BMSYN: Bus Matrix Communication Architecture Synthesis for MPSoC

Sudeep Pasricha, *Student Member, IEEE*, Nikil D. Dutt, *Senior Member, IEEE*, and Mohamed Ben-Romdhane, *Member, IEEE*

***Abstract*—Modern multiprocessor system-on-chip designs have high bandwidth constraints which must be satisfied by the underlying communication architecture. Traditional hierarchical shared bus communication architectures can only support limited bandwidths and are not scalable for very high-performance designs. 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, while satisfying all performance constraints in the design.**

***Index Terms*—Communication system performance, digital systems, high-level synthesis.**

I. Introduction

MULTIPROCESSOR 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 intercomponent data traffic and has a significant impact on the overall system performance.

Traditionally used hierarchical shared bus-based communication architectures such as those proposed by advanced microprocessor bus architecture (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

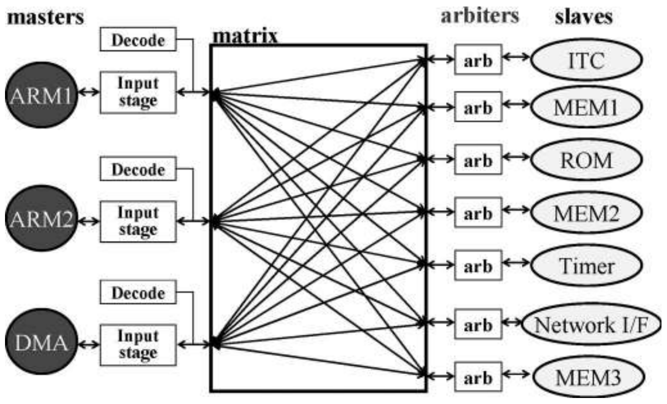

Fig. 1. Full bus matrix communication architecture.

systems. As the number of components connected to a shared bus increases, the capacitive load of the component output pins leads to an increase in signal propagation delay, which puts a limit on the practically achievable bus clock frequency and consequently overall performance in the system [31]. Additionally, a shared bus can only support a single active communication stream at any time, which limits the amount of possible parallelism in hierarchical shared bus-based communication architectures. Point-to-point connections between communicating components can theoretically support several parallel communication streams, but practical limitations on the number of ports and excessive wiring congestion makes such a scheme practical for even fewer components. Network-on-chip (NoC)-based communication architectures [5], such as CHAIN [32] and AEthereal [33], have recently emerged as a promising alternative to handle communication needs for the next generation of high-performance MPSoC designs. These packet switched networks can allow improved physical predictability because of their regular geometries. This regularity is predicted to ease timing closure even in the face of deep-submicrometer effects and also enable support for higher wire clock frequencies for better throughput performance. 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

Manuscript received February 2, 2006; revised June 21, 2006. This work was supported in part by grants from Conexant Systems, Inc., SRC under Contract 1330, CPCC fellowship, and UC Micro under Contract 03-029. This paper was recommended by Associate Editor L. Benini.

S. Pasricha and N. D. Dutt are with the Center for Embedded Computer Systems, University of California, Irvine, CA 92697 USA (e-mail: sudeep@cecs.uci.edu; dutt@cecs.uci.edu).

M. Ben-Romdhane is with Newport Media, Inc., Lake Forest, CA 92630 USA (e-mail: mbenromdhane@newportmediainc.com).

Digital Object Identifier 10.1109/TCAD.2007.891376

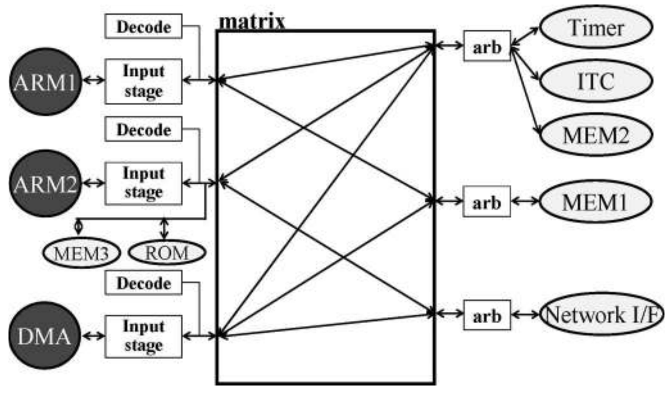

Fig. 2. Partial bus matrix communication architecture.

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 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.

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 in the matrix. 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, which reduces wire congestion and leads to better utilization of bus wires. Additionally, reducing crossbar size also reduces components in the matrix (such as arbiters) and simplifies the design of logic components (such as decoders) which in turn reduces power consumption of the design. This reduction also translates into a reduced bus matrix area footprint on the chip, because of the reduced wiring and bus logic components in the matrix.

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 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 [bus matrix synthesis (BMSYN)], which generates not only the matrix topology, but also communication parameter values for bus clock speeds, OO buffer sizes and arbitration strategies. Most importantly, BMSYN 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 BMSYN 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 shared bus, full crossbar, and partial crossbar topologies of the AMBA and STBus communication architectures, 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], [27]–[30] and NoC architecture synthesis [15], [16], [24]–[26], few efforts have focused on BMSYN. 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 and De Micheli in [8] is the only work that comes closest to our goal of automated BMSYN. 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 [19], [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, which 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, suboptimal 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 BMSYN. First, we formulate the problem and present our assumptions. Next, we describe our simulation engine and elaborate on communication parameter constraints, which guide the BMSYN process. Finally, we present our automated BMSYN approach in detail.

A. Problem Formulation

We start with an MPSoC application 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.

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 $(\mathrm{CTG}) = G(V, A)$ which is a directed graph, where each vertex ν 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 subgraph 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. The concept of a TCP is a useful abstraction to capture critical communication streams, which govern application performance in our approach. All dependencies which affect a critical communication stream are encapsulated as part of a TCP. These TCPs are dependent on the application being considered and are typically derived from the knowledge of standards and protocols supported or from marketing specification documents which specify performance constraints for subsystems that must be satisfied by the application implementation. 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 Gb/s. Note that any edge in a CTG can be part of multiple critical communication streams, in which case the edge is shared between multiple TCPs. It is also possible for an edge which is part of a TCP to have other noncritical communication streams on it. For the purposes of creating a CTG with TCPs for an application, we ignore such noncritical communication streams, since they do not impact critical communication performance.

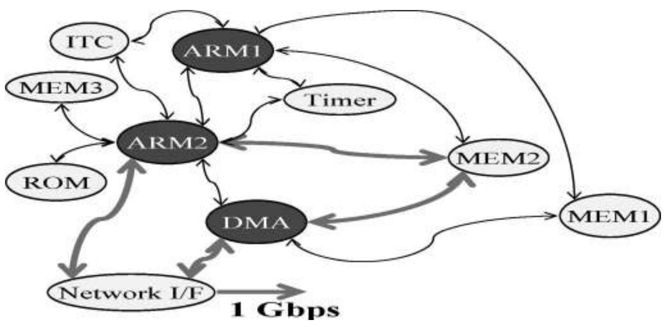

Fig. 3. CTG.

Problem Definition A bus $\boldsymbol{B}$ can be considered to be a partition of the set of components $\boldsymbol{V}$ 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 $\boldsymbol{V}$ is partitioned onto a minimal number of busses $\boldsymbol{N}$ while satisfying 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. However, relying solely on simulation-based exploration can limit the amount of space that can be explored in a reasonable amount of time. As we describe later, BMSYN makes use of a combination of static and simulation-based dynamic analysis to speed up the synthesis process.

For the simulation part of our flow, we capture behavioral models of components and bus architectures in SystemC [18], [22] 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 kHz [13], while running embedded software applications on processor instruction-set-simulator 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

The exploration space for a typical MPSoC bus matrix communication architecture consists of combinations of bus topology configurations with communication parameter values for bus clock speeds, OO buffer sizes, and arbitration schemes. If we allow these parameters to have any arbitrary values, an incredibly vast design space is created. The time required to traverse this space as we search for the most cost-effective configuration (which also satisfies all performance constraints) would become unreasonably large. More importantly, once we manage to find such a system configuration, there would be no guarantee that the values generated for the communication parameters would be practically feasible. To ensure that BMSYN generates a realistic bus matrix communication architecture configuration, we allow the designer to specify a communication parameter constraint set (Ψ). The constraints are in the form of a discrete set of valid values for the communication parameters to be synthesized. 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}). The designer can specify local constraints for a resource if these constraint values are different from the global constraints. Otherwise, the designer can leave the local constraints unspecified, thus allowing the resource to inherit the more generic global constraints. For instance, a designer might set the allowable bus clock speeds for a set of busses 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. As long as Ψ is populated with any discrete set of values for the parameters, our framework will attempt to synthesize a feasible low-cost optimal matrix architecture. However, informed decisions can greatly reduce the time taken for synthesis and help the designer generate a more practical system.

D. BMSYN Synthesis Flow

We now describe our automated BMSYN approach (BMSYN). Fig. 4 gives a high-level overview of the flow. The inputs to the flow include a 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 transaction-level model (TLM) simulation of the system to get application-specific data traffic statistics. This information is then 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 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)

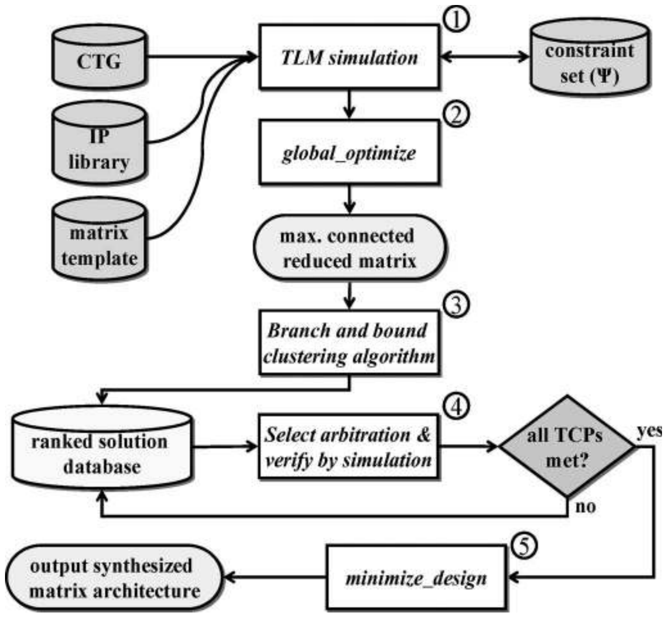
CTG
IP library
matrix template
TLM simulation
constraint set (Ψ)
global_optimize
max. connected reduced matrix
Branch and bound clustering algorithm
ranked solution database
Select arbitration & verify by simulation
all TCPs met?
yes
no
output synthesized matrix architecture
minimize_design
1
2
3
4
5

Fig. 4. Automated BMSYN flow.

and save them in the database. We then use a fast bus cycle accurate simulation engine [13], [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.

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 phase 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 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 ($\Gamma_{\mathrm{TLM}/B}$) from the TLM simulation, for any bus B in the matrix is given by

$$\Gamma_{\mathrm{TLM}/B} = (\mathrm{numT}_B \times \mathrm{sizeT}_B \times \mathrm{width}_B \times \Omega_B)/\sigma$$

where numT is the number of data transactions on the bus, sizeT is the average size of these data transactions, width is the data 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 $\Gamma_{\mathrm{TCP}/B}$ for the bus B

$$\begin{aligned}&\Gamma_{\mathrm{TLM}/B} \geq \Gamma_{\mathrm{TCP}/B}\\ &\therefore \quad \Omega_B \geq (\sigma \times \Gamma_{\mathrm{TCP}/B})/(\mathrm{width}_B \times \mathrm{numT}_B \times \mathrm{sizeT}_B).\end{aligned}$$

The minimum bus clock speed thus found is used to create (or update) the local bus speed constraint set $\Psi_{\mathrm{L(speed)}}$ for 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 nonoverlapping regions [20]. 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 clustering algorithm to cluster slave components to reduce the number of busses in the matrix even further. Note that we do not consider clustering masters in the matrix, in our approach. While clustering masters can result in some savings for simple SoC systems, for the highly parallel, high-performance MPSoC applications that we target, clustering masters can drastically degrade system performance. This is because master clustering adds two levels of contention, one at the master end and another at the slave end, in a data path, which lengthens the completion time for transactions issued by any of the clustered masters. Additionally, clustering masters also severely limits the parallelism in the system, since if one master in a "master cluster" is active with a transaction, for instance a burst of several data transfers, none of the other masters in that cluster can issue transactions. In our experience, even increasing the bus clock frequency to compensate for the reduced parallelism and longer transaction latency in the system does not prevent throughput constraint violations when masters are clustered. However, in-depth analysis of master clustering solutions is outside the scope of this paper.

Before describing the algorithm, we present a few definitions. A slave cluster SC $= \{s_1 \ldots s_n\}$ 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 $\Pi_{\mathrm{SC1/SC2}}$ be a superset of sets of busses which are merged when slave clusters SC1 and SC2 are merged. Finally, for a merged bus set $\beta = \{b_1 \ldots b_n\}$, where $\beta \subset \Pi_{\mathrm{SC1/SC2}}$, 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

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

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.

```
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 $\beta \in \Pi_{\mathrm{SC1/SC2}}$ 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.

Fig. 5 shows the pseudocode 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; 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.

Fig. 6 shows the branch and bound clustering flow for the example shown earlier in Figs. 1–3. Every valid branch in the solution space corresponds to a valid clustering of slave components and is represented by a node in the figure. The nodes annotated with an X correspond to clustering solutions that are eliminated by the bounding function in Fig. 5 for being duplicate solutions; nodes annotated with a B correspond to solutions that do not meet the other criteria in the bounding function. The figures above the nodes correspond to the cumulative weights of the clustering solution. This allows us to determine the quality of the solution—the node with the maximum cumulative weight corresponds to a bus matrix with the least number of busses. The highlighted node in Fig. 6 corresponds to the optimal solution. Fig. 7 shows all the clustering operations for the solution branch corresponding to this node.

The solutions obtained from the static branch and bound clustering algorithm are ranked from best to worst and stored

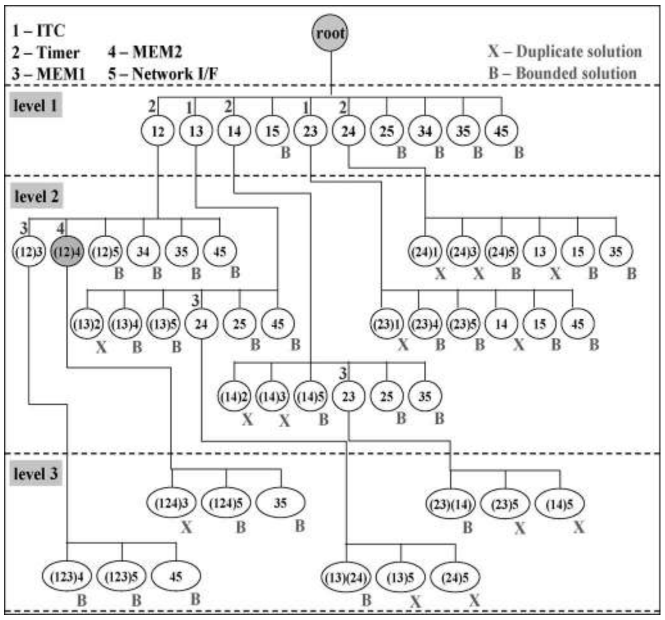

Fig. 6. Branch and bound clustering illustration.

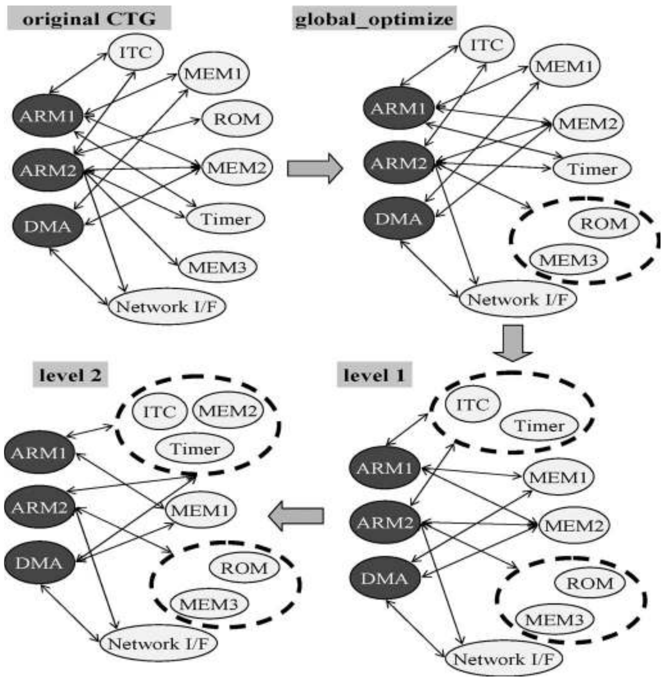

Fig. 7. Flow of clustering operations of best solution, for the example MPSoC system.

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 [13], [19] to verify that the reduced matrix still satisfies all the constraints in the design. The designs are simulated after setting the bus clock frequency values to the maximum allowed by Ψ, for the buses in the matrix. We perform arbitration strategy selection at this stage (from the allowed schemes in the constraint set Ψ). The lowest cost arbitration schemes from Ψ are applied to the slave clusters and the entire design is simulated. If a TCP constraint violation is detected for the edges connected to a slave cluster, we apply other arbitration schemes on the slave cluster, in increasing order of implementation costs, and simulate the design to verify constraint satisfaction. For example, if a simple static priority-based scheme for a slave cluster (with priorities distributed among slave ports according to throughput requirements) results in TCP constraint violations, we first make use of simpler arbitration scheme like round-robin (RR), before resorting to the more elaborate two-level 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 until 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.

TABLE I
NUMBER OF CORES IB MPSoC APPLICATIONS

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

IV. CASE STUDIES

We applied our BMSYN 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 I shows the number of components in each of these applications. The Masters column includes the processors in the design, which are primarily ARM-based microprocessors.

Fig. 8 shows the CTG for the VIPER MPSoC application. For clarity, the TCPs are presented separately in Table II. The ARM1 processor is used for overall system control, generating data cells for signaling, operating and maintenance, communicating and controlling external hardware, and to setup and close data stream connections. The ARM2 processor interacts with data streams from external interfaces and performs data packet/frame encryption and compression. These processors interact with each other via shared memory and a set of shared registers (not shown here). The DMA engine is used

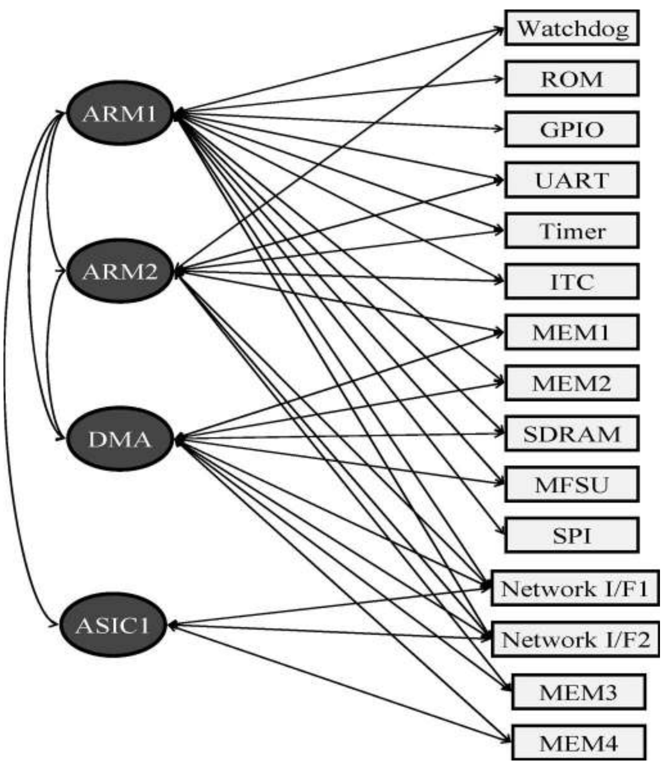

Fig. 8. CTG for VIPER MPSoC application.

TABLE II
TCPs FOR VIPER

IP cores in Throughput Constraint Path (TCP)	Throughput Requirement
ARM2, MEM1, Network I/F1, DMA, MEM3	200 Mbps
ASIC1, MEM4, Network I/F1, DMA, Network I/F2	960 Mbps
ARM2, Network I/F1, MEM4, DMA, SDRAM	640 Mbps
ARM1, MFSU, MEM1, DMA, MEM3, Network I/F1	400 Mbps

TABLE III
CUSTOMIZABLE PARAMETER CONSTRAINT SET (VIPER)

Set	Values
bus speed	33, 66, 100, 133
arbitration strategy	static, RR, TDMA/RR
OO buffer size	1 – 12

to handle fast memory to memory and network interface data transfers, freeing up the processors for more useful work. The application-specific integrated circuit (ASIC1) block performs data packet segmenting and reassembling for multiple concurrent data streams. VIPER also has several peripherals such as a multifunctional serial port interface, a serial flash interface, a universal asynchronous receiver/transmitter block (UART), a general purpose I/O block (GPIO), timers (Timer, Watchdog), an interrupt controller (ITC), proprietary external network interfaces (Network I/F1, Network I/F2), on-chip static-random-access-memory modules (MEMx) and a synchronous dynamic random access memory (SDRAM) memory block.

Table III shows the global customizable parameter set Ψ_G. For the synthesis, we target an AMBA3 AXI [21]-based bus matrix structure. Fig. 9 shows the matrix structure output by our synthesis flow, which satisfies all four throughput constraints in the design (Table II). The data bus width used in the matrix

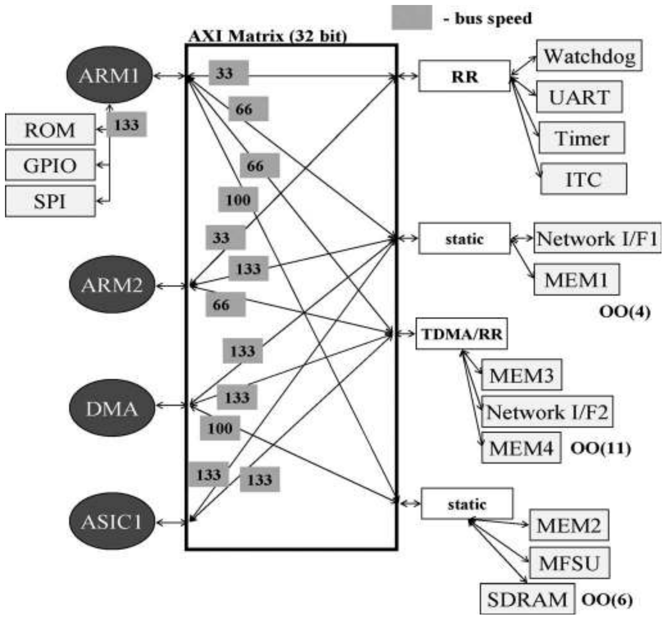

Fig. 9. Synthesized bus matrix for VIPER MPSoC application.

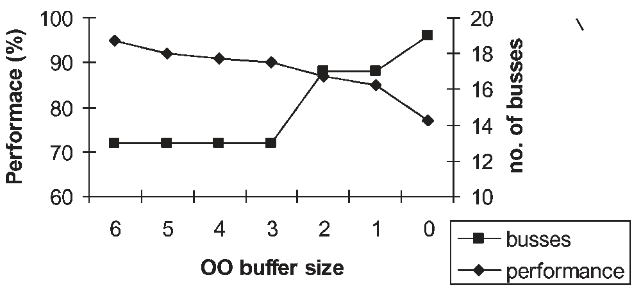

Fig. 10. Effect of changing OO buffer size.

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 architecture used 60 busses, after the global optimization phase (Fig. 4) we were able to reduce this number to 29 for the maximally connected reduced matrix. The final synthesized matrix further reduces the number of busses to as few as 13 (this includes the local busses for the masters) which is almost a 5× saving in the number of busses used when compared to the original full bus matrix.

To demonstrate the importance of synthesizing parameter values during communication architecture synthesis, we performed experiments involving the VIPER application. In the first experiment, we focused on OO buffer size and varied the values for this parameter, to observe the effect on system cost and performance. In VIPER, since MEM1, MEM4, and SDRAM are the only blocks which support OO buffering of transactions, we attempt to change their buffer sizes for this purpose. Fig. 10 shows the results of this experiment. The x axis indicates OO buffer sizes for the components—we keep the same buffer size for the MEM1, MEM4, and SDRAM components and reduce the size along the x axis. The y axis represents a performance metric in the form of application completion time—a decrease in performance is a result of a corresponding increase in application completion time.

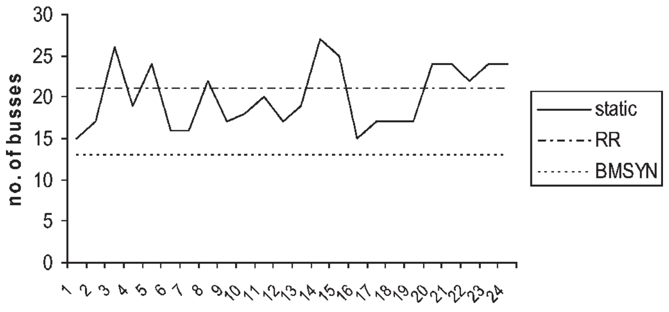

Fig. 11. Effect of ignoring arbitration space during BMSYN.

As the buffer size of the components decreases, it can be seen that the performance of the system deteriorates. With decreasing buffer size, we also find that the minimum number of busses needed to meet all TCP constraints increases. Note that performance numbers in Fig. 10 are obtained for a constant number of busses (13 busses)—adding additional busses tends to improve performance due to the additional parallelism introduced in the system. Of course, the improved performance comes at the price of additional wire congestion (busses) in the matrix. Without taking the concurrence inherent in the target application into consideration (which we do during our synthesis approach), designers might end up fixing OO buffer sizes to large conservative values resulting in increased system cost and an overdesigned system; reducing buffer sizes on the other hand can end up increasing the number of busses in a matrix and again increase system cost.

For the next experiment, we focused on another communication parameter—the arbitration strategy used for conflict resolution by shared slaves. We attempted to determine the effect of not taking this parameter into consideration, as done in [8], during synthesis for the VIPER application. Fig. 11 shows the consequence of such an assumption. BMSYN is our approach applied on the VIPER application, which considers a combination of several different arbitration schemes during synthesis. We compared the result from our approach with two approaches, which perform topology synthesis and keep a fixed arbitration scheme for all shared slaves (which is the approach used in [8]). The first approach uses a fixed static priority-based arbitration (static) and the second uses RR arbitration. For the case of static arbitration, there are 24 static priority combinations possible, because there are four masters in VIPER, and the number of static priority combinations possible for a system having n masters is $n!$ (and consequently $4! = 24$). We synthesize a bus matrix architecture for each of the possible combinations, for the static case.

As can be seen from the figure, if we fix the arbitration policy for the entire system to a single type of arbitration scheme such as static priority or RR, the resulting synthesized system needs a larger number of busses in order to meet all performance constraints in the design, compared to our approach which considers a combination of several different arbitration schemes during synthesis. As shown in Fig. 9, BMSYN uses a combination of several different arbitration schemes such as static priority, RR, and TDMA/RR, which allows better management of traffic conflicts for different subsystems within VIPER and consequently reduces matrix cost.

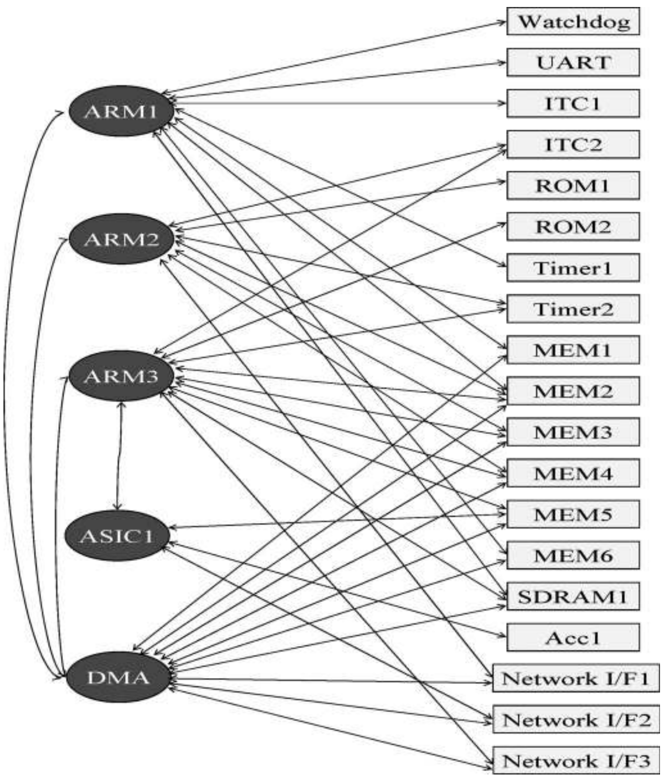

Fig. 12. CTG for SIRIUS MPSoC application.

TABLE IV
TCPs FOR SIRIUS

IP cores in Throughput Constraint Path (TCP)	Throughput Requirement
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

From these two experiments involving OO buffer sizes and arbitration strategies, we can see that these communication parameters can have a significant impact on system cost and performance, and thus should not be ignored in a communication architecture synthesis effort.

Next, we describe the BMSYN process for the SIRIUS application. Fig. 12 shows the CTG for SIRIUS. Again, for clarity, the TCPs are presented separately in Table IV. 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 Data Encryption Standard (DES), Triple Data Encryption

TABLE V
CUSTOMIZABLE PARAMETER CONSTRAINT SET (SIRIUS)

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

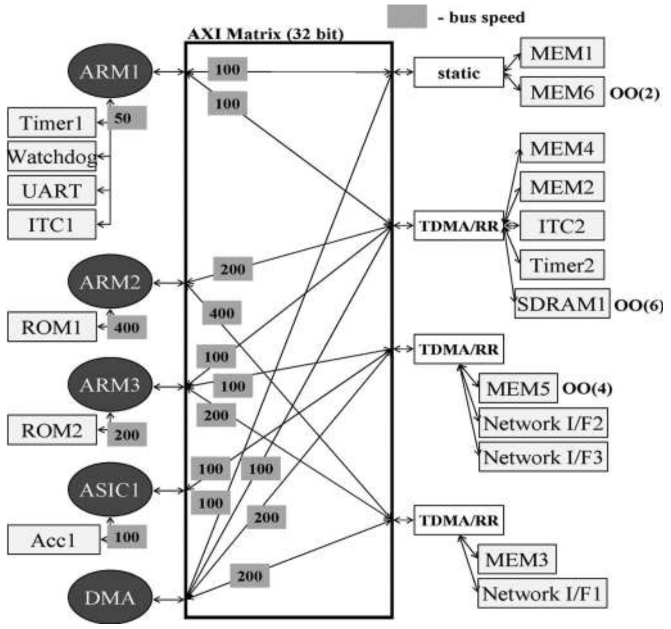

Fig. 13. Synthesized bus matrix for SIRIUS MPSoC.

Standard (3DES), and Advanced Encryption Standard (AES). The DMA is used to handle fast memory to memory and network interface data transfers, freeing up the processors for more useful work. Besides these master cores, SIRIUS also has a number of memory blocks, network interfaces and peripherals such as ITCs (ITC1, ITC2), timers (Watchdog, Timer1, Timer2), UART, and a packet accelerator (Acc1).

Table V shows the global customizable parameter set Ψ_G. For the synthesis, we target an AMBA3 AXI [21]-based bus matrix communication architecture. Fig. 13 shows the matrix structure output by our synthesis flow, which satisfies all six throughput constraints in the design (Table IV). 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 architecture 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× saving in the number of busses used when compared to the original full bus 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 results from three sets of experiments. The first experiment compares the synthesis results of our approach using variable bus frequencies with an approach using a fixed bus frequency for the entire matrix. The second experiment compares our synthesis results with previous work in the area of BMSYN. Finally, the third experiment compares the results of applying our synthesis approach on the four MPSoC applications mentioned earlier in the section.

TABLE VI
NUMBER OF BUSES IN SYNTHESIZED SOLUTION FOR SIRIUS, USING VARIABLE AND FIXED FREQUENCIES IN MATRIX

Frequency in matrix	*Variable (Original case)*	*Fixed (100MHz)*	*Fixed (200 MHz)*	*Fixed (400 MHz)*
Number of Buses	16	constraint violation	18	16

BMSYN allows a designer the flexibility to assign variable bus clock frequencies for each of the buses in the matrix. However, there is an overhead in the form of frequency converters at the interfaces (which might use buffering for timing isolation) when compared to an approach where a fixed bus clock frequency is assigned to all the buses in the matrix.

Table VI compares the synthesis result of applying BMSYN on the SIRIUS application, for the original case with variable bus clock frequencies in the matrix, and for three additional cases, where all the buses in the matrix are assigned a fixed bus clock frequency of 100, 200, and 400 MHz, respectively. It can be seen that for the case where all the buses in the matrix have a single fixed bus clock frequency of 100 MHz in the matrix, no solution can be found which satisfies the throughput constraints in SIRIUS. For the case that uses a single fixed bus clock frequency of 200 MHz in the matrix, the number of buses is more than for the variable frequency case, because of excessive traffic overlap for one of the higher throughput paths, which consequently reduces the clustering and increases the number of buses. For the high single fixed bus clock frequency case of 400 MHz, the solution quality (number of buses) is the same as compared to the variable frequency case, but there are important differences between these two approaches, which are not apparent from the table. The 400-MHz fixed frequency case has the advantage of not needing any frequency converters, and thus has less complex interfaces with a lower area footprint (due to the absence of frequency conversion logic). Since it operates at a higher frequency, its throughput performance is also better. However, it must be noted that we are not concerned with absolute throughput values—instead we are interested in satisfying throughput constraint values, while minimizing the number of buses in the matrix, which both the approaches manage to do. The fixed 400-MHz approach has a drawback in the form of a larger power dissipation in its bus lines and bus logic components, due to its high frequency of operation. Thus, a designer needs to be aware of the tradeoff between using fixed frequency for the entire bus matrix and a variable frequency for the buses in the matrix when using BMSYN to synthesize a matrix. If simplicity of design is preferred, BMSYN can be made to synthesize a bus matrix for a single fixed frequency value; otherwise, if the overhead of frequency converters can be tolerated, a variable frequency can be used. When using a single fixed frequency in the matrix, choosing lower values can result in infeasible or high congestion matrix solutions; while choosing a higher value can result in higher power consumption.

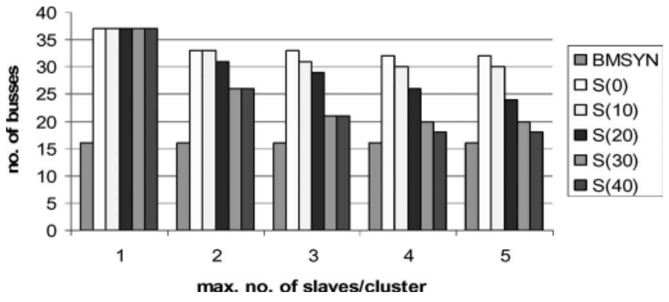

Fig. 14. Comparison with threshold-based approach for SIRIUS.

TABLE VII
SYNTHESIS TIME FOR MPSoC APPLICATIONS

Applications	Simulation Runs	Total Synthesis Time (in hours)
VIPER	11	3.6
SIRIUS	16	8
ORION4	23	14.5
HNET8	32	35.2

To compare the quality of the synthesis results using BMSYN, 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: 1) the maximum number of slaves per cluster and 2) the traffic overlap threshold, which if exceeded prevents two slaves from being assigned to the same bus cluster. The results of our comparison study are shown in Fig. 14. BMSYN is our approach while the other comparison points are obtained from [8]. $S(x)$, for $x = 0,\ 10,\ 20,\ 30,\ 40$, 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. 14 varies the maximum number of slaves allowed in a bus cluster for these comparison points. Note that the values of x are chosen based on the recommendations from [8]. It is clear from Fig. 14 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.

Table VII presents the total number of simulation runs and time taken for synthesis for each of the four MPSoC applications, while Fig. 15 compares the number of busses in a full bus matrix, a maximally connected reduced matrix and the final synthesized bus matrix using our approach, for these applications. It can be seen that our BMSYN approach takes in the order of a few hours to complete, instead of the several days or even weeks it would take for a manual effort. The branch and bound clustering algorithm and the static optimization phases take a negligible amount of time compared to the simulation time, for verification and design minimization in the last step in Fig. 4. Our BMSYN approach results in significant matrix component savings, ranging from 2.1× to 3.2× when compared to a maximally connected bus matrix, and savings ranging from 4.6× to 9× when compared with a full bus matrix.

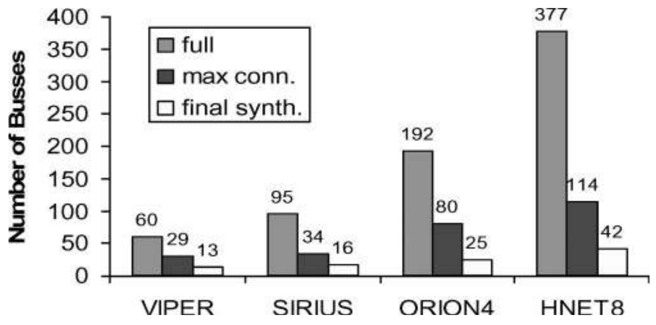

Fig. 15. Comparison of number of busses for MPSoC applications.

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 (requiring repeaters, thus limiting performance), and serious routability issues for multiple wires crossing the chip in a nonregular manner. NoC-based communication architectures, with a regular wire layout and having all links of the same length, offer a predictable model for wire crosstalk and delay. This predictability will permit aggressive clock rates and thus be able to support much larger throughput constraints. Therefore, we believe that for very large MPSoC systems in the future having several hundreds of cores and with terabits per second data throughput constraints, 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 (BMSYN) 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. Future work will deal with extending our BMSYN framework to include the effect of power consumption and area overhead, so that the designer can trade-off power, performance, and area characteristics of the synthesized bus matrix architecture.

References

[1] *ARM AMBA Specification and Multi Layer AHB Specification*, 2001, (rev2.0). [Online]. Available: http://www.arm.com
[2] *IBM On-Chip CoreConnect Bus Architecture Specification*, 2001, (rev2.1). [Online]. Available: http://www.chips.ibm.com/products/coreconnect/index.html

[3] "STBus communication system: Concepts and definitions," *Reference Guide*, STMicroelectronics, Geneva, Switzerland, May 2003, pp. 1–111.
[4] *Sonics Integration Architecture*, Sonics Inc., 2006, (rev1.0). [Online]. Available: http://www.sonicsinc.com
[5] L. Benini and G. D. Micheli, "Networks on chips: A new SoC paradigm," *Computers*, vol. 35, no. 1, pp. 70–78, Jan. 2002.
[6] J. Henkel, W. Wolf, and S. Chakradhar, "On-chip networks: A scalable, communication-centric embedded system design paradigm," in *Proc. VLSI Des.*, 2004, pp. 845–851.
[7] M. Nakajima *et al.*, "A 400 MHz 32b embedded microprocessor core AM34-1 with 4.0 GB/s cross-bar bus switch for SoC," in *Proc. ISSCC*, 2002, pp. 274–504.
[8] S. Murali and G. De Micheli, "An application-specific design methodology for STbus crossbar generation," in *Proc. DATE*, 2005, pp. 1176–1181.
[9] V. Lahtinen, E. Salminen, K. Kuusilinna, and T. Hamalainen, "Comparison of synthesized bus and crossbar interconnection architectures," in *Proc. ISCAS*, 2003, pp. 433–436.
[10] K. K. Ryu, E. Shin, and V. J. Mooney, "A comparison of five different multiprocessor SoC bus architectures," in *Proc. DSS*, 2001, pp. 202–209.
[11] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon, "Analyzing on-chip communication in a MPSoC environment," in *Proc. DATE*, 2004, pp. 752–757.
[12] M. Gasteier and M. Glesner, "Bus-based communication synthesis on system level," in *Proc. ACM TODAES*, Jan. 1999, pp. 65–70.
[13] S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Fast exploration of bus-based on-chip communication architectures," in *Proc. CODES+ISSS*, 2004, pp. 242–247.
[14] S. Pasricha, N. Dutt, E. Bozorgzadeh, and M. Ben-Romdhane, "Floorplan-aware automated synthesis of bus-based communication architectures," in *Proc. DAC*, 2005, pp. 565–570.
[15] K. Srinivasan, K. S. Chatha, and G. Konjevod, "Linear programming based techniques for synthesis of network-on-chip architectures," in *Proc. ICCD*, 2004, pp. 422–429.
[16] D. Bertozzi *et al.*, "NoC synthesis flow for customized domain specific multiprocessor systems-on-chip," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 2, pp. 113–129, Feb. 2005.
[17] O. Ogawa *et al.*, "A practical approach for bus architecture optimization at transaction level," in *Proc. DATE*, 2003, pp. 176–181.
[18] *SystemC Language Reference Manual*, May 2005, (ver2.1). [Online]. Available: http://www.systemc.org
[19] S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Extending the transaction level modeling approach for fast communication architecture exploration," in *Proc. DAC*, 2004, pp. 113–118.
[20] S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Automated throughput-driven synthesis of bus-based communication architectures," in *Proc. ASPDAC*, 2005, pp. 495–498.
[21] *ARM AMBA AXI Specification*, 2004, (ver:1.0). [Online]. Available: http://www.arm.com/armtech/AXI
[22] S. Pasricha, "Transaction level modeling of SoC with SystemC 2.0," in *Proc. SNUG*, 2002, pp. 55–59.
[23] K. Lahiri, A. Raghunathan, and S. Dey, "Efficient exploration of the SoC communication architecture design space," in *Proc. ICCAD*, 2000, pp. 424–430.
[24] U. Ogras and R. Marculescu, "Energy–and performance-driven NoC communication architecture synthesis using a decomposition approach," in *Proc. DATE*, 2005, pp. 352–357.
[25] A. Pinto, L. P. Carloni, and A. L. Sangiovanni-Vincentelli, "Efficient synthesis of networks on chip," in *Proc. ICCD*, 2003, pp. 146–150.
[26] A. Jalabert, S. Murali, L. Benini, and G. De Micheli, "XpipesCompiler: A tool for instantiating application specific networks on chip," in *Proc. DATE*, 2004, pp. 884–889.
[27] A. Pinto, L. Carloni, and A. Sangiovanni-Vincentelli, "Constraint-driven communication synthesis," in *Proc. DAC*, 2002, pp. 783–788.
[28] K. K. Ryu and V. J. Mooney, III, "Automated bus generation for multiprocessor SoC design," in *Proc. DATE*, 2003, pp. 202–209.
[29] S. Pandey and M. Glesner, "Statistical on-chip communication bus synthesis and voltage scaling under timing yield constraint," in *Proc. DAC*, 2006, pp. 663–668.
[30] D. Lyonnard, S. Yoo, A. Baghdadi, and A. A. Jerraya, "Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip," in *Proc. DAC*, 2001, pp. 518–523.
[31] S. Pasricha, N. Dutt, E. Bozorgzadeh, and M. Ben-Romdhane, "FABSYN: Floorplan-aware bus architecture synthesis," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 14, no. 3, pp. 241–253, Mar. 2006.
[32] W. J. Bainbridge and S. B. Furber, "CHAIN: A delay insensitive CHip area interconnect," *IEEE Micro—Special Issue Design Test System Chip*, vol. 22, no. 5, pp. 16–23, Sep./Oct. 2002.
[33] K. Goossens, J. Dielissen, and A. Radulescu, "The Aethereal network on chip: Concepts, architectures, and implementations," *IEEE Des. Test Comput.*, vol. 22, no. 5, pp. 21–31, Sep./Oct. 2005.

Sudeep Pasricha (S'02) received the B.E. degree in electronics and communication engineering from Delhi Institute of Technology, Delhi, India, in 2000 and the M.S. degree in computer science from the University of California, Irvine, in 2005, where he is currently working toward the Ph.D. degree.

His general area of research interest is design automation and synthesis of embedded systems, and more specifically multiprocessor system on chips, with a particular focus on on-chip communication architecture design. His other interests include system-level modeling languages and design methodologies, computer architecture, very large scale integration (VLSI) computer-aided design (CAD) algorithms and middleware for distributed embedded systems. He has a filed for a U.S. patent, presented tutorials on the topic of on-chip communication architectures at ASPDAC 2006 and VLSID 2007, and coauthored over 20 technical papers.

Mr. Pasricha received the Best Paper Award at ASPDAC 2006. He is a member of Association for Computing Machinery (ACM).

Nikil D. Dutt (S'81–M'89–SM'97) received the Ph.D. degree in computer science from the University of Illinois, Urbana–Champaign, in 1989.

He is currently a Chancellor's Professor of electrical engineering and computer science with the University of California, Irvine, and is affiliated with the following centers at UCI: CECS, CPCC, and CAL-IT2. His research interests are in embedded systems design automation, computer architecture, optimizing compilers, system specification techniques, and distributed embedded systems.

Dr. Dutt currently serves as Editor-in-Chief of *ACM Transactions on Design Automation of Electronic Systems* and as Associate Editor of *ACM Transactions on Embedded Computer Systems*. He was an ACM SIGDA Distinguished Lecturer during 2001–2002, and an IEEE Computer Society Distinguished Visitor for 2003–2005. He has served on the steering, organizing, and program committees of several premier CAD and Embedded System Design conferences and workshops, including ASPDAC, CASES, CODES+ISSS, DATE, ICCAD, ISLPED, and LCTES. He serves or has served on the advisory boards of ACM SIGBED and ACM SIGDA and is Vice-Chair of IFIP WG 10.5. He received best paper awards at CHDL89, CHDL91, VLSIDesign2003, CODES+ISSS 2003, and ASPDAC-2006.

Mohamed Ben-Romdhane (M'90) received the B.S. and M.S. degrees in electrical engineering, both from Ecole National des Ingenieurs de Tunis, Tunisia, in 1987 and 1989, respectively, and the Ph.D. degree in digital signal processing from Georgia Institute of Technology, Atlanta, in 1995.

He is currently the Vice President of baseband engineering for Newport Media, Inc., Lakeforest, CA, focusing on designing chips for mobile audio and video standards such as DVB-H, ISDB-T, T-DMB, and DAB. Previously, he served as Executive Director of SOC, IP, and Software for Conexant Systems, Inc. His research interests include wireless systems, low-power design, embedded systems, DSP algorithms and implementation, and SOC design.