

# **A Framework for Memory and Communication Architecture Co-synthesis in MPSoCs**

Sudeep Pasricha and Nikil Dutt

Center for Embedded Computer Systems  
University of California Irvine  
Irvine, CA 92697-3425, USA  
1 (949) 824-2248  
{sudeep, dutt}@cecs.uci.edu

CECS Technical Report #06-03  
February, 2006

# A Framework for Memory and Communication Architecture Co-synthesis in MPSoCs

Sudeep Pasricha and Nikil Dutt

Center for Embedded Computer Systems  
University of California Irvine  
Irvine, CA 92697-3425, USA  
1 (949) 824-2248  
{sudeep, dutt}@cecs.uci.edu

CECS Technical Report #06-03  
February, 2006

## 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 technical report, we propose an automated application specific co-synthesis framework 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.*

# A Framework for Memory and Communication Architecture Co-synthesis in MPSoCs

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 technical report, we propose an automated application specific co-synthesis framework 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

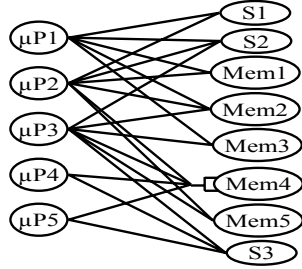
Modern multi-processor system-on-chip (MPSoC) designs are rapidly increasing in complexity. These designs are characterized by large bandwidth requirements and massive data sets which must be stored and accessed from memories, especially for applications in the multimedia and networking domains. The 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].

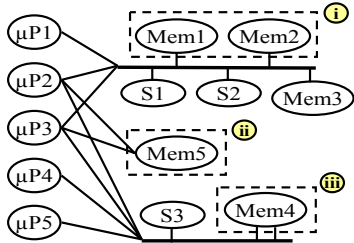
Traditionally, in platform-based design, memory synthesis is performed before the communication architecture synthesis step [7-11]. While treating these two steps separately is done mainly due to tractability issues [5][12], it can lead to sub-optimal design decisions. Consider the example of a networking 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 using the COSMECA approach. Now let us consider the implications of using a co-synthesis framework. 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 COSMECA co-synthesis approach is able to make better synthesis decisions by exploiting the synergy and interdependence between the memory and communication architecture design spaces, to reduce the overall cost of the synthesized system.

In this technical report, we propose an automated application specific co-synthesis framework 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.

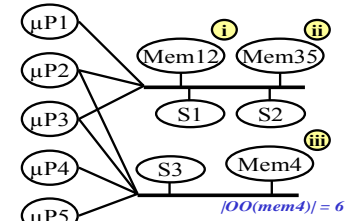


(a) MPSoC system



memory area = 34.12 mm<sup>2</sup>, |bus|=11

(b) Result of performing memory synthesis before communication architecture synthesis



memory area = 25.93 mm<sup>2</sup>, |bus|=9

(c) Result of performing co-synthesis of memory and communication architectures

**Fig. 1 Comparison of traditional approach (separate memory and communication architecture synthesis) and co-synthesis approaches for MPSoC example**

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]. 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. [2][25-26][36-40]) and NoC architecture synthesis (e.g. [27-28][41-43]), 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 have considerable impact on system performance [12][26][44]. *COSMECA* overcomes these shortcomings by automatically synthesizing both topology and communication 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 [34-35]. Such approaches are unable to capture dynamic effects such as contention and address only a limited exploration space. More importantly, none of the abovementioned approaches attempts to perform co-synthesis. *COSMECA* is a novel memory and communication architecture co-synthesis framework which improves upon existing synthesis 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. Results of applying the *COSMECA* approach to several industrial strength case studies (presented in Section 6) emphasizes the usefulness and need of such an approach for MPSoC designs.

### 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.

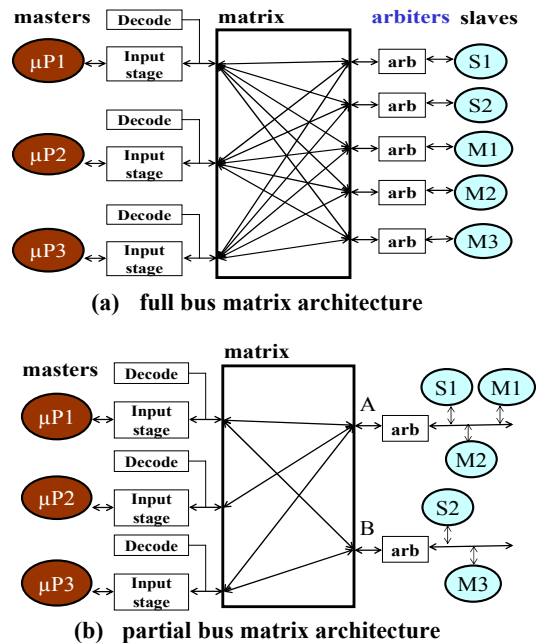


Fig. 2 Bus Matrix Communication Architecture

### 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-ported memories because of their excessive area and cost overhead.

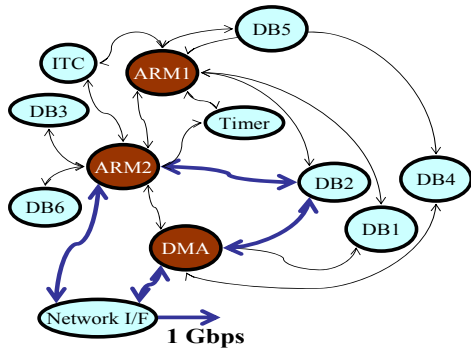


Fig. 3 Communication Throughput Graph (CTG)

## 5 COSMECA Co-Synthesis Framework

This section describes the *COSMECA* co-synthesis framework. First we state our assumptions and present the problem definition. Next, we describe our simulation engine and elaborate on the communication-memory constraint set, which guides the co-synthesis process. Finally, we describe the *COSMECA* co-synthesis flow in detail.

### 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 component 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.

Fig. 3 shows a CTG for a network subsystem, with a TCP involving the ARM2, DB2, DMA and ‘Network I/F’ components, where the rate of data packets streaming out of the ‘Network I/F’ component must not fall below 1 Gbps.

**Problem Definition:** A bus  $B$  can be considered to be a partition of the set of components  $V$  in a CTG, where  $B \subset V$ . Then our primary objective is to determine an optimal component to bus assignment for a bus matrix architecture, such that the partitioning of  $V$  onto  $N$  busses results in a minimal number of busses  $N$  and satisfies memory area bounds while meeting all performance 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][45], and keep them in an IP library database. SystemC provides a rich set of primitives for modeling concurrency, timing and synchronization - channels, ports, interfaces, events, clocks, signals and wait-state insertion. Concurrent execution is performed by multiple threads and processes (lightweight threads) and execution schedule is governed by the scheduler. SystemC also supports capture of a wide range of modeling abstractions from high level specifications to pin and timing accurate system models. Since it is a library based on C++, it is object oriented, modular and allows data encapsulation – all of which are essential for easing IP distribution, reuse and adaptability across different modeling abstraction levels.

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][44], 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 $\Psi$

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  $\Psi$ ) for communication parameters such as bus clock speeds, OO buffer sizes and arbitration schemes. Additionally,  $\Psi$  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 ( $\Psi_G$ ) and a local constraint set ( $\Psi_L$ ). 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 250 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 Co-Synthesis Flow

We describe the *COSMECA* co-synthesis flow in more detail in this section. Fig. 4 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 ( $\Psi$ ) – which includes  $\Psi_G$  and  $\Psi_L$ . 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 slave access point (*SAP*) in the bus matrix 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

prune out-of-order (OO) buffer sizes. We now elaborate on the five phases in the *COSMECA* flow, shown in Fig. 4.

**Phase 1. mem preprocess:** In the first phase, we merge data blocks (DBs) in the CTG into virtual memories (VMs) to reduce memory area cost, by potentially reducing the number of memory modules in the system. Only DBs satisfying the two criteria of having (i) similar edges (i.e. edges from the same masters) and (ii) non-overlapping access are merged, so as not to constrain the mapping freedom and eliminate useful channel clustering possibilities later in the flow. Fig. 5(a) shows a CTG for an example MPSoC system, with the following groups of DBs having similar edges: (DB1, DB2) and (DB4, DB5, DB6). We use a Data Block Dependency Graph (DBDG) to determine if DBs have non-overlapping access. The DBDG is a directed graph which shows the dependency of DB accesses on each other. It can either be created manually or derived automatically from a Control Data Flow Graph (CDFG). A node in a DBDG represents a DB access while an edge represents a dependency between DBs – a DB cannot be accessed till the source DBs of all its input edges have been accessed. Fig. 5(b) shows the DBDG for the example in Fig. 5(a). If two DBs have similar edges and non-overlapping access, they are eligible for merger (e.g. DB1, DB2 in Fig. 5(b)). The size of the VM created, after merger, 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 5(b) shows the lifetime of DB1. It is possible for DB2 to overwrite DB1, thus saving memory space.

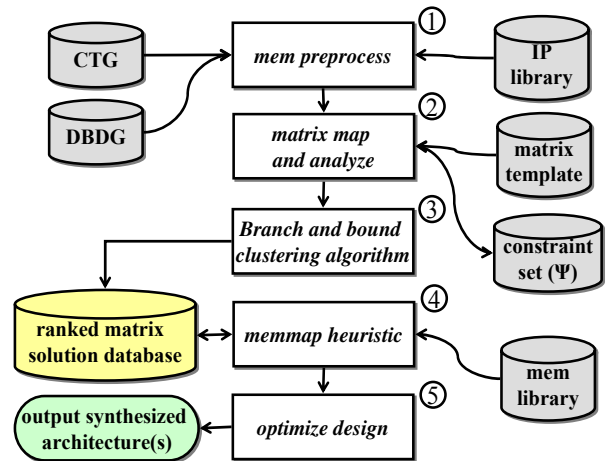


Fig. 4 COSMECA co-synthesis flow

**Phase 2. matrix map and analyze:** In the second phase, the modified CTG is mapped onto a full bus matrix template. The full bus matrix is subsequently pruned by removing unused busses on which there are no data transfers. Dedicated slave and memory components are

also migrated to the local busses of their corresponding masters to further reduce busses in the matrix. Fig. 5(d) shows the bus matrix after these steps, for the example in Fig. 5(a). Finally, we perform a fast high level, Transaction Level (TLM) simulation [26] of the application, using communication protocol-independent channels for communication and 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

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

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

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

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

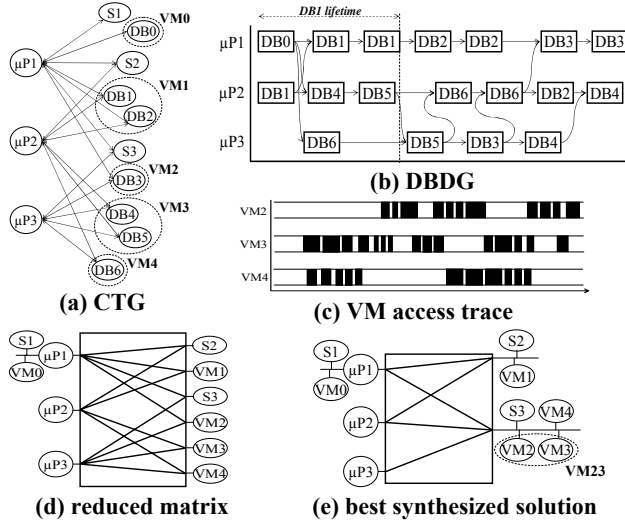


Fig. 5 COSMECA co-synthesis example

**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  $SC = \{s_1 \dots s_n\}$  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*  $\beta = \{b_1 \dots b_n\}$ , where  $\beta \subset \Pi_{SC1/SC2}$ ,  $K_\beta$  refers to the set of allowed bus speeds for the newly created bus when the busses in set  $\beta$  are merged, and is given by

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

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

```

Step 1: if (exists lookupTable(SC1,SC2)) then discard duplicate clustering
        else updatelookupTable(SC1, SC2)
Step 2: if (MSC1 ∩ MSC2 == ∅) then bound clustering
        else cum_weight = cum_weight + |MSC1 ∩ MSC2|
Step 3: for each set β ∈ ΠSC1/SC2 do
        if ((Kβ == ∅) || (∑i=1|β| ΓTCPi > (widthB × max_speedB))) then
            bound clustering

```

Fig. 6 bound function

Fig. 6 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 ( $K_\beta == \emptyset$  for any  $\beta$ ), 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. 5(e) shows the best solution after this phase, for the example in Fig. 5(a). For each of the solutions, we set OO buffer sizes to the maximum allowed in  $\Psi$ , for the components which support it. For the arbitration scheme at the *SAPs*, we initially use a possible more expensive-to-implement arbitration strategy such as the *TDMA/RR* scheme 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]. More details on the branch and bound clustering algorithm can be found in [46].

```

1: procedure memmap()
2:   while (num_sol < N) do
3:     select next candidate from ranked matrix solution database
4:     simulate design; //to generate memory trace
5:     for each SAP do
6:       merge VMs with overlap  $\leq \tau$ 
7:     for each VM do
8:       if (VM data overlap  $\leq \tau$ )
9:         map to single port physical mem with best size match, max. port b/w
10:      else
11:        map to dual port physical memory with best size match, max. port b/w
12:      simulate design; //to verify mem area, performance constraint satisfaction
13:      if (performance constraint violation) then
14:        remove candidate from ranked matrix solution database; goto 3
15:      else if ((perf. constraint satisfied)&&(mem area constraint satisfied)) then
16:        add to final solution database; num_sol++
17:        area_improvement_possible = true
18:      while ((num_sol < N) && (area_improvement_possible)) do
19:        for each SAP do
20:          randomly select eligible VM
21:          map physical memory with best size, port match, lower area
22:          simulate design; //to verify area, performance constraint satisfaction
23:          if ((perf constraint satisfied)&&(mem area constraint satisfied)) then
24:            add to final solution database; num_sol++
25:          else
26:            undo mapping for VM with port bandwidth violation
27:            make VM with violation ineligible for further selection
28:          if (all VMs ineligible) then
29:            area_improvement_possible = false
30:    end memmap

```

Fig. 7 *memmap* heuristic

**Phase 4. memmap heuristic:** In the next phase, we use the *memmap* heuristic to guide the mapping of VMs to physical memories in the memory library. Fig. 7 shows the pseudo code for the *memmap* heuristic. The goal is to find  $N$  solutions which satisfy memory area and performance constraints of the design. We begin by selecting the best solution from the *ranked matrix solution database*, populated in the previous phase, and simulate the design (lines 3-4), with the simulation engine described in Section 5.2. The output of this simulation is a set of 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  $\tau$ , we merge the VMs (lines 5-6). Fig. 5(e) shows how we merge *VM2* and *VM3*, as their memory access trace shown in Fig. 5(c) has an overlap less than the chosen value for  $\tau$ . The size of the merged VM is the sum of the memory sizes, unless the lifetimes do not overlap, in which case it is the size of the larger of the two VMs being merged. This VM merge step further reduces the number of memories, and consequently memory area cost.

Next we proceed to map the VMs in the design to physical memories from the memory library (lines 8-11). We choose the best memory from the library which fits the size requirement and has the maximum port bandwidth (i.e. combination of access time and operating frequency, which determines performance, expressed in terms of port bandwidth). The mapping step takes into consideration any memory mapping constraints in  $\Psi$ . It is possible that a VM has self conflict greater than  $\tau$ , in which case we map a dual port memory if possible, otherwise we use single port memories. The type of port (R,W,R/W) is determined by the maximum simultaneous reads/writes from the memory trace. The reason for using physical memories with the best performance is that we want to check the feasibility of the matrix solution being considered, and eliminate a solution quickly if it is not a good match. Once the mapping is complete, we simulate the design. If throughput constraints are not met even for the memory mapping with best performance, we discard the matrix solution, and go back to select the next best matrix solution from the *ranked matrix solution database*. If performance constraints are met, we check if memory area constraints are met. If the area constraint is also met, we add the solution to the final solution database (lines 12-16). Next, we attempt to lower memory area, while still meeting performance constraints, by changing the memory mapping for the current matrix solution (lines 17-29). We do this by selecting one eligible VM at each SAP randomly and replacing the mapped physical memory with one which meets the size (capacity) requirements, but has lower area. All VMs are initially eligible for this mapping optimization. Next we simulate the design. If we find a performance violation at one or more SAPs, we undo the change in mapping for the VM at each violated SAP, and make it ineligible for further mapping optimization. The reason for selecting just one VM per SAP is that it makes it easier to determine which physical memory to VM mapping caused a performance violation, if one is found. 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* (line 3), and repeat the process.

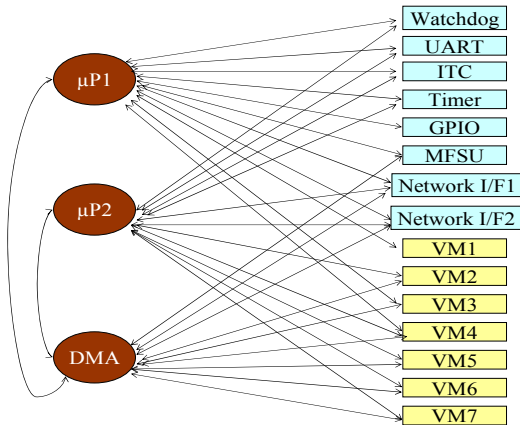
**Phase 5. optimize design:** Finally, we call the *optimize design* procedure for each of the  $N$  solutions obtained in the last phase. This simple procedure attempts to further reduce system cost by minimizing (i) bus speeds, (ii) arbitration scheme implementation cost and (iii) fix OO buffer sizes. 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 an arbitration scheme which is more expensive to implement (e.g. *TDMA/RR*) with one which is less expensive to implement (e.g. a static priority based scheme with priorities assigned depending on bandwidth requirements) at each SAP. Finally we fix the OO buffer sizes wherever applicable to the maximum number of buffers used during simulation of the application, 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. 8 PYTHON Communication Throughput Graph (CTG)**

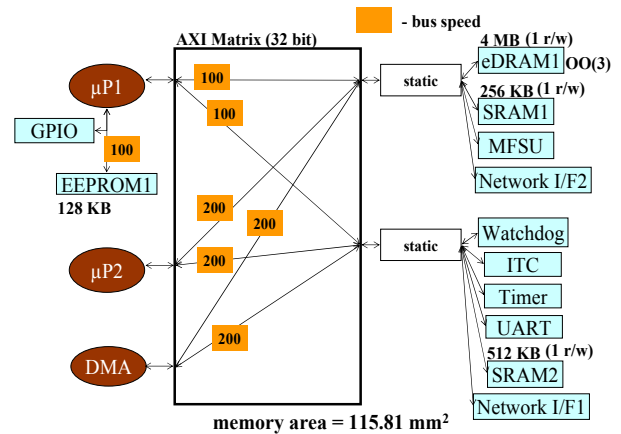
We will first consider the PYTHON MPSoC and make use of the *COSMECA* co-synthesis framework to synthesize memory and communication architectures for it. Fig. 8 shows the CTG for the PYTHON application, after the initial memory preprocessing phase in which DBs are merged into VMs. Not shown in the CTG, but included in our memory area analysis are the 32 KB instruction and data caches for each of the two processors. For clarity, the TCPs are presented separately in Table 2.  $\mu P1$  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.  $\mu P2$  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 to handle fast memory to memory and network interface data transfers, freeing up the processors for more useful work. PYTHON also has several peripherals such as a multi functional serial port interface (MFSU), a universal asynchronous receiver/transmitter block (UART), a general purpose I/O block (GPIO), timers (Timer, Watchdog), an interrupt controller (ITC) and two proprietary external network interfaces.

**Table 2. PYTHON Throughput Constraint Paths (TCPs)**

IP cores in Throughput Constraint Path (TCP)	TCP constraint
$\mu P2$ , VM2, VM3, Network I/F1, DMA, VM6	400 Mbps
$\mu P2$ , VM2, VM6, VM7, DMA, Network I/F2	960 Mbps
$\mu P1$ , MFSU, VM3, VM4, DMA, Network I/F1	400 Mbps
$\mu P2$ , VM4, VM5, VM7, DMA, Network I/F1, Network I/F2	600 Mbps

**Table 3. PYTHON Global Constraint Set  $\Psi_G$**

Set	Values
bus speed	25, 50, 100, 200, 300, 400
arbitration strategy	static, RR, TDMA/RR
OO buffer size	1 – 8
mem mapping	VM1 $\Rightarrow$ EEPROM



**Fig. 9 Synthesized Output for PYTHON**

Table 3 shows the global constraint set  $\Psi_G$  for PYTHON.

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 120 mm<sup>2</sup> and the estimated memory area numbers are for a 0.18- $\mu$ m technology. We assume the value for overlap threshold  $\tau = 10\%$  for this example. Fig. 9 shows the best solution (least number of busses) with the least memory area for PYTHON. The figure also shows bus speeds, memory sizes, number of ports and OO buffer sizes.

Fig. 10 shows the variation in memory area and number of busses in the matrix for the ten best solutions ( $N=10$ ), for PYTHON. From the figure we can see that no solution having 7 busses in the bus matrix exists for PYTHON. The dotted line indicates the solution shown in Fig. 9. We can see that there is a significant variation of combinations of memory area and number of busses, in the solution space. *COSMECA* thus allows a designer to tradeoff memory area and bus count during the solution selection process.

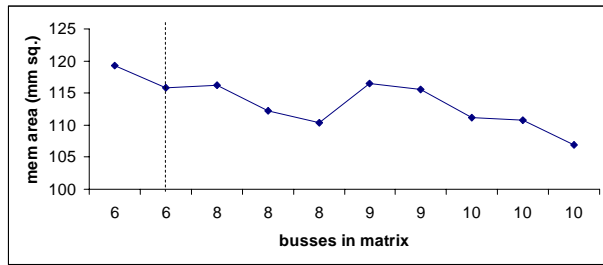


Fig. 10 PYTHON final solution space (for N=10)

During the course of the *COSMECA* co-synthesis flow, we made use of a threshold factor  $\tau$  (Fig. 7; *memmap heuristic*) to determine the extent to which virtual memories are merged at SAPs in the bus matrix. This parameter is specified by the designer. To understand the effect of this threshold factor  $\tau$  on the quality of solution, we varied the threshold value and repeated our *COSMECA* co-synthesis flow for the PYTHON MPSoC. The result of this experiment is shown in Fig. 11.

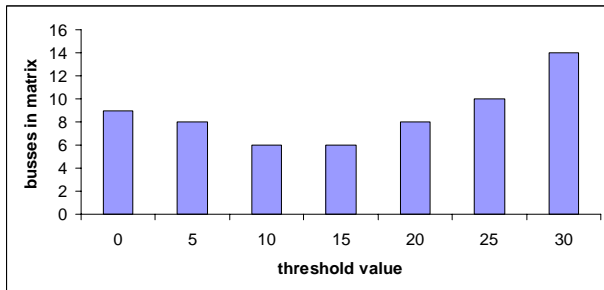


Fig. 11 Effect of varying threshold value on solution quality for PYTHON

It can be seen that for very low values of  $\tau$  (e.g.  $< 10\%$ ),

the number of busses in the matrix for the best solution is high. This is because low values of  $\tau$  discourage merger of virtual memories, which ends up creating a system with several physical memories that exceed memory area bounds due to their excessive area overhead. For larger values of  $\tau$  (e.g.  $\geq 20\%$ ), the number of busses for the best solution is also high, because it becomes harder to meet application throughput constraints with the large overlap. There might be slight variations to this trend, depending upon a complex amalgamation of factors such as stringency of throughput requirements, allowed maximum bus speeds, available memory port bandwidths and data traffic schedules for the application. Typically however, for the *COSMECA* co-synthesis framework, our experience shows that lower values around 10 – 20% for overlap threshold  $\tau$  give the best quality solutions.

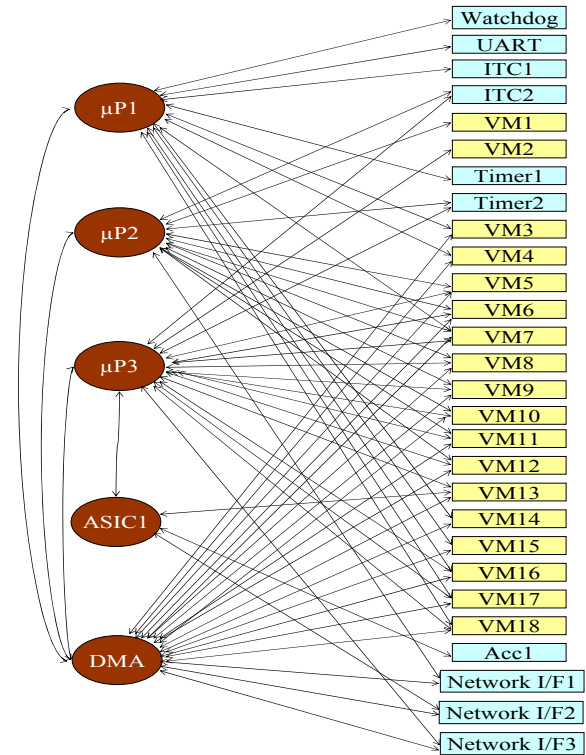


Fig. 12 SIRIUS Communication Throughput Graph (CTG)

Next we consider a more complex application: the SIRIUS MPSoC, and go into more detail of how it was used as another driver for the *COSMECA* framework. Fig. 12 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 4.  $\mu$ P1 is a protocol processor (PP) while  $\mu$ P2 and  $\mu$ P3 are network processors (NP). The  $\mu$ P1 PP is responsible for setting up and closing network connections, converting data from one protocol

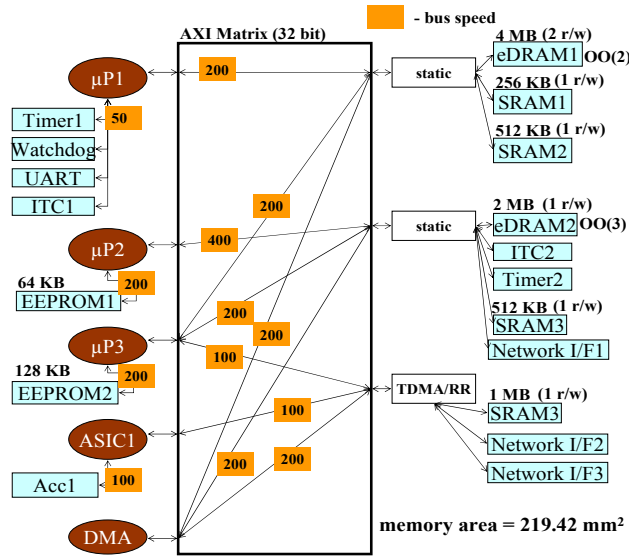
type to another, generating data frames for signaling, operating and maintenance and exchanging data with NP using shared memory. The  $\mu P2$  and  $\mu P3$  NPs directly interact with the network ports and are used for assembling incoming packets into frames for the network connections, network port 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 4. SIRIUS Throughput Constraint Paths (TCPs)**

IP cores in Throughput Constraint Path (TCP)	TCP constraint
$\mu P1$ , VM3, VM4, DMA, VM16, VM17, VM18	640 Mbps
$\mu P1$ , VM5, VM6, VM14, VM15, DMA, Network I/F2	480 Mbps
$\mu P2$ , Network I/F1, VM8, VM9	5.2 Gbps
$\mu P2$ , VM10, VM11, VM12, DMA, Network I/F3	1.4 Gbps
ASIC1, $\mu P3$ , VM16, VM17, VM18, Acc1, VM13, Network I/F2	240 Mbps
$\mu P3$ , DMA, Network I/F3, VM13	2.8 Gbps

**Table 5. SIRIUS Global Constraint Set  $\Psi_G$**

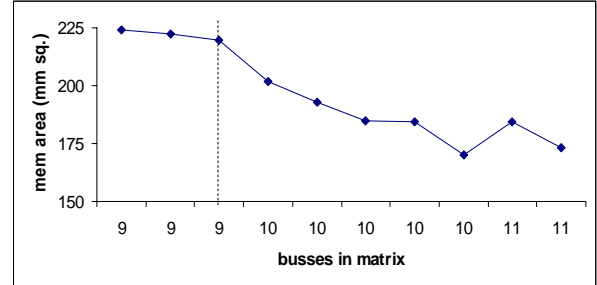
Set	Values
bus speed	25, 50, 100, 200, 300, 400
arbitration strategy	static, RR, TDMA/RR
OO buffer size	1 – 8
mem mapping	VM16, VM17 $\Rightarrow$ DRAM; VM1, VM2 $\Rightarrow$ EEPROM



**Fig. 13 Synthesized output for SIRIUS**

Table 5 shows the global constraint set  $\Psi_G$  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

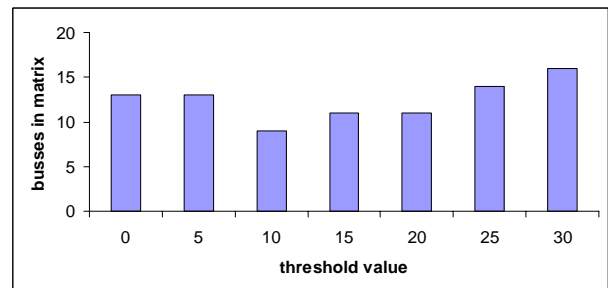
to 225 mm<sup>2</sup> and the estimated memory area numbers are for a 0.18- $\mu$ m technology. We assume the value for overlap threshold  $\tau = 10\%$  for this example. Fig. 13 shows the best solution (least number of busses) with the least memory area for SIRIUS. The figure also shows bus speeds, memory sizes, number of ports and OO buffer sizes.



**Fig. 14 SIRIUS final solution space (for N=10)**

Fig. 14 shows the variation in memory area and number of busses for the ten best solutions ( $N=10$ ) for SIRIUS. The dotted line indicates the solution shown in Fig. 13. 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. Again, the key observation from this experiment is that *COSMECA* enables a designer to select a solution having the desired tradeoff between memory area and bus count in the matrix.

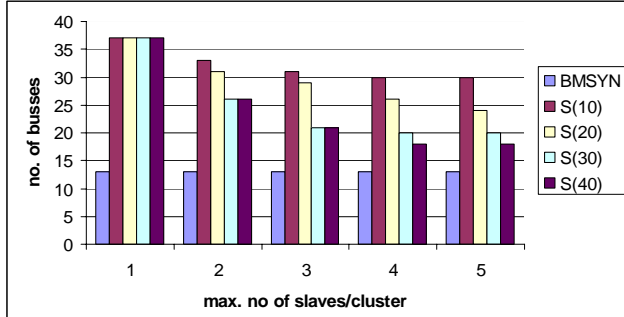
To determine the impact of varying the threshold factor  $\tau$  on the quality of solution for the SIRIUS MPSoC, we varied the threshold value and repeated our *COSMECA* co-synthesis flow for SIRIUS. The result of this experiment is shown in Fig. 15. The trend for this experiment is similar to our observation for Fig. 11, which showed the results for this experiment on the PYTHON MPSoC. As observed earlier, lower values around 10 – 20% for overlap threshold  $\tau$  give the best quality solutions for the SIRIUS application.



**Fig. 15 Effect of varying threshold value on solution quality for SIRIUS**

The entire *COSMECA* flow took only a few hours to complete, including simulation time, for each of the four

MPSoC applications considered. 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.

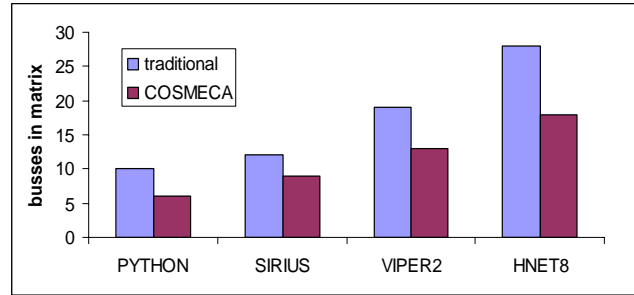


**Fig. 16 Comparison of bus matrix synthesis approach (BMSYN) used in COSMECA with a threshold based approach for SIRIUS MPSoC**

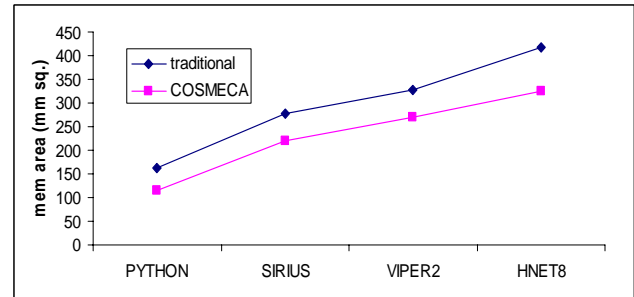
Next, we will compare the quality of the results obtained from the bus matrix communication architecture synthesis approach used in *COSMECA*, with the closest existing piece of work that deals with automated matrix synthesis with the aim of minimizing number of busses [30]. Since their bus matrix synthesis 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 [30] requires the designer to statically specify (i) the maximum number of slaves per cluster and (ii) the traffic overlap threshold, which if exceeded prevents two slaves from being assigned to the same bus cluster. The results of our comparison study are shown in Fig. 16. BMSYN is the name given to the bus matrix synthesis approach used in *COSMECA*, while the other comparison points are obtained from [30].  $S(x)$ , for  $x = 10, 20, 30, 40$ , represents the threshold based approach where no two slaves having a traffic overlap of greater than  $x\%$  can be assigned to the same bus, and the X-axis in Fig. 16 varies the maximum number of slaves allowed in a bus cluster for these comparison points. The values of 10 – 40% for traffic overlap are chosen as per recommendations from [30]. It is clear from Fig. 16 that our bus matrix synthesis approach used in *COSMECA* produces a lower cost system (having lesser number of busses) than approaches which force the designer to statically approximate application characteristics.

Finally, Fig. 17 and 18 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. 17 Comparison of best solution bus count**



**Fig. 18 Comparison of best solution memory area**

## 7 Conclusion and Future Work

In this technical report, we have presented an automated application specific framework 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.

## 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 Approach", *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](http://www.arm.com/armtech/AXI)
- [15] ARM AMBA Specification (rev2.0), [www.arm.com](http://www.arm.com), 2001
- [16] "IBM On-chip CoreConnect Bus Architecture", [www.chips.ibm.com](http://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](http://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] A. Nandi, R. Marculescu, "System-level power/performance analysis for embedded systems design", *DAC 2001*
- [36] A. Pinto, L. Carloni, A. Sangiovanni-Vincentelli, "Constraint-driven communication synthesis", *DAC 2002*
- [37] K. K. Ryu, V. J. Mooney III, "Automated Bus Generation for Multiprocessor SoC Design", *DATE 2003*
- [38] M. Gasteier, M. Glesner, "Bus-based communication synthesis on system level", *ACM TODAES, January 1999*
- [39] D. Lyonard, S. Yoo, A. Baghdadi, A. A. Jerraya, "Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip", *DAC 2001*

- [40] S. Pasricha, N. Dutt, M. Ben-Romdhane, "Automated Throughput-driven Synthesis of Bus-based Communication Architectures", *In Proc of ASPDAC 2005*
- [41] U. Ogras, R. Marculescu, "Energy- and Performance-Driven NoC Communication Architecture Synthesis using a Decomposition Approach", *DATE 2005*
- [42] A. Pinto, L. P. Carloni, A. L. Sangiovanni-Vincentelli, "Efficient Synthesis of Networks On Chip," *ICCD 2003*
- [43] A. Jalabert, S. Murali, L. Benini, G. De Micheli. "xpipesCompiler: A Tool for instantiating application specific Networks on Chip," *DATE 2004*
- [44] S. Pasricha, N. Dutt, M. Ben-Romdhane, "Extending the Transaction Level Modeling Approach for Fast Communication Architecture Exploration", *DAC 2004*
- [45] S. Pasricha, "Transaction Level Modeling of SoC with SystemC 2.0", *SNUG, 2002*
- [46] S. Pasricha, N. Dutt, M. Ben-Romdhane, "Constraint-Driven Bus Matrix Synthesis for MPSoC", *ASPDAC 2006*