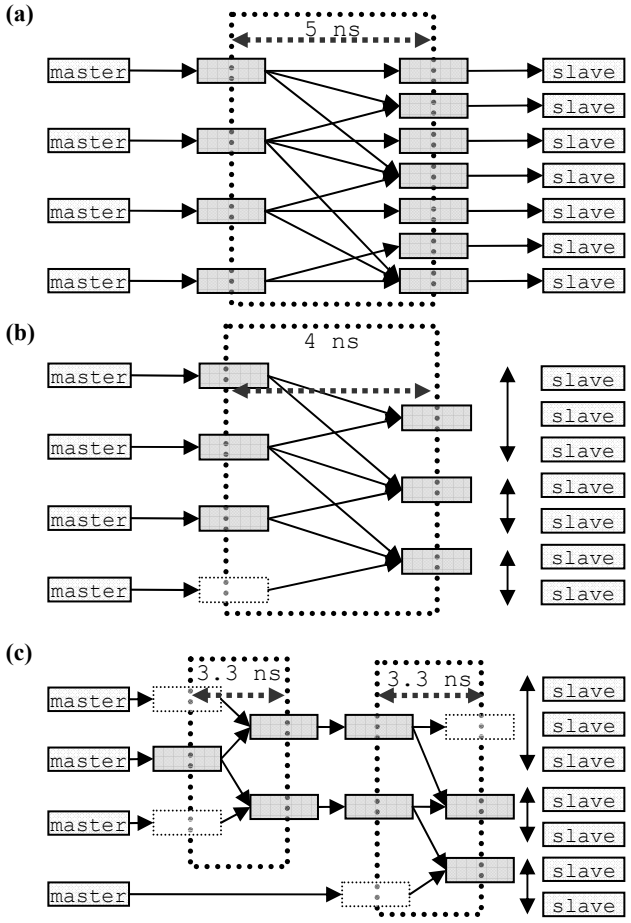

Figure 1. A motivational example.

communication architecture with high frequency constraints and a large number of IP's, we present a method of designing communication architecture with a cascaded matrix architecture, which consists of multiple smaller bus matrices rather than one large bus matrix connecting the local buses in the existing studies.

A. Motivational Example

Figure 1 shows an example system. The system consists of one bus matrix, which connects 4 masters and 7 slaves. If all components are directly connected to the bus matrix, the bus matrix (shown as a dashed rectangle in the figure) becomes excessively large. The logic delay will also be large (e.g., 5ns). However, by grouping some of the slaves into local buses and connecting it to the bus matrix, the bus matrix size can be reduced. The logic delay may be reduced (e.g., 4ns).

If we design the same system with multiple bus matrices as shown in Figure 1 (c), we can see that the sizes of the bus matrices decrease thereby giving a smaller logic delay (e.g. 3.3ns), at the expense of an additional cycle of latency. Thus, a higher-frequency communication architecture can be obtained.

B. Our Contribution

Cascaded bus matrix design has a huge design space since, with a given number of IPs, the number of possible compositions of small matrices is very large. Thus, we need a method of efficiently exploring the huge design. In our work, the design space exploration is based on simulated annealing. In this paper, we propose two methods for efficient design space exploration: bus topology encoding and two-step simulated annealing.

C. Paper Organization

The paper is organized as follows. Section II will present related work. Section III defines the problem, and Section IV describes the overall architecture of the synthesis flow for solving the problem. Section V explains the encoding method – how to describe the system. Section VI shows the experiment results, and Section VII concludes the paper, along with some possible future work to be done.

II. Related Work

There have been many communication architecture synthesis flows for conventional bus protocols [5,6,8,9] and network-on-chips[7,10]. Many of these automated synthesis flows focus on mapping each IPs on a pre-defined interconnect topology, thus reducing the flexibility on transforming the interconnect topology itself. Although there also has been some researches on transforming the interconnect topology itself, the amount of freedom on topology generation is still limited.

An example is the FlexBus [8,9], which has multiple local buses connected to each other via master-master bridges. The IPs are connected to one of the local buses, and the connectivity can be reconfigured on-the-fly. However, the authors didn't consider methods of changing the global communication architecture's topology, and uses a fixed global communication topology.

Another example is the method described on [5]. The authors generated a system with a single bus matrix, and local buses are attached to each of the ports on the matrix. We extend the approach proposed by [5], by synthesizing communication architectures with multiple bus matrices rather than only one. Although this will introduce additional complexity on the synthesis flow, we expect that this additional complexity will worth the cost on large systems with hundreds of master/slave ports.

[13] also deals with a similar problem, which connects components into smaller local buses, and uses switches to forward traffic from one bus to another bus. Our work is a more aggressive approach compared to [13], as their work does not consider topologies with multiple cascaded switches.

III. Problem Definition

Our problem is to find a cascaded bus matrix which satisfies the given communication specification, i.e. communication behavior and the requirements of bandwidth and latency.

A. Communication trace graph

As the communication behavior, we use communication trace graph. A communication trace graph CTG = (V, E) is a directed graph, where each node of the graph is a port that can be connected to the network, and the directed edge represents a communication trace between the two nodes. We use a CTG of a special case - a node may be an 'outgoing' node, which has outgoing edges only, or an 'incoming node', which has incoming edges only. Thus, the CTG becomes a bipartite graph. For IPs with both incoming communication and outgoing communication, the IP can be modeled as two separate nodes. In this paper, the edge direction represents the direction of the transaction. Thus, masters are 'outgoing' nodes, and slaves are 'incoming' nodes.

There are two function that represents bandwidth and latency. For every e, $bw(e)$ denotes the bandwidth of the traffic, and for every e, $maxdelay(e)$ denotes the maximum latency (in microseconds) of the traffic.

B. Implementation Graph and Path Function

To make a formal definition, we define the implementation graph $IG = (IV, IE)$ as a directed graph as a graph which has the following properties;

- $V \subseteq IV$
- For all $e = (v_i, v_j) \in E$, There exists a path from v_i to v_j on IG. This path will be represented as $PATH(e)$, which will be defined later.

Implementation graphs represent the topology of the communication architecture which satisfies the communication requirements of the given CTG. The nodes

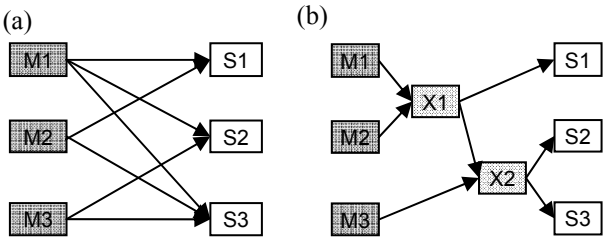

Figure 2. (a) An example CTG, and (b) a IG for the CTG.

represent the objects in the communication architecture, and the edges represent the communication channel between the objects. The first property means that the nodes of the implementation graph includes the ports on the CTG, and the second property shows that the implementation graph requires being able to satisfy all the communication requirements of the CTG.

Like the CTG, implementation graphs also have two functions for bandwidth and latency. For each edge $ie \in IG$, $maxbw(ie)$ represents the bandwidth that the edge can sustain, and $delay(ie)$ represents the transmission delay required for data transmission. In real world situations, $maxbw(ie)$ is determined by various variables, such as data bus width or clock frequency, and $delay(ie)$ is determined by the latency caused by the various communication components on the architecture.

Additionally, we define the path function $PATH : E \rightarrow \{IV\}$, where $PATH(e)$ is a path on the implementation graph. The path function is the mapped result of the CTG on the implementation graph – the path function represents the path the traffic flows on the communication architecture.

Figure 2 shows an example CTG and an example IG for the CTG. For each edge on the CTG, there is a corresponding path on the IG. For example, the CTG edge from M1 to S1 corresponds to the path M1->X1->S1, and the CTG edge from M1 to S3 corresponds to the path M1->X1->X2->S3.

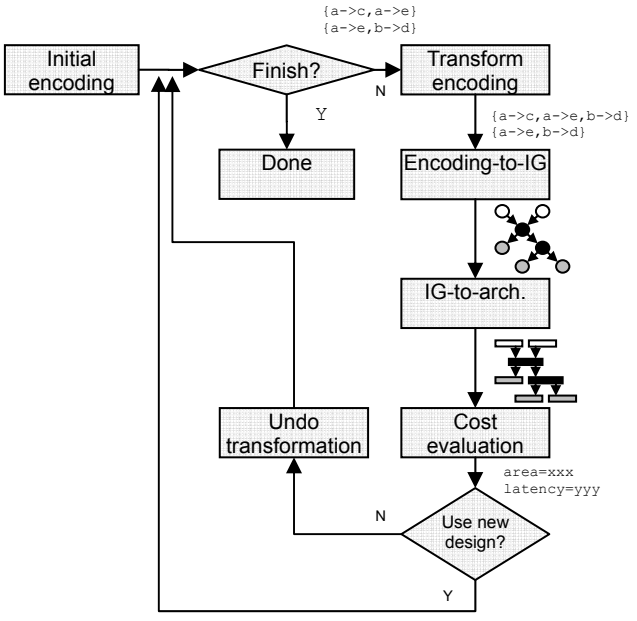

Figure 3. The communication architecture synthesis flow

C. Problem definition

The communication architecture synthesis problem can be defined as:

Given:

- a communication trace graph CTG=(V,E)
- the bandwidth function $bw(e)$, and
- the latency requirement function $maxdelay(e)$,

Find the $IG = (IV, IE)$ for CTG and the corresponding path function with the minimum cost, which

- (latency requirement) for every CTG edge $e \in E$,

$$\sum_{ie \in PATH(e)} delay(ie) \leq maxdelay(e),$$

- (bandwidth requirement) for every IG edge $ie \in IE$,

$$\sum_{ie \in PATH(e)} bw(e) \leq maxbw(ie)$$

Informally, the goal of communication architecture synthesis is to generate an implementation graph from the given CTG, which satisfies the latency requirement and the bandwidth requirement. The method of evaluating the constraints will be case-dependent, and the later chapters will explain it for our case.

IV. Communication Architecture Synthesis Flow

Figure 3 shows our communication architecture design flow. The communication synthesis flow is a simulated annealing flow, which tries to minimize the communication architecture's size, while meeting the latency requirement of each of the traffics.

A. Input specification

The input specification of the design flow is given as a CTG, which can be given by the designer or derived by profiling the behavioral specification. In cases when multi-mode specifications are given, methods proposed on [10] can be used. The ports have additional properties such as clock speed and data bus width, so that the synthesizer can add clock conversion / data width conversion bridges on points where it is required.

B. Encoding for bus topology exploration

The biggest problem on implementing the simulated annealing flow is that it is difficult to implement a transition function – that is, defining the neighbors of the annealing state. This is mainly because an arbitrary topology cannot be a feasible solution – the topology should at least have communication paths that the CTG requires. Therefore, applying arbitrary transformations will not work.

A possible solution would be by transforming the topology randomly, and then repair the topology so that the solution becomes feasible. However, this is also non-trivial, since this will require a repair function which requires

generating a solution that has a similar cost (i.e. the gate count of communication architecture) to the original topology.

In order to resolve this problem, we take an indirect approach of using a different encoding method which always yields a feasible solution. The IG is generated from the encoding. Thus, a move in simulated annealing corresponds to a transformation of the encoding, rather than the transformation of the bus topology itself. The encoding method and the detailed implementation will be discussed in Section V.

C. Implementation graph to architecture

On each annealing phase, using the transformed IG, the corresponding bus architecture is generated. Then, the timing analysis of the bus architecture is performed. The timing analysis of the logic delay of bus matrices is based on the bandwidth requirements and data width of the bus matrices. The timing analysis will determine (1) how to apply existing pipelining solutions to the generated architecture and (2) the penalty in the cost calculation of Section IV.D.

For instance, in the case of AXI PL300-based cascaded matrix architecture [2], register slices are added into the AXI channels wherever the timing requirement cannot be met. However, if there are any cases that the timing cannot be fixed even with enough register slices, a penalty – the negative slack of the channel multiplied by a constant value – is added to the cost function.

The timing information of the bus matrix is prepared before the communication architecture synthesis. We will report how we prepared the timing information in our experiments with ARM AXI bus components on Section VI.

D. Cost function of communication architecture

The objective of communication architecture design is to find a communication architecture that minimizes the area (in gate counts) while satisfying the latency and bandwidth requirements of the CTG, and make all the communication channels meet the timing requirements. The cost function in simulated annealing is defined as:

$$cost = gate\_count \cdot e^{penalty}$$

where $gate\_count$ represents the estimated gate count of the communication architecture. $penalty$ is the penalty value, which is 0 when all the timing requirements and latency requirements are satisfied, and increases linearly as the number of paths that violates the timing and/or the number of CTG edges that surpasses the latency requirement increases.

Due to the exponential nature of the penalty value, the annealing is likely to be stuck on a local optimum. Thus, we use two cost functions during the simulated annealing. Initially, the cost function in simulated annealing is $penalty$. However, as soon as $penalty$ converges to zero, the cost function is changed to $gate\_count \cdot e^{penalty}$. This helps the solution to converge more quickly to a feasible one.

V. The Encoding Method and Two-Step Simulated Annealing

A. Defining the traffic group encoding

In order to define the encoding, we define two more terms:

- We define a Traffic group as an unordered set of edges from the CTG – that is, for a traffic group TG, $TG \subseteq E$.
- We define a Traffic group encoding(TGE) as an ordered set of traffic groups – that is, $TGE = \{TG_i \mid TG_i \subset E, 1 \le i \le n\}$

A traffic group corresponds to a switching element, such as a bus, crossbar, or a router. A CTG edge in the traffic group represents that the traffic represented by the CTG passes the switching element.

The traffic group encoding represents all the switching elements on the system. The order represents the direction of traffic flow – that is, if $e \in TG_i, e \in TG_j$ and $1 \le i < j \le n$, the final system has a path from the node represented by TG_i to the node represented by TG_j.

We define that $TGE = \{TG_i \mid TG_i \subset E, 1 \le i \le n\}$ from the CTG = (V, E) generates $IG = (IV, IE)$ and the path function $PATH(e)$, if and only if:

- There is a one-to-one mapping between a node v_i and $TG_i \in TGE$, where $v_i \in IV$ and $v_i \notin V$
- For any $e \in E$, $iv \in PATH(e)$ if and only if $e \in TG_i$.
- For any $e \in E$, if $e \in TG_i, e \in TG_i (1 \le i < j \le n)$, there exists a path from v_i to v_j.
- For any $e = (v_x, v_y) \in E$, if $e \in TG_i$, there exists a path from v_x to v_i, and exists a path from v_i to v_y.

The first property means that every traffic group corresponds to a node on the IG, which is an interconnect component, such as crossbars, buses, or bus matrices. The second property means that if the CTG edge exists on a traffic group, the node is part of the path that the traffic takes. The last two properties mean that the traffic flows only to the increasing order of the node number, starting from the source to the destination.

The advantage of using TGEs instead of IGs for general optimization algorithms is that TGEs always lead to an implementation that satisfies the CTG. This removes the requirement of a repair function, which can be quite complicated, because checking the CTG requirement for an arbitrary graph may be quite expensive.

For example, the IG from Figure 2 is generated by the following TGE:

```
TGE= {(M1->S1), (M1->S2), (M1->S3), (M2->S1), (M2->S3)},
     {(M1->S2), (M1->S3), (M2->M3), (M3->S2), (M3->S3)}
```

B. Generating System Architecture from Encoding

Once the implementation graph and traffic group encodings are defined, generating IGs from TGEs become

trivial. Figure 4 shows the pseudo-code of tge2ig, which generates an IG from TGE.

To support multiple data widths and multiple clock domains, each group on the TGE has two additional properties – clock speed and data width.

The simulated annealing starts with an empty TGE with the default clock speed and default data width – that is, a point-to-point communication architecture which resembles a fully-connected bus matrix. During the simulated annealing, the following transformations are randomly done:

- Creating a random group with two random CTG edges
- Removing a random edge from a random group
- Merging two random groups into one
- Adding all edges from a random group to another random group ($TG_i \leftarrow TG_i \cup TG_j$)
- Removing edges in a random group from a random group ($TG_i \leftarrow TG_i - TG_j$)
- Changing a random parameter (clock speed or data width) of a random group. For changing clock speeds, we determine the candidate clock speeds by using the clock speeds of each individual IPs in the system, and use the GCD (greatest common divisor) and LCMs (least common multiplier) of each clocks' pair. For example, if some IPs operate on 200MHz while others run on 300MHz, then the crossbars may use 100 MHz, 200MHz, 300MHz, or 600MHz.
- Adding/removing all edges from the same random master/slave, and
- Changing the order of two random groups

Once the IG is generated, the system architecture is generated from the IG. This step is done using these steps:

- Each switching element on the IG is mapped to a PL300 interconnect. The edges are mapped to an AXI channel.
- If the two AXI ports have different parameters (such as different data widths), conversion bridges are added to the communication channel. Figure 5 shows an example case. For example, if there is a connection between a 300MHz master with 32-bit data bus, and a 400MHz slave with 64-bit data bus, an asynchronous bridge and a data width converter (expander) is added in between the two components. Likewise, other types of bridges are added in between ports with different characteristics.
- There are some cases that the required bandwidth of the channel surpasses the channel's capacity. For those channels, another channel is replicated between the two nodes on the IG. Although this practice is very rare in real-world designs, we found that this is unlikely to be a final solution, because a smaller design usually can be generated by splitting one of the two PL300s into two smaller ones.
- Timing analysis is done. For paths that cannot meet the timing requirement, register slices are added, thus pipelining the communication path.

```
function tge2ig(Graph ctg, Array<Set<Edge> > tge)
{
  Graph ig;
  Hashtable<Edge,Vertex> positionTable;
  foreach(Edge e from ctg) {
    positionTable.add(e, e.source);
    ig.addNodeIfDoesntExist(e.source);
    ig.addNodeIfDoesntExist(e.destination);
  }
  foreach(Set<Edge> set from tge) {
    ig.addNode(set);
    foreach(Edge e from set) {
      Vertex prev = positionTable.get(e);
      ig.addEdgeIfDoesntExist(prev, set);
      positionTable.set(e, set);
    }
  }
  foreach(Edge e from ctg) {
    Vertex prev = positionTable.get(e);
    ig.addEdgeIfDoesntExist(prev,
      e.destination);
  }
  return ig;
}
```

Figure 4. A C++-like pseudo-code of tge2ig

C. Alternative bus-matrix based implementation

As an alternative to the TGE-based method, we have implemented two more methods:

- (2-layer bus matrix) Implement the shared bus and the bus matrix's port using a PL300 bus matrix. This method also uses cascaded bus matrices, but the number of levels the crossbars are cascaded is limited to two. (Figure 6 (a))
- (single bus matrix) Implement the shared bus by using bus matrices with one master port and multiple slave ports, or bus matrices with one slave port and multiple master ports. (Figure 6 (b))

Both use the same encoding method, which we will call port partition encoding (PPE). We define the PPE as a pair of sets, where the first set is the partition of all the outgoing

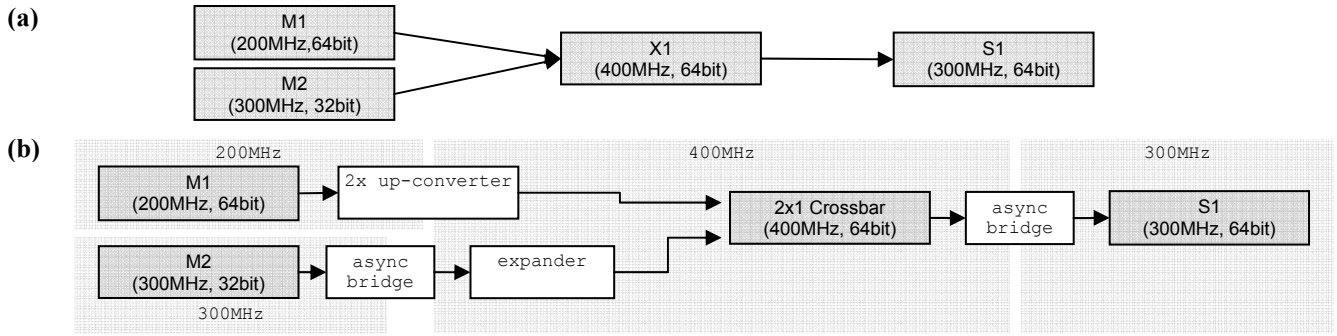

Figure 5. (a) an implementation graph, and (b) a corresponding architecture

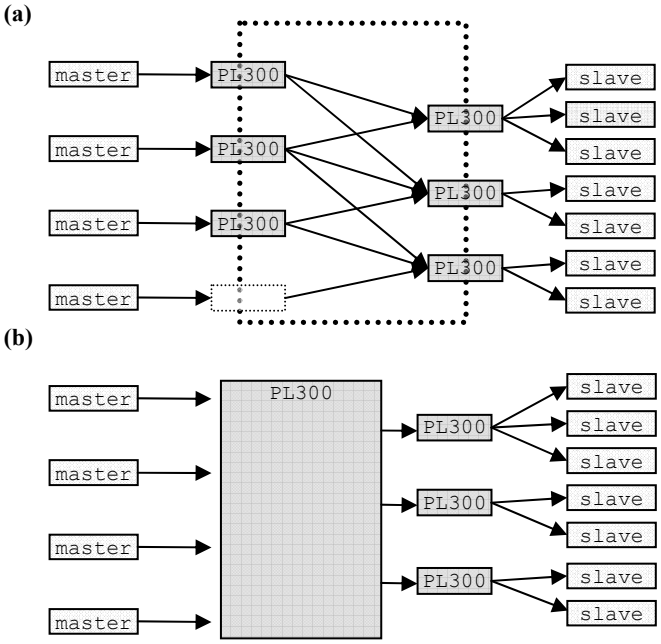

Figure 6 Two alternative bus matrix architectures

nodes (which are masters) from the CTG, and the second set is the partition of all the incoming nodes (which are slaves) from the CTG.

Generating the IG from PPE is trivial. For the 2-layer bus matrix method, a group on the partition from the PPE corresponds to a node of the IG, which connects the nodes in the group, which is a PL300 on the final architecture. For each partition, if a master on a group has traffic to a slave on another group, a connection is made between the two groups.

For the single bus matrix method, a group on the partition on the PPE corresponds to the shared bus which is connected to the ports on the partition. Each of the local buses is connected to the central bus matrix.

D. Two-step Simulated Annealing

An ideal simulated annealing method gives the optimal solution with a long runtime. However, if the solution space is too large, it may require an infeasible amount of computation power to search the large solution space. Thus, we use a two-step simulated annealing. First, the 70% of the annealing is done within a restrictive set of bus topologies. The first step may give a good starting solution to the second step. Then, the last 30% is done starting with the result of the first step, with a method that has a wider design space. We found that this approach might help, since TGE-based architectures have a large design space that requires a large number of annealing steps to yield a good enough solution. Using the two-step approach will help the solution to converge more quickly.

The two step approach is done by first starting the annealing using one of the PPE-based architectures. Then, after the 70% of the annealing finishes, the architecture is encoded to TGE, and the remaining 30% of the annealing is done using the TGE representation.

VI. Experiment results

The communication architecture synthesis flow uses the ARM's AXI components [2]. The ARM AXI components that we use for generating the system architecture include:

- PrimeCell AXI Configurable Interconnect (PL300),
- PrimeCell Infrastructure AMBA 3 AXI Asynchronous Bridge (BP132), Downwards-synchronizing Bridge (BP133), and Upwards-synchronizing Bridge (BP134) for clock speed conversion,
- PrimeCell Infrastructure AMBA 3 AXI Downsizer (BP131) and Expander, for data bus width conversion, and
- PrimeCell Infrastructure AMBA 3 AXI Register Slice (BP130) for pipelining the interconnect where the critical path is too long, so that the timing can meet the requirement.

The timing and area information of the ARM IPs were obtained by synthesizing the RTL code using Synopsys Design compiler using Samsung's 90nm low-voltage ASIC process. The gate counts and port I/O delays were obtained from the synthesis reports of Design Compiler. The IPs was synthesized using all possible combinations of configuration parameters. For example, in the case of PL300s, we synthesized 286 configurations with 1 to 9 master ports, 1 to 16 slave ports, and two possible data bus widths (32 or 64).

Our experiment was done using 120 synthetic CTGs generated using a CTG generator. This was mainly because we could not find any publicly available CTG large enough that our approach can be found useful. The CTGs that we used consists of 6 masters and 11 slaves for the smallest ones, and 31 master and 71 slaves for the largest ones. The CTG generator first generates a reasonable number of components – processors, DSPs, custom logic, peripherals, or SRAM/DRAM blocks, and CTG edges are generated according to each IP's characteristic. For example, DSPs typically have two master ports, where one of them have a low-bandwidth read access path to a SDRAM block (which represents instruction stream), and the other port with high-bandwidth read/write accesses to local buffers and global SDRAM blocks (which represents data streams).

Each of the CTGs were synthesized using four synthesis methods –TGE, single bus matrix (SBM), two-layers of bus matrices(2BM), and an two-phase approach, where the first 70% of the annealing is done using either 2BM or SBM (depending on the number of nodes), and the last 30% is done using TGE. Each of the simulated annealing ran $10 \cdot sizeof(master) \cdot sizeof(slave) \cdot sizeof(traffic)$ iterations, where $sizeof(master)$, $sizeof(slave)$ and $sizeof(traffic)$ each represents the number of masters, number of slaves, and the number of CTG edges, respectively.

We have implemented the communication architecture synthesis flow in Java 5.0, which executes on Sun's Java VM 5.0[11]. Since running 480 synthesis tasks requires a lot of computation power, we implemented a simple

in-house synthesis farm, which consists of four x86 machines with different processors, different amount of RAM, and different operating systems.

TABLE I

Synthesis result of 120 CTGs[1]

Method		SBM	2BM	TGE	Two phase
All designs	Average size (geometric mean)	-	6.403	7.172	5.325
	Number of successful synthesizes	78	120	119	120
	number of best results among 4 methods	33	1	31	55
small 27 designs (< 25 ports)	Average size (geometric mean)	2.991	3.641	3.097	3.159
	Number of successful synthesizes	27	27	27	27
	number of best results among 4 methods	9	0	4	14
large 22 designs (> 45 ports)	Average size (geometric mean)	-	9.434	15.799	8.309
	Number of successful synthesizes	2	22	22	22
	number of best results among 4 methods	1	0	5	16

Table I shows the experiment results for the 120 designs. The three methods that generate cascaded bus matrices successfully generate designs that meet the timing requirements for all 120 designs, while the SBM method failed to generate a design for approximately half of the CTGs. The situation became worse when the number of components increased to more than 45, where almost none of the CTGs could be successfully synthesized. This shows that it is required to use multiple cascaded bus matrices when the system gets larger.

Among the three cascaded bus matrix methods, the two-phase approach's result was better than both TGE-based method and 2-layer bus matrix methods. The reason that TGE's average was much higher was because the TGE method and 2BM method had a lower probability to generate a reasonable solution – thus, the result was 10x bigger for some CTGs, while other CTGs had marginal difference (around 15% size difference).

For all four methods, execution time was tens of seconds for the smallest designs, and around 2 hours for the largest designs.

VII. Conclusions

In this paper, we present a novel approach based on cascaded bus matrix to synthesize communication architectures for large systems with nearly 100 IP's. We present two methods, an encoding method and two-step simulated annealing, for efficient design space exploration. Experiment results show that our approach is able to synthesize communication architectures with hundreds of components, which none of the previous approaches were capable.

Still, there is a lot of more research to do. Since our optimization flow requires quite a lot of computation power, and the architecture required to synthesize is expected to grow exponentially, better optimization approaches are needed. We plan to extend our approach using newer meta-heuristic optimization algorithms, such as the popular genetic algorithm, or ant colony optimization [12].

Additionally, since our flow does not run simulations on every phase of performance and cost calculation, but estimates, especially, the performance, there must be an accurate system performance modeling method. We plan to add statistical communication modeling methods, so that we can obtain accurate communication architecture performance without simulating the system within a small error margin.

[1] The size of the generated architecture is in the number of gates normalized to an arbitrary value. We could not release the number of gates, due to the licensing agreements with our IP vendor.

References

[1] International Technology Roadmap for Semiconductors 2005, Design, http://public.itrs.net/

[2] ARM documents – System-on-chip http://www.arm.com/documentation/SoC/index.html

[3] Synopsys DesignWare IP – AMBA solutions http://www.synopsys.com/products/designware/amba_solutions.html

[4] SonicsMX datasheet, available at http://www.sonicsinc.com/documets/SMX_Data_Sheet.pdf

[5] S. Pasricha, N. Dutt, M. Men-Romdhane, "Constraint-Driven Bus Matrix Synthesis for MPSoC", in proc. of ASPDAC, Jan. 2006

[6] S. Pasricha, N. Dutt, E. Bozorgzadeh, M. Ben-Romdhane. "Floorplan-Aware Automated Synthesis of Bus-based Communication Architectures", in proc. of DAC, June 2005

[7] K. Srinivasan and K. S. Chatha, "A Low Complexity Heuristic for Design of Custom Network-on-Chip Architectures", in proc. of DATE, March 2006

[8] K. Sekar, K. Lahiri, A. Raghunathan, and S. Dey, "FLEXBUS: A high-performance system-on-chip communication architecture with a dynamically configurable topology," in proc. of DAC, June 2005

[9] K. Sekar, K. Lahiri, A. Raghunathan, and S. Dey, "Integrated data relocation and bus reconfiguration for adaptive system-on-chip platforms," in proc. of DATE, March 2006

[10] S. Murali, M. Coenen, A. Radulescu, K. Goossens and G. De Micheli , "Mapping and Configuration Methods for Multi-Use-Case Networks on Chips", in proc. Of ASPDAC, Jan 2006

[11] Java 2 SE from Sun Microsystems, http://java.sun.com/javase/index.jsp

[12] M. Dorigo and T.Stützle, *Ant Colony Optimization*, MIT Press, 2004

[13] T. van Meeuwen et al, "System-level Interconnect Architecture Exploration for Custom Memory Organizations", in proc. Of ISSS, October 2001