COSMECA: Application Specific Co-Synthesis of Memory and Communication Architectures for MPSoC

Sudeep Pasricha and Nikil Dutt

Center for Embedded Computer Systems
University of California, Irvine, CA 92697, USA
{sudeep, dutt}@cecs.uci.edu

Abstract

Memory and communication architectures have a significant impact on the cost, performance, and time-to-market of complex multi-processor system-on-chip (MPSoC) designs. The memory architecture dictates most of the data traffic flow in a design, which in turn influences the design of the communication architecture. Thus there is a need to co-synthesize the memory and communication architectures to avoid making sub-optimal design decisions. This is in contrast to traditional platform-based design approaches where memory and communication architectures are synthesized separately. In this paper, we propose an automated application specific co-synthesis methodology for memory and communication architectures (COSMECA) in MPSoC designs. The primary objective is to design a communication architecture having the least number of busses, which satisfies performance and memory area constraints, while the secondary objective is to reduce the memory area cost. Results of applying COSMECA to several industrial strength MPSoC applications from the networking domain indicate a saving of as much as 40% in number of busses and 29% in memory area compared to the traditional approach.

1 Motivation

Modern multi-processor system-on-chip (MPSoC) designs are rapidly increasing in complexity. These designs are characterized by large bandwidth requirements and massive data sets which must be stored and accessed from memories, especially for applications in the multimedia and networking domains. The communication architecture in such systems, which must cope with the entire inter-component traffic, not only impacts performance considerably, but also consumes a significant chunk of the design cycle [1-2]. Another major factor influencing performance is the memory architecture, which can occupy upto 70% of the die area [3]. Estimates indicate that this figure will go up to 90% in the coming years [4]. Since memory and communication architectures have such a significant impact on system cost, performance and time-to-market, it becomes imperative for designers to focus on their exploration and synthesis early in the design flow, with the help of efficient design flow concepts such as those proposed in platform-based design [6].

Traditionally, in platform-based design, memory synthesis is performed before the communication architecture synthesis step [7-11]. While treating these two steps separately is done mainly due to tractability issues [5][12], it can lead to sub-optimal design decisions. Consider the example of a networking MPSoC subsystem shown in Fig. 1(a). The figure shows the system after HW/SW partitioning, with all the IPs defined, including memory which is synthesized based on data size and high-level bandwidth constraint analysis. Fig. 1(b) shows the traditional approach where communication architecture synthesis is performed after memory synthesis, while Fig. 1(c) shows the case where memory and communication architectures have been co-synthesized. Now let us consider the implications of using a co-synthesis methodology. Firstly, the co-synthesis approach is able to detect that the data arrays stored in *Mem1* and *Mem2* end up sharing the same bus, and automatically merges and then maps the arrays onto a larger single physical memory from the library, thus saving area. Secondly, the co-synthesis approach is able to merge data arrays stored in *Mem3* and *Mem5* onto a single memory from the library, saving not only area but also eliminating two busses, as shown in Fig. 1(c). However, *Mem5* cannot share the same bus as *Mem3* (or *Mem4*) in Fig. 1(b) because the access times of the pre-synthesized physical memories are such that they cause traffic conflicts which violate bandwidth constraints. Thirdly, due to the knowledge of support for out-of-order (OO) transaction completion [14] by the communication architecture, the co-synthesis approach is able to add an OO buffer of depth 6 to *Mem4*, which enables it to reduce the number of ports from 2 to 1, thus saving area, while still meeting bandwidth constraints. It is thus apparent that the co-synthesis approach is able to make better decisions by taking the communication architecture into account while allocating/mapping data arrays to memory components, which reduces the cost of the system.

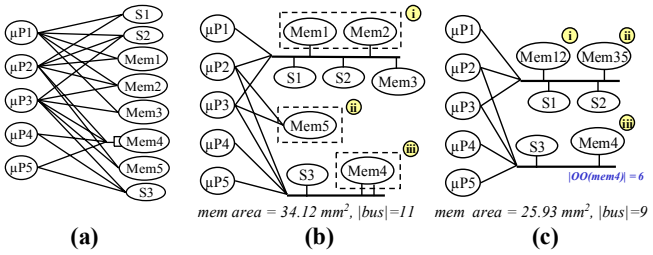


Fig. 1 (a) MPSoC system example with (b) memory synthesis before communication architecture synthesis, and (c) co-synthesis of memory and communication architectures

In this paper, we propose an automated application specific co-synthesis methodology for memory and communication architectures (*COSMECA*) in MPSoC designs. The primary objective is to design a communication architecture having the least number of busses, which satisfies performance and memory area constraints, while the secondary objective is to reduce the memory area cost. We consider a bus matrix (sometimes also called *crossbar switch*) [18] type of communication architecture for synthesis, since it is increasingly being used by designers in high bandwidth designs today. Our approach tailors the memory and communication architectures to the application being considered, to reduce system cost. Using a combination of an efficient *static branch and bound hierarchical clustering algorithm* and heuristics, we are able to quickly prune the uninteresting portion of the design space, while using fast transaction-based bus cycle-accurate SystemC [19] simulation models to capture dynamic system-level effects accurately and verify the results. *COSMECA* effectively synthesizes bus topology, arbitration schemes, bus speeds and OO buffer sizes for the communication architecture; and simultaneously performs data array allocation/mapping to memory blocks, deciding their number, sizes, ports and types from the memory library, for the memory subsystem. To the best of our

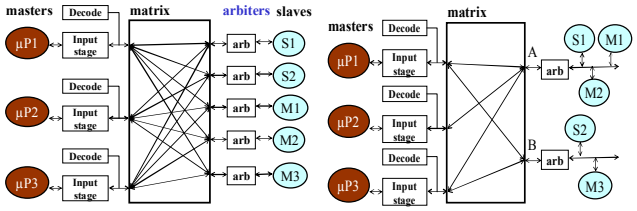
knowledge, no previous work has performed automated co-synthesis considering so many exploration parameters. Results of applying *COSMECA* to several industrial strength MPSoC networking applications indicate a saving of as much as 40% in number of busses and 29% in memory area, compared to the traditional approach of separate synthesis.

2 Related Work

Communication architectures have been the focus of much research over the past several years because of their significant impact on system performance [12][24][26]. Hierarchical shared bus communication architectures such as those proposed by AMBA [15], CoreConnect [16] and STbus [17] can cost effectively connect few tens of IPs, but are not scalable to cope with the demands of modern MPSoC systems. Network-on-Chip (NoC) based communication architectures [20] have recently emerged as a promising alternative to handle communication needs for the next generation of high performance designs, but research on the topic is still in its infancy, and few concrete implementations of complex NoCs exist to date [21]. Currently, designers are increasingly making use of *bus matrix* [18] communication architectures to meet the bandwidth requirements of modern MPSoC systems. The need for bus matrix architectures in high performance designs and its superiority over hierarchical shared busses has been emphasized in previous work [22-24]. Accordingly, we focus on the synthesis of bus matrix communication architectures.

Although a lot of work has been done in the area of hierarchical shared bus architecture synthesis (e.g. [25-26]) and NoC architecture synthesis (e.g. [27-28]), few efforts have focused on bus matrix synthesis. [29] proposed a transaction based simulation environment that allows designers to explore and design a bus matrix. But the designer needs to manually specify the communication topology, and arbitration scheme, which is too time consuming for today's complex systems. The automated synthesis approach for STBus crossbars proposed in [30] generates crossbar topology, but does not consider generation of parameters such as arbitration schemes, bus speeds and OO buffer sizes, which considerably impact system performance [12][26]. *COSMECA* overcomes these shortcomings by automating the synthesis of both topology and parameters for the bus matrix.

Previous research in the area of memory and communication architecture synthesis has either ignored the co-synthesis aspect, or focused on a small subset of the problem. Typically, high-level synthesis approaches perform memory allocation and mapping before communication architecture synthesis [7-11], ignoring the overhead of the communication protocol during synthesis. While treating these two steps separately is mainly due to tractability issues [5][12], the merits of integrating communication synthesis with memory synthesis are clearly demonstrated in [13]. Only a few approaches have attempted to simultaneously explore memory and communication subsystems. [31] presents a tool to automatically generate a full crossbar and a dynamic memory management unit (DMMU). [32] considers the connectivity topology early in the design flow in conjunction with memory exploration, for simple processor-memory systems. More recently, [33] deals with bus topology and static priority based arbitration exploration, to determine the best memory port-to-bus mapping for pre-synthesized memory blocks. Other approaches which deal with memory synthesis make use of static estimations of communication architectures such as those proposed in [34]. However, these techniques are unable to capture dynamic effects such as contention and capture only a limited exploration space. *COSMECA* improves upon previous approaches by *(i)* automatically generating bus topology and parameter values for arbitration schemes, bus speeds and OO buffer sizes, while considering dynamic simulation effects, and *(ii)* simultaneously determining a mapping of data arrays to physical memories while also deciding the number, size, ports and type of these memories, from a memory library.



(a) full bus matrix **(b) partial bus matrix**

Fig. 2 Bus matrix architecture

3 Bus Matrix Communication Architectures

This section describes bus matrix architectures. Fig. 2 (a) shows a three-master, five-slave full AMBA bus matrix. 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. *Decode* generates select signals for slaves. Unlike in traditional shared bus architectures, arbitration in a bus matrix is not centralized, but distributed so that every slave has its own arbitration. Also, typically, all busses within a bus matrix have the same data bus width, which usually depends on the application.

One drawback of the *full bus matrix* structure shown in Fig. 2(a) is that it connects every master to every slave in the system, resulting in a prohibitively large number of busses. The excessive wire congestion can make it practically impossible to route and achieve timing closure for the design [1-2]. Fig. 2(b) shows a *partial bus matrix* which has fewer busses and consequently uses fewer components (e.g. decoders, arbiters, buffers), has a smaller area and also utilizes less power. The basic idea here is to group slaves/memories on shared busses, as long as performance constraints are met. Points *A* and *B* in Fig. 2(b) are referred to as *slave access points* (SAPs). The communication architecture synthesis in *COSMECA* attempts to generate a partial bus matrix tailored to the target application, with a minimal number of busses in the matrix. Additionally, we generate arbitration schemes at the SAPs, bus clock speed values and OO buffer size values.

4 Memory Subsystem

There are a variety of different memory types available to satisfy memory requirements in applications. Typically, designers have used off-chip DRAMs for larger memory requirements and on-chip embedded SRAMs for smaller memory requirements. Lately, on-chip embedded DRAMs are gaining in popularity as they eliminate I/O signals to separate memory chips, boosting performance and reducing noise, as well as pin count, which ends up lowering system cost. Although SRAMs have smaller access times than DRAMs, they also take up a larger area, requiring a tradeoff between area and performance between the two memory types during synthesis. There is also a need for non-volatile memories such as EPROMs and EEPROMs to typically store read-only data in a system. The memory synthesis in *COSMECA* uses a memory library populated by on-chip SRAMs, on-chip DRAMs, EPROMs and EEPROMs having different capacities, areas, ports and access times. We assume that the word size of these memories is fixed, based on the application. Data arrays and groups of scalars in the application are grouped together into *virtual memories* (VMs) based on certain rules, before being mapped onto the appropriate physical memories from the library, which allow the application to meet its area and performance constraints. This grouping of data blocks allows us to reduce the number of memories in the design, thus reducing area. We also try to avoid multi-port memories because of their excessive area and cost overhead.

5 COSMECA Co-Synthesis Methodology

5.1 Assumptions and Problem Definition

We are given an application for which we assume the HW/SW partitioning has already been performed. The resulting MPSoC design has possibly several hardware and software IPs onto which application functionality has been mapped. Memory in this model is initially represented by abstract *data blocks* (DBs) which are collections of scalars or arrays accessed by the application, similar to *basic groups* in [10]. Generally, this MPSoC design will have performance constraints, dependent on the application. The *throughput* of communication between components is a good measure of the performance of a system [25]. To represent performance constraints in *COSMECA*, we define a **Communication Throughput Graph** *CTG = G(V,A)* [2] which is a directed graph, where each vertex *v* represents an IP (or DB) in the system, and an edge *a* connects components that need to communicate with each other. A **Throughput Constraint Path** (TCP) is a sub-graph of a CTG, consisting of a single master for which data throughput must be maintained and other masters, slaves and DBs which are in the critical path that impacts the maintenance of the throughput.

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 our primary objective is to determine an optimal component to bus assignment for a bus matrix architecture, such that the partitioning of* $\boldsymbol{V}$ *onto* $\boldsymbol{N}$ *busses results in a minimal number of busses* $|\boldsymbol{N}|$ *and satisfies memory area bounds while meeting all constraints in the design, represented by the TCPs in a CTG. As a secondary objective, we attempt to reduce memory area cost of the solution.*

5.2 Simulation Engine

Since communication behavior in a system is characterized by unpredictability due to dynamic bus requests from IPs, contention for shared resources, buffer overflows etc., a simulation engine is necessary for accurate performance estimation. *COSMECA* uses a hybrid approach based on static estimation as well as dynamic simulation. For the dynamic simulation part, we capture behavioral models of IPs and bus architectures in SystemC [19][26], and keep them in an IP library database. Since simulation speed is important, we chose a fast transaction-based, bus cycle accurate modeling abstraction, which averaged simulation speeds of 150–200 Kcycles/sec [26], while running embedded software applications on processor ISS models. The communication model in this abstraction is extremely detailed, capturing delays arising due to frequency and data width adapters, bridge overheads, interface buffering and all the static and dynamic delays associated with the standard bus architecture protocol being used.

5.3 Communication-Memory Constraint Set Ψ

In the interest of generating a practically realizable system, we allow a designer to specify a discrete set of valid values (referred to as a constraint set Ψ) for communication parameters such as bus clock speeds, OO buffer sizes and arbitration schemes. Additionally, Ψ allows the specification of constraints on the type of memory to allocate for DBs, for instance, in the case of a DB which the designer knows must be read from an EEPROM memory. 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 presence of a local constraint overrides the global constraint, while the absence of it results in the resource inheriting 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. This provides a convenient mechanism for the designer to bias the co-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 co-synthesis framework, but informed decisions can help avoid the synthesis of unrealistic system configurations.

5.4 *COSMECA* Flow

We describe the *COSMECA* flow in more detail in this section. Fig. 3 gives a high level overview of the flow. The inputs to *COSMECA* include a Communication Throughput Graph (*CTG*), a library of behavioral IP models (*IP library*) and memory models (*mem library*), a Data Block Dependency Graph (*DBDG*), a target bus matrix template (e.g. AMBA [15] bus matrix) and a communication-memory constraint set (Ψ) – which includes Ψ_G and Ψ_L. The general idea is to first preprocess the memory (represented by DBs in the CTG) in the design by merging the non conflicting DBs into *virtual memory* (VM) blocks to reduce memory cost. Then we map the modified CTG to a full bus matrix template and optimize the matrix by removing unused busses. Next, we perform a *static branch and bound hierarchical clustering* of slave components in the matrix which further reduces the number of busses, and store prospective matrix architecture solutions in a *ranked matrix solution database*. We then use a heuristic (*memmap*), which first merges VMs at each SAP to further reduce memory cost and then maps these VMs to physical memory modules from the memory library. The output of *memmap* is a set of *N* valid solutions which meet memory area and performance constraints. Finally we *optimize* the output solutions to reduce bus speeds, arbitration costs and fix OO buffer sizes. We now elaborate on the five phases in the COSMECA flow, shown in Fig. 3.

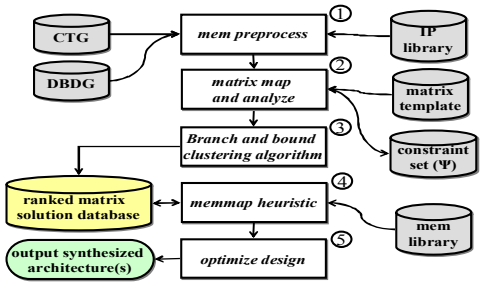


Fig. 3 COSMECA flow

Phase 1. mem preprocess: In the first phase, we merge DBs in the CTG into VMs to reduce memory area cost, by potentially reducing the number of memory modules in the system. Only DBs with *(i)* similar edges (i.e. edges from the same masters) and *(ii)* non-overlapping access are merged, so as not to constrain mapping freedom later in the flow. We use a Data Block Dependency Graph (DBDG) to determine if DBs have non-overlapping access. Fig. 4(b) shows the DBDG for the example in Fig. 4(a). The DBDG shows the dependency of DB accesses on each other – a DB cannot be accessed till the source DBs of all its input edges have been accessed. If two DBs have similar edges and non-overlapping access, they are eligible for merger (e.g. DB1, DB2 in Fig. 4(b)). The size of the VM created depends on the lifetime analysis of merged DBs – it is the sum of the sizes of the merged DBs, unless the lifetimes do not overlap, in which case it is the size of the larger DB being merged. Fig 4(b) shows the lifetime of DB1. It is possible for DB2 to overwrite DB1, thus saving memory space. More details can be found in our technical report [35].

Phase 2. matrix map and analyze: In the second phase, the modified CTG is mapped onto a full bus matrix, which is then pruned by removing unused busses. Dedicated slave and memory components are migrated to the local busses of their corresponding

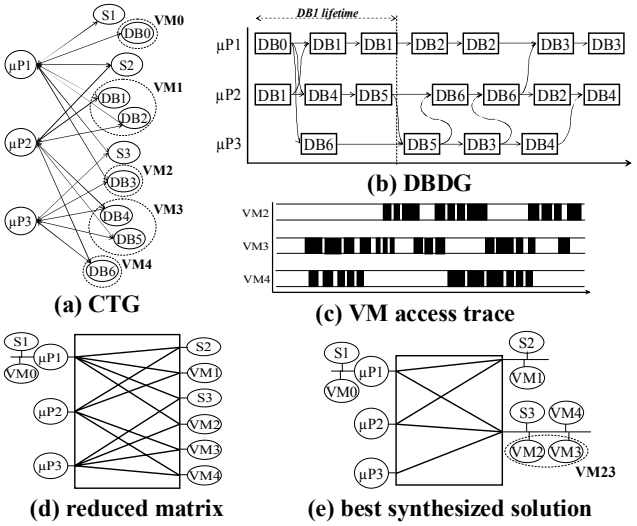


Fig. 4 *COSMECA* co-synthesis example

masters to further reduce busses in the matrix. Fig. 4(d) shows the bus matrix after these steps, for the example in Fig. 4(a). Finally, we perform Transaction Level (TLM) simulation [26] of the application, assuming no arbitration contention, to obtain application-specific data traffic statistics such as the number of transactions on a bus and average transaction burst size on a bus. Knowing the bandwidth to be maintained on a bus 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 ($\Gamma_{TLM/B}$) from the TLM simulation, for any bus B in the matrix is given by

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

where *numT* is the number of data transactions on bus B, *sizeT* is the average data transaction size, *width* is the bus width, Ω is the clock speed, and c is the total number of cycles of TLM simulation for the application. The values for *numT*, *sizeT* and c are obtained from the TLM simulation. To meet throughput constraint $\Gamma_{TCP/B}$ for bus B,

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

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

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

Phase 3. Branch and bound clustering algorithm: In the third phase, a *static branch and bound hierarchical clustering algorithm* is used to cluster slave/memory components to reduce the number of busses in the matrix even further. Note that we do not consider merging masters because it adds two levels of contention (one at the master end and another at the slave end) in a data path, which can drastically degrade system performance. Before describing the algorithm, we present a few definitions. A slave cluster $SC = \{s_1 \ldots s_n\}$ 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 $\Pi_{SC1/SC2}$ 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 $\beta \subset \Pi_{SC1/SC2}$, K_β refers to the set of allowed bus speeds for the newly created bus when the busses in set β are merged, and is given by

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

The branching algorithm starts 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, that

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

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

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

Fig. 5 *bound* function

is too time-consuming to traverse. In order to consider only valid clustering configurations, we make us of a bounding function.

Fig. 5 shows the pseudocode for the *bound* function which is called after every clustering operation of any two slave clusters *SC1* and *SC2*. In Step 1, we use a look up table to see if the clustering operation has already been considered previously, and if so, we discard the duplicate clustering. Otherwise we update the lookup table with the entry for the new clustering. In Step 2, we check to see if the clustering of *SC1* and *SC2* results in the merging of busses in the matrix, otherwise the clustering is not beneficial and the solution can be bounded. If the clustering results in bus mergers, we calculate the number of merged busses for the clustering and store the cumulative weight of the clustering operation in the branch solution node. In Step 3, we check to see if the allowed set of bus speeds for every merged bus is compatible or not. If the allowed speeds for any of the busses being merged are incompatible ($K_\beta == \phi$ 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 bus. If this is not the case, we bound the solution. The *bound* function thus enables a conservative pruning process which quickly eliminates invalid solutions and allows us to rapidly converge on the optimal solution. The solutions obtained from the algorithm are ranked from best (least number of busses) to worst and stored in a *ranked matrix solution database*. Fig. 4(e) shows the best solution after this phase, for the example in Fig. 4(a). For each of the solutions, we set OO buffer sizes to the maximum allowed in Ψ, for the components which support it. For the arbitration scheme at the SAPs, we use a *TDMA/RR* strategy to proportionally grant accesses to masters based on the magnitude of throughput requirements. Our previous work has shown the effectiveness of TDMA/RR for this purpose [26].

Phase 4. memmap heuristic: In the next phase, we use a heuristic (*memmap*) to guide the mapping of VMs to physical memories in the memory library. The goal is to find N solutions which satisfy memory area and performance constraints of the design. The general idea is to first simulate the best solution from the *ranked matrix solution database,* to generate memory access traces, which are used to determine the extent of access overlap of VMs at each SAP. If the overlap is below a user defined overlap threshold τ, we merge the VMs. Fig. 4(e) shows how we merge *VM2* and *VM3*, as their memory access trace shown in Fig. 4(c) has an overlap less than the chosen value for τ. We then map the VMs in the design to physical memories from the memory library. We initially choose the best memory from the library which fits the size requirement and has the maximum port bandwidth. If we find that the throughput constraints are not met even for the memory with best performance, we discard the matrix solution, and go back to select the next best matrix solution from the *ranked matrix solution database*. Otherwise, if throughput constraints are met, and memory area constraints are also met, we add the solution to the final solution database. We then attempt to lower memory area by randomly selecting a VM at every SAP and replacing the mapped physical memory with one which meets the size requirements, but has lower area. If there is no performance violation, and if the area bounds are met, we have found a solution. We keep

repeating this process till all VMs become ineligible for mapping optimization, or if the required *N* solutions have been found. If we encounter the former case and the number of solutions found is less than *N*, we proceed to select the next best solution from the *ranked matrix solution database* and repeat the process. A detailed description of the heuristic can be found in our technical report [35].

Phase 5. optimize design: Finally, we call the *optimize design* procedure for each of the *N* solutions obtained in the last phase. This simple procedure attempts to minimize (i) bus speeds, (ii) arbitration scheme implementation cost and (iii) fix OO buffer sizes. The procedure first iterates over the busses in a solution, reducing the bus speed to the lowest possible allowed, simulating the design to ensure that no performance constraints are violated. Similarly, the procedure attempts to iteratively replace the TDMA/RR arbitration scheme with a static priority based scheme (which has lower implementation cost) at each SAP, with priorities assigned depending on bandwidth requirements. Finally we fix the size of the OO buffer sizes wherever applicable to the maximum number of buffers used during simulation, if the number is less than the maximum allowed buffer size.

6 Case Studies

We applied the *COSMECA* approach to four industrial strength MPSoC applications – PYTHON, SIRIUS, VIPER2 and HNET8 – from the networking domain. PYTHON and SIRIUS are variants of existing industrial strength designs, VIPER2 and HNET8 are larger systems which have been derived from the next generation of MPSoC applications currently in development. Table 1 shows the number of components in each of these applications, after HW/SW partitioning. Note that the *Masters* column includes the processors in the design, while the *Slaves* column does not include the memory blocks, which will be co-synthesized with the communication architecture later.

Table 1. Core distribution in MPSoC applications

Applications	Processors	Masters	Slaves
PYTHON	2	3	8
SIRIUS	3	5	10
VIPER2	5	7	14
HNET8	8	13	17

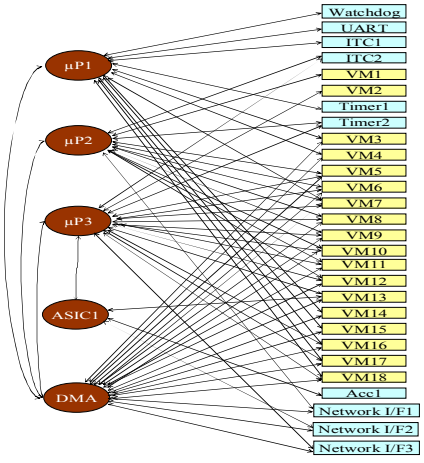


Fig. 6 SIRIUS Communication Throughput Graph (CTG)

Fig. 6 shows the CTG for the SIRIUS application, after the initial memory preprocessing phase in which DBs are merged into VMs. Not shown in the CTG, but included in our memory area analysis are the 32 KB instruction and data caches for each of the three processors. For clarity, the TCPs are presented separately in Table 2. µP1 is a protocol processor (PP) while µP2 and µP3 are network processors (NP). The µP1 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 µP2 and µP3 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. ASIC1 performs hardware cryptography acceleration for DES, 3DES and AES. The DMA is used to handle fast memory to memory and network interface data transfers, freeing up the processors for more useful work. SIRIUS also has a number of network interfaces and peripherals such as interrupt controllers (ITC1, ITC2), a UART, timers (Watchdog, Timer1, Timer2) and a packet accelerator (Acc1).

Table 2. SIRIUS Throughput Constraint Paths (TCPs)

IP cores in Throughput Constraint Path (TCP)	TCP constraint
µP1, VM3, VM4, DMA, VM16, VM17, VM18	640 Mbps
µP1, VM5, VM6, VM14, VM15, DMA, Network I/F2	480 Mbps
µP2, Network I/F1, VM8, VM9	5.2 Gbps
µP2, VM10,VM11,VM12, DMA, Network I/F3	1.4 Gbps
ASIC1, µP3, VM16, VM17, VM18, Acc1, VM13, Network I/F2	240 Mbps
µP3, DMA , Network I/F3, VM13	2.8 Gbps

Table 3. SIRIUS Global Constraint Set Ψ_G

Set	Values
bus speed	25, 50, 100, 200, 300, 400
arbitration strategy	static, RR, TDMA/RR
OO buffer size	1 – 8
mem mapping	VM16,VM17=>*DRAM*; VM1,VM2=>*EEPROM*

Table 3 shows the global constraint set Ψ_G for SIRIUS. For the synthesis we target an AMBA3 AXI [14] bus matrix. We assume a fixed bus width of 32 bits, as per application requirements. The memory area constraint is set to 225 mm^2 and the estimated memory area numbers are for a 0.18-µm technology. We assume the value for overlap threshold τ = 10% for this example. Fig. 7 shows the best solution (least number of busses) with the least memory area for SIRIUS. The figure also shows bus speeds, memory sizes, number of ports and OO buffer sizes.

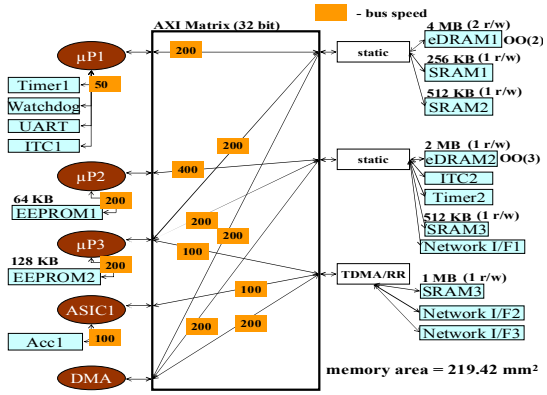


Fig. 7 Synthesized output for SIRIUS

Fig. 8 shows the variation in memory area and number of busses for the ten best solutions (*N=10*). *COSMECA* allows a designer to tradeoff memory area with bus count in the final solution. The dotted line indicates the solution shown in Fig. 7. It can be seen that the memory area cost varies dramatically, not only when the bus matrix configuration is changed (by changing number of busses), but also for the same configuration, for different memory mapping decisions. The entire *COSMECA* flow took only a few hours to complete, including simulation time. This is in contrast to the traditional semi-automated (or manual) communication architecture synthesis techniques which can take several days [2], and would take even longer with the added

complexity of handling memory synthesis. Due to lack of space, we omit details of the co-synthesis effort for the other three applications. More information on co-synthesis for these applications as well as an analysis of the effect of the overlap factor τ on solution quality can be found in our technical report [35].

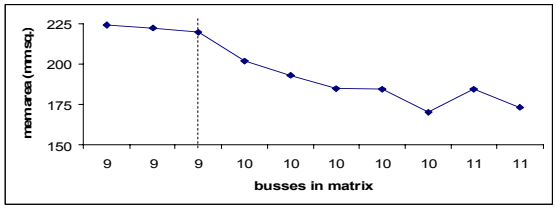


Fig. 8 SIRIUS final solution set (for N=10)

Finally, Fig. 9 and 10 compare the number of busses and memory areas for the best solution (having least number of busses, minimum memory area for the solution) obtained with *COSMECA* and the traditional approach (where memory synthesis is done before communication architecture synthesis) for the four applications. It can be seen that *COSMECA* performs much better for each of the applications, saving from 25-40% in the number of busses in the matrix and from 17-29% in memory area, because it is able to make better decisions by taking the communication architecture into account while allocating and mapping data blocks to physical memory components.

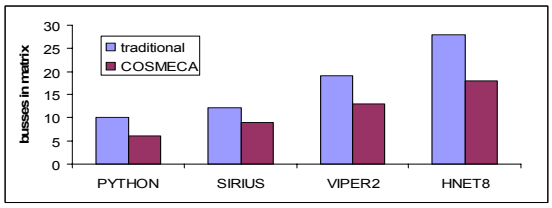


Fig. 9 Comparison of best solution bus count

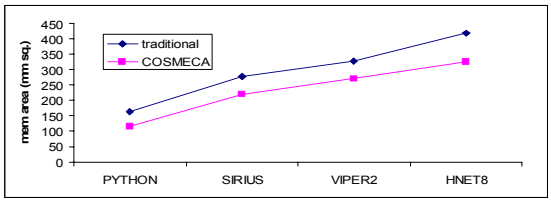


Fig. 10 Comparison of best solution memory area

7 Conclusion and Future Work

In this paper, we have presented an automated application specific methodology to co-synthesize memory and communication architectures (*COSMECA*) in MPSoC designs. The primary objective is to design a communication architecture having the least number of busses, which satisfies performance and memory area constraints, while the secondary objective is to reduce the memory area cost. *COSMECA* couples the decision making process during memory and communication architecture synthesis, which enables it to generate a lower cost system. Results of applying *COSMECA* to several industrial strength MPSoC applications from the networking domain indicate a saving of as much as 40% in number of busses and 29% in memory area compared to the traditional approach, where memory synthesis is performed before communication architecture synthesis. Our ongoing work is trying to integrate more detailed memory access protocol models for the memories in the library. Future work will deal with incorporating power as another metric to guide the co-synthesis and including cache customization in the memory synthesis process.

Acknowledgements

This research was partially supported by grants from SRC (2005-HJ-1330) and a CPCC fellowship.

References

[1] D. Sylvester, K. Keutzer, "Getting to the bottom of deep submicron", *ICCAD 1998*

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

[3] S. Meftali et al, "An optimal memory allocation for application-specific multiprocessor system-on-chip", *ISSS 2001*

[4] A. Allan et al, "2001 Technology Roadmap for Semiconductors", *IEEE Computer, Vol. 35, No. 1, 2002*

[5] J. A. Rowson et al., "Interface based design" *DAC 1997*

[6] K. Keutzer et al. "System-level design: Orthogonalization of concerns and platform-based design," *IEEE TCAD, Dec. 2000*

[7] I.-M. Daveau, et al. "Synthesis of System-Level Communication by an Allocation-Based Aporoach", *ISSS, 1995*

[8] S. Narayan, D. Gajski, "Protocol generation for communication channels" *DAC 1994*

[9] I. Madsen, B. Hald, "An Approach to Interface Synthesis", *ISSS, 1995*

[10] S. Wuytack et al. "Minimizing the required memory bandwidth in VLSI system realizations", *IEEE TVLSI Vol 7, Issue 4, Dec. 1999*

[11] L. Cai, H. Yu, D. Gajski, "A novel memory size model for variable-mapping in system level design", *ASP-DAC 2004*

[12] K. Lahiri, et al, "System-level performance analysis for designing system-on-chip communication architecture", *IEEE TCAD Jun, 2001*

[13] P. Knudsen, J. Madsen, "Integrating communication protocol selection with partitioning in hardware/software codesign," *ISSS, 1998*

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

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

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

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

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

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

[20] L.Benini, G.D.Micheli, "Networks on Chips: A New SoC Paradigm", *IEEE Computers, Jan. 2002*

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

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

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

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

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

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

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

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

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

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

[31] M. Shalan, et al, "DX-Gt: Memory Management and Crossbar Switch Generator for Multiprocessor System-on-a-Chip" *SASIMI, 2003*

[32] P. Grun, et al, "Memory system connectivity exploration", *DATE 2002*

[33] S. Kim, C. Im, S. Ha, "Efficient Exploration of On-Chip Bus Architectures and Memory Allocation", *CODES+ISSS, 2004*

[34] P. V. Knudsen and J. Madsen, "Communication estimation for hardware/software codesign", *CODES 1998*

[35] S. Pasricha, N. Dutt, "A Framework for Memory and Communication Architecture Co-Synthesis in MPSoCs", *CECS Tech Report, Feb 2006*