A Framework for Cosynthesis of Memory and Communication Architectures for MPSoC

Sudeep Pasricha, *Student Member, IEEE*, and Nikil D. Dutt, *Senior Member, IEEE*

***Abstract*—Memory and communication architectures have a significant impact on the cost, performance, and time-to-market of complex multiprocessor 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 cosynthesize the memory and communication architectures to avoid making suboptimal design decisions. This is in contrast to traditional platform-based design approaches where memory and communication architectures are synthesized separately. In this paper, the authors propose an automated application-specific cosynthesis framework for memory and communication architecture (COSMECA) in MPSoC designs. The primary objective is to design a communication architecture having the least number of buses, 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 buses and 29% in memory area compared to the traditional approach.**

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

I. Motivation

MODERN multiprocessor 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 copes with the entire intercomponent 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 up to 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].

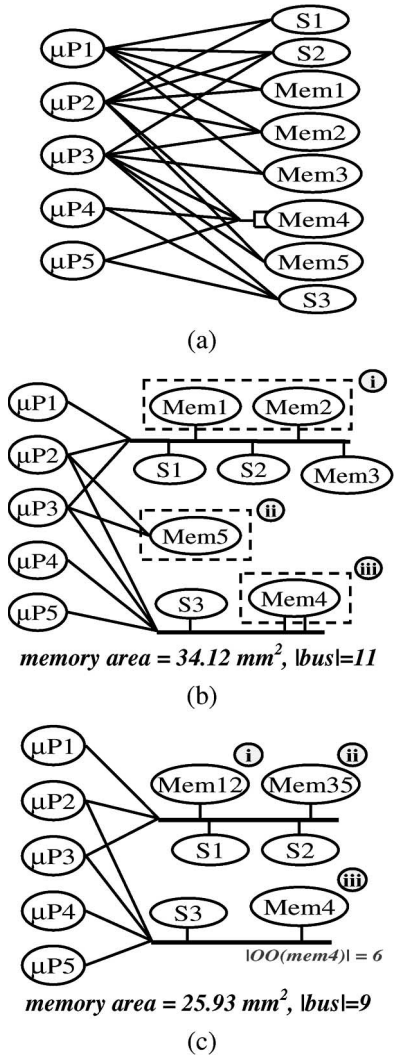


Fig. 1. Comparison of traditional (separate synthesis) approach and cosynthesis approaches for MPSoC example. (a) MPSoC system. (b) Result of performing memory synthesis before communication-architecture synthesis. (c) Result of performing cosynthesis of memory and communication architectures.

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 suboptimal 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 intellectual properties (IPs) defined, including memory, which is synthesized based on data size and high-level bandwidth

Manuscript received March 17, 2006; revised June 28, 2006. This work was supported in part by grants from Semiconductor Research Corporation (SRC) (2005-HJ-1330) and in part by a Center for Pervasive Communications and Computing (CPCC) fellowship. This paper was recommended by Guest Editor D. Sciuto.

The authors are with the Center for Embedded Computer Systems, University of California, Irvine, CA 92697 USA (e-mail: sudeep@cecs.uci.edu; dutt@cecs.uci.edu).

Digital Object Identifier 10.1109/TCAD.2006.884487

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 cosynthesized using the cosynthesis framework for memory and communication architecture (COSMECA) approach. Now, let us consider the implications of using a cosynthesis framework. First, the cosynthesis 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. Second, the cosynthesis 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 buses, 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 presynthesized physical memories are such that they cause traffic conflicts that violate bandwidth constraints. Third, due to the knowledge of support for out-of-order (OO) transaction completion [14] by the communication architecture, the cosynthesis 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 cosynthesis 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 paper, we propose an automated application-specific COSMECA in MPSoC designs. The primary objective is to design a communication architecture having the least number of buses, 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.

COSMECA tailors the memory and communication architectures to the application being considered to reduce the 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. In its essence, COSMECA is a novel memory and communication-architecture cosynthesis framework, which improves upon existing synthesis approaches by: 1) automatically generating bus topology and parameter values for arbitration schemes, bus speeds and OO buffer sizes, while considering dynamic simulation effects and 2) 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. To the best of our knowledge, no previous work has performed automated cosynthesis 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 buses and 29% in memory area compared to the traditional approach of separate synthesis.

II. 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 buses 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. Ogawa *et al.* [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 cosynthesis 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. Shalan *et al.* [31] present a tool to automatically generate a full crossbar and a dynamic memory management unit (DMMU). Grun *et al.* [32] consider the connectivity topology early in the design flow in conjunction with memory exploration, for simple processor-memory systems. Kim *et al.* [33] deal with bus topology and static priority-based arbitration exploration to determine the best memory

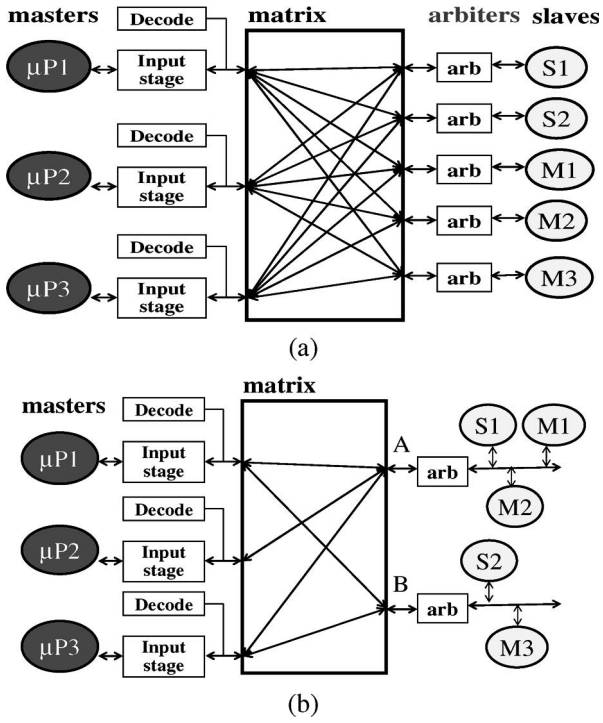


Fig. 2. Bus-matrix communication architecture. (a) Full bus-matrix architecture. (b) Partial bus-matrix architecture.

port-to-bus mapping for presynthesized memory blocks. More recently, Srinivasan *et al.* [47] present an approach to simultaneously consider bus-topology splitting and memory bank partitioning during synthesis. While they consider a limited design space compared to our approach (they do not consider the effect of communication parameters or different memory types), their focus is on the problem of system energy reduction, which is not currently addressed by our approach. Other approaches that deal with memory synthesis make use of static estimations of communication architectures such as those proposed in [34] and [35]. Such approaches are unable to capture dynamic effects such as contention and address only a limited exploration space. More importantly, none of the aforementioned approaches attempts to perform cosynthesis. COSMECA is a novel memory and communication-architecture cosynthesis framework that improves upon existing synthesis approaches by: 1) automatically generating bus topology and parameter values for arbitration schemes, bus speeds, and OO buffer sizes, while considering dynamic simulation effects and 2) 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 VI) emphasize the usefulness and need of such an approach for MPSoC designs.

III. 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 buses in parallel that 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 buses 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 buses. In the AXI [14] specification for instance, each bus consists of read address, write address, read data, write data, write response, and control signals. 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 buses 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 buses 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 buses in the matrix. Additionally, we generate arbitration schemes at the SAPs, bus-clock-speed values, and OO buffer size values.

IV. 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 the 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 nonvolatile 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 equal and 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. Note that, since the focus of this paper is not on system-level energy reduction, we do not perform fine grain application level data reuse analysis to cluster frequently accessed data onto a smaller memory like in [47] and [48]. The grouping of data blocks (DBs) in our approach allows us to reduce the number of memories in the design, thus reducing area. We also try to avoid multiport memories because of their excessive area and cost overhead.

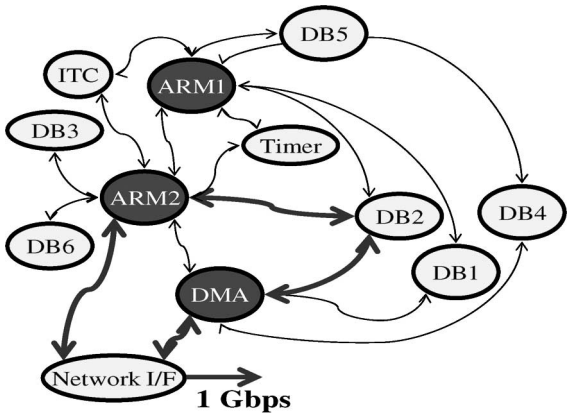


Fig. 3. Communication throughput graph (CTG).

V. COSMECA Cosynthesis Framework

This section describes the COSMECA cosynthesis 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 cosynthesis process. Finally, we describe the COSMECA cosynthesis flow in detail.

A. Assumptions and Problem Definition

We are given an application for which we assume that 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 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, which is 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 ν 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 subgraph 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 Gb/s.

Problem definition: A bus $\mathbf{B}$ can be considered to be a partition of the set of components $\mathbf{V}$ in a CTG, where $\mathbf{B} \subset \mathbf{V}$. Then, our primary objective is to determine an optimal component to bus assignment for a bus-matrix architecture, such that the partitioning of $\mathbf{V}$ onto $\mathbf{N}$ buses results in a minimal number of buses $\mathbf{N}$ 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 the memory-area cost of the solution.

B. 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 concurrence, 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 kHz [26], [44], while running embedded software applications on processor instruction set simulator (ISS) models. The communication model in this abstraction is extremely detailed, capturing delays arising due to frequency and data width adapters, bridge overheads, interface buffering, and all the static and dynamic delays associated with the standard bus-architecture protocol being used.

C. Communication-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 buses 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 a maximum of 250 MHz. This provides a convenient mechanism for the designer to bias the cosynthesis process based on knowledge of the design and the technology being targeted. Such knowledge about the design is not a prerequisite for using our cosynthesis framework, but informed decisions can help avoid the synthesis of unrealistic system configurations. The size of the constraint

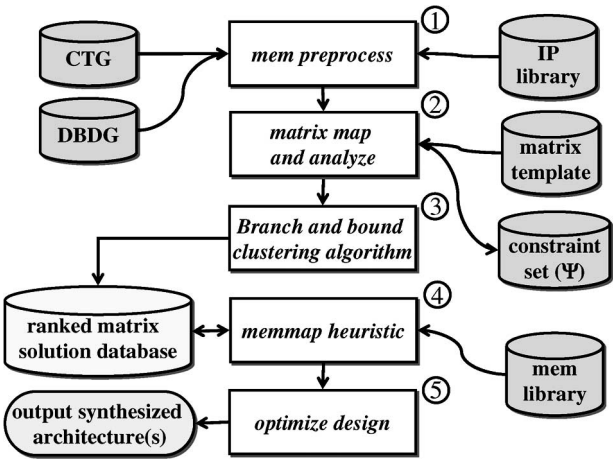


Fig. 4. COSMECA cosynthesis flow.

set directly affects the time taken for cosynthesis. The larger the number of values for the communication parameter constraints in the set (e.g., larger number of allowed bus speeds), the longer it takes for the cosynthesis framework to arrive at a solution since a larger design space has to be considered. However, the larger the number of memory mapping constraints in the set, the lesser is the amount of time taken to arrive at a solution, since the memory mappings to be considered during cosynthesis are now reduced.

D. COSMECA Cosynthesis Flow

We describe the COSMECA cosynthesis flow in more detail in this section. Fig. 4 gives a high-level overview of the flow. The inputs to COSMECA include a CTG, a library of behavioral IP models (IP library) and memory models (mem library), a DB dependence 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 nonconflicting DBs into 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 buses. Next, we perform a static branch and bound hierarchical clustering of slave components in the matrix, which further reduces the number of buses, 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 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 the memory area and performance constraints. Finally, we optimize the output solutions to reduce bus speeds, arbitration costs, and prune 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 DBs in the CTG into VMs to reduce the memory-area cost by potentially reducing the number of memory modules in the system. Only DBs satisfying the two criteria of having: 1) similar edges (i.e., edges from the same masters) and 2) nonoverlapping 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 DBDG to determine if DBs have nonoverlapping access. The DBDG is a directed graph, which shows the dependence 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 dependence between DBs—A DB cannot be accessed until 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 nonoverlapping 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 the memory space.

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 buses on which there are no data transfers. Dedicated slave and memory components are also migrated to the local buses of their corresponding masters to further reduce the buses 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 $(\Gamma_{\mathrm{TLM}/B})$ from the TLM simulation, for any bus B in the matrix, is given by

$$\Gamma_{\mathrm{TLM}/B} = (\mathrm{num}T_B \times \ \mathrm{size}T_B \times \mathrm{width}_B \times \Omega_B)/\sigma$$

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

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

$$\therefore \quad \Omega_B \geq (\sigma \times \Gamma_{\mathrm{TCP}/B})/(\mathrm{num}T_B \times \mathrm{size}T_B \times \mathrm{width}_B).$$

The minimum bus speed thus found is used to create (or update) the local bus speed constraint set $\Psi_{L(\mathrm{speed})}$ 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 buses in the matrix even further. Note that we do

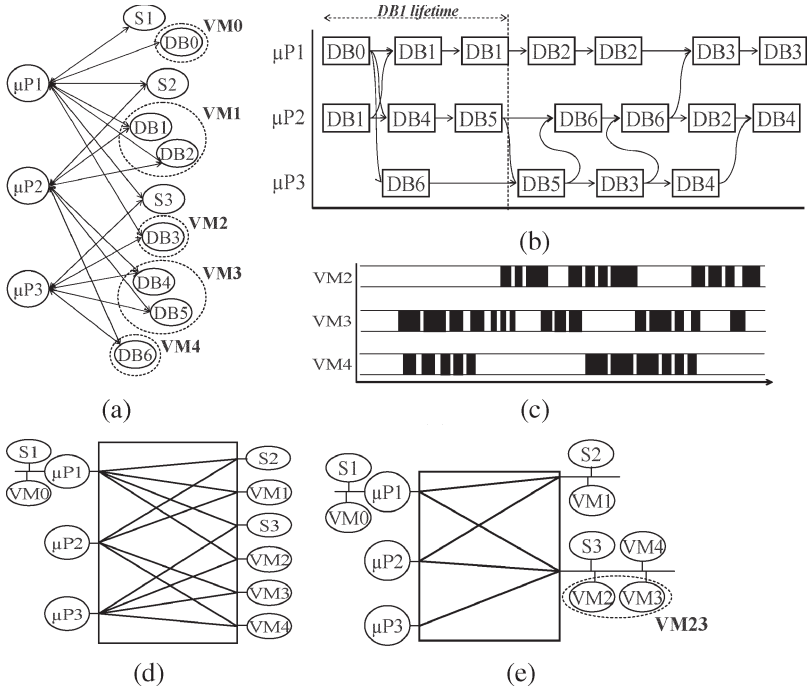


Fig. 5. COSMECA cosynthesis example. (a) CTG. (b) DBDG. (c) VM access trace. (d) Reduced matrix. (e) Best synthesized solution.

not consider clustering masters in the matrix in our approach. While clustering masters can result in some savings for some SoC systems, for the highly parallel high-performance multiprocessor SoC applications that we target, clustering masters can drastically degrade the system performance. This is because master clustering adds two levels of contention: one at the master end and another at the slave end, in a data path, which lengthens the completion time for transactions issued by any of the clustered masters. Additionally, clustering masters also severely limits the parallelism in the system, since if one master in a "master cluster" is active with a transaction, none of the other masters in that cluster can issue transactions. In our experience, even increasing the bus clock frequency to compensate for the reduced parallelism and longer transaction latency in the system is unable to resolve the resulting throughput constraint violations.

Before describing the algorithm, we present a few definitions. A slave cluster SC $= \{s_1, \ldots, s_n\}$ refers to an aggregation of slaves that share a common arbiter. Let M_{SC} refer to the set of masters connected to a slave cluster SC. Next, let $\Pi_{\mathrm{SC1/SC2}}$ be a superset of sets of buses, which are merged when slave clusters SC1 and SC2 are merged. Finally, for a merged bus set $\beta = \{b1, \ldots, bn\}$, where $\beta \subset \Pi_{\mathrm{SC1/SC2}}$, K_β refers to the set of allowed bus speeds for the newly created bus when the buses in set β are merged, and is given by

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

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

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 (($K_\beta == \phi$) || ($\sum_{i=1}^{|\beta|} \Gamma_{TCP/i} > (\mathrm{width_B} \times \mathrm{max\_speed_B})$)) **then**
bound clustering

Fig. 6. Bound function.

to traverse. In order to consider only valid clustering configurations, we make use of a bounding 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 lookup 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 buses 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 buses 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 buses 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 buses 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

```
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 ≤ τ %
7:      for each VM do
8:        if (VM data overlap ≤ τ %)
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.

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 buses) 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 Ψ for the components that 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].

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 pseudocode for the memmap heuristic. The goal is to find N solutions that 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 earlier (in Section V-B). 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 τ, 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 τ. 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, the 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 Ψ. It is possible that a VM has self-conflict greater than τ, 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 the 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 that meets the size (capacity) requirements but has a 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 until 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 the system cost by minimizing: 1) the bus speeds; 2) the arbitration-scheme implementation cost; and 3) fix OO buffer sizes. The procedure first iterates over the buses 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 that is more expensive to implement (e.g., TDMA/RR) with one that 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.

TABLE I
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

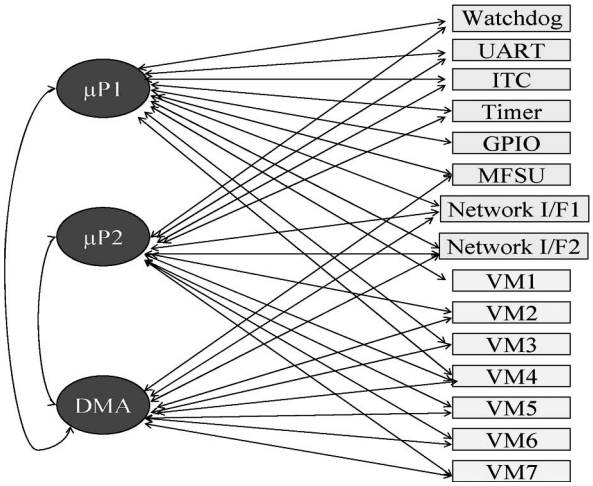


Fig. 8. PYTHON CTG.

VI. 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 I 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 cosynthesized with the communication architecture later. While this simulation speed of our system-level modeling abstraction (Section V-B) is fast for the amount of detail that it captures, modern MPSoC applications such as the ones we consider can still take several hours to simulate in their entirety. In order to reduce this overhead, we make use of representative testbenches for each of these applications, which capture the critical portions of the application functionality in a shorter execution time.

We will first consider the PYTHON MPSoC and make use of the COSMECA cosynthesis 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 II.

$\mu P1$ 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

TABLE II
PYTHON TCPS

IP cores in Throughput Constraint Path (TCP)	TCP constraint
μP2, VM2, VM3, Network I/F1, DMA, VM6	400 Mbps
μP2, VM2, VM6, VM7, DMA, Network I/F2	960 Mbps
μP1, MFSU, VM3, VM4, DMA, Network I/F1	400 Mbps
μP2, VM4,VM5,VM7, DMA, Network I/F1, Network I/F2	600 Mbps

TABLE III
PYTHON 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	VM1=>*EEPROM*

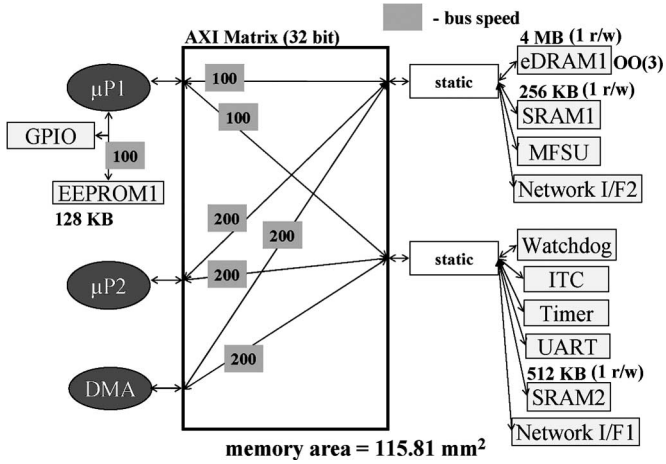


Fig. 9. Synthesized output for PYTHON.

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 multifunctional serial port interface, a universal asynchronous receiver/transmitter block (UART), a general purpose I/O block, timers (Timer, Watchdog), an interrupt controller (ITC), and two proprietary external network interfaces.

Table III shows the global constraint set Ψ_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^2, and the estimated memory-area numbers are for a 0.18-μm technology. We assume the value for overlap threshold $\tau = 10\%$ for this example. Fig. 9 shows the best solution (least number of buses) 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 buses in the matrix for the ten best solutions $(N = 10)$, for PYTHON. From the figure, we can see that no solution having seven buses in the bus matrix exist for PYTHON. The dotted

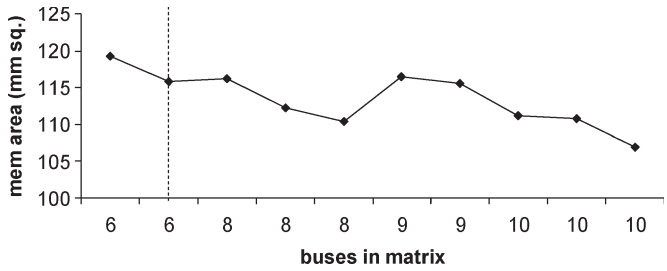


Fig. 10. PYTHON final solution space (for $N = 10$).

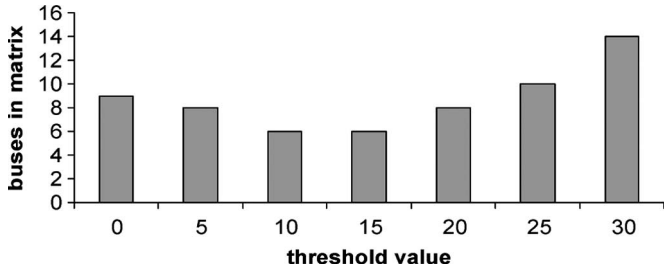


Fig. 11. Effect of varying threshold value on solution quality for PYTHON.

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 buses in the solution space. COSMECA thus allows a designer to tradeoff memory area and bus count during the solution selection process.

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

It can be seen that for very low values of τ (e.g., $< 10\%$), the number of buses in the matrix for the best solution is high. This is because low values of τ discourage merger of VMs, 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 τ (e.g., $> 20\%$), the number of buses 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 cosynthesis framework, our experience shows that lower values around 10%–20% for overlap threshold τ give the best quality solutions.

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.

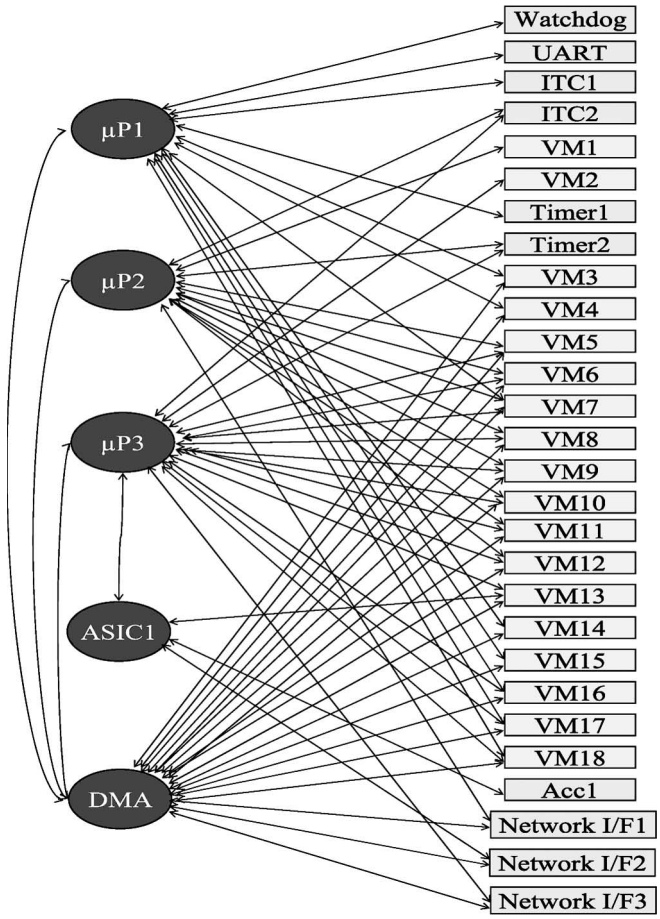


Fig. 12. SIRIUS CTG.

TABLE IV
SIRIUS THROUGHPUT 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

For clarity, the TCPs are presented separately in Table IV. $\mu P1$ 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

TABLE V
SIRIUS GLOBAL PATH 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*

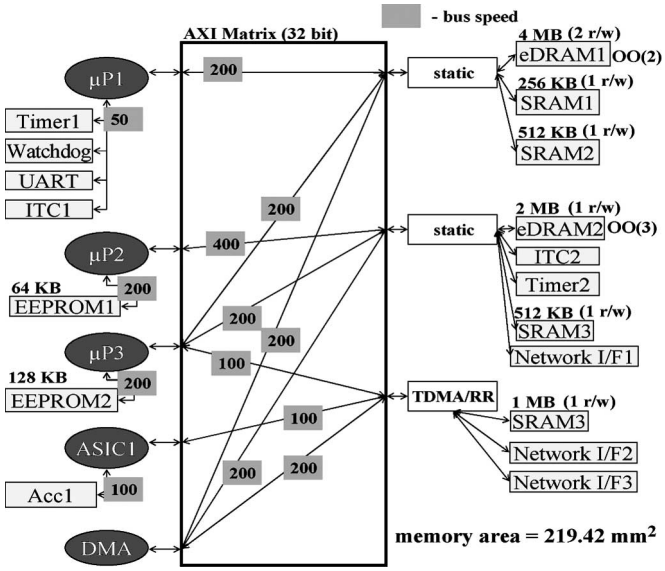


Fig. 13. Synthesized output for SIRIUS.

such as ITs (ITC1, ITC2), a UART, timers (Watchdog, Timer1, Timer2), and a packet accelerator (Acc1).

Table V 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 $\tau = 10\%$ for this example. Fig. 13 shows the best solution (least number of buses) with the least memory area for SIRIUS. The figure also shows bus speeds, memory sizes, number of ports, and OO buffer sizes.

It should be noted that while COSMECA allows a designer the flexibility to assign variable bus clock frequencies for the buses in the matrix, this entails an overhead in the form of frequency converters at the interfaces (which might use buffering for timing isolation). As an alternative, a single low frequency for all the buses in the matrix is usually practically insufficient to meet high-throughput requirements. In contrast, a higher fixed frequency for the entire matrix can end up dissipating excessive power in the bus logic and bus wires. Therefore, a designer needs to be aware of this tradeoff. Note that COSMECA can be made to synthesize a matrix with either different or one fixed bus clock frequency for the buses in the matrix.

Fig. 14 shows the variation in memory area and number of buses 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

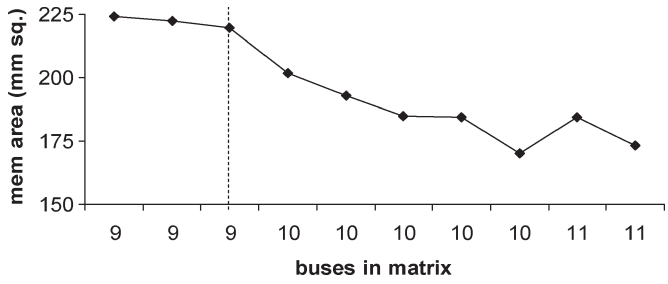


Fig. 14. SIRIUS final solution space (for $N = 10$).

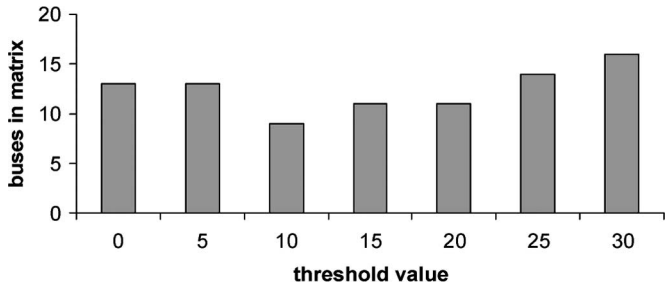


Fig. 15. Effect of varying threshold value on solution quality for SIRIUS.

TABLE VI
COSYNTHESIS TIME FOR MPSOC APPLICATIONS

Applications	Simulation runs	Total co-synthesis time (in hours)
PYTHON	13	3.5
SIRIUS	19	8.6
VIPER2	26	17.8
HNET8	38	28.5

of buses), 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 τ on the quality of solution for the SIRIUS MPSoC, we varied the threshold value and repeated our COSMECA cosynthesis 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 τ give the best quality solutions for the SIRIUS application.

We now present results for the number of simulation runs and total time taken during cosynthesis. Table VI shows the total number of simulation runs and total simulation time in hours for the MPSoC applications. Note that the contribution of the static estimation phases such as the branch and bound clustering is almost negligible, and simulation takes up most of the time during cosynthesis. It can be seen that the entire COSMECA flow took in the order of hours to generate the best solution 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.

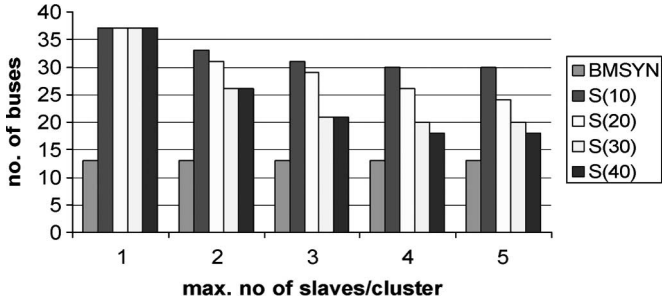


Fig. 16. Comparison of bus-matrix-synthesis approach (BMSYN) used in COSMECA with a threshold-based approach for SIRIUS MPSoC.

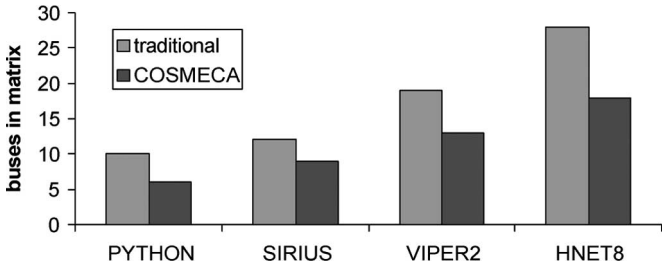


Fig. 17. Comparison of best solution bus count.

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 buses [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 buses in the final synthesized design. The threshold-based approach proposed in [30] requires the designer to statically specify: 1) the maximum number of slaves per cluster and 2) the traffic overlap threshold, which if exceeded prevents two slaves from being assigned to the same bus cluster. The results of our comparison study are shown in Fig. 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]. The number of slaves in a cluster has been limited to five in the figure, because no reduction in bus count was apparent for higher values of clustering. This is because of the inherent limitations in [30], which prevent critical data streams from sharing a bus, and also because the effect of communication parameters, such as arbitration strategies and bus speeds on performance during synthesis, is not considered. It is clear from Fig. 16 that our bus-matrix-synthesis approach used in COSMECA produces a lower cost system (having lesser number of buses) than approaches, which force the designer to statically approximate application characteristics.

Finally, Figs. 17 and 18 compare the number of buses and memory areas for the best solution (having least number of buses, 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 buses 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 DBs to physical-memory components.

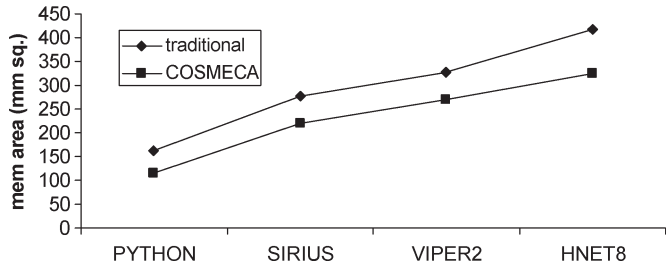


Fig. 18. Comparison of best solution memory area.

VII. Conclusion and Future Work

In this paper, we have presented an automated application-specific framework to cosynthesize memory and communication architectures (COSMECA) in MPSoC designs. The primary objective is to design a communication architecture having the least number of buses, 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 buses 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 cosynthesis and including cache customization in the memory synthesis process.

Appendix A
Acronyms

In this section, we present a list of the acronyms used in this paper and their corresponding expanded full forms.

Acronym	Expanded form
BMSYN	Bus-Matrix SYNthesis framework.
CDFG	Control data flow graph.
CTG	Communication throughout graph.
DB	Data block.
DBDG	Data block dependence graph.
DMA	Direct memory access.
DMMU	Direct memory management unit.
DRAM	Dynamic random access memory.
EPROM	Erasable programmable read-only memory.

EEPROM	Electrically erasable programmable read-only memory.
IP	Intellectual property block.
ISS	Instruction set simulator for processors.
MPSoC	MultiProcessor system-on-chip.
NOC	Network-on-chip.
OO	Out of order.
RR	Round robin arbitration.
SC	Slave cluster.
SAP	Slave access point.
SRAM	Static random access memory.
TCP	Throughput constraint graph.
TDMA	Time division multiple access arbitration.
TLM	Transaction level model.
VM	Virtual memory.

APPENDIX B
SYMBOLS

In this section, we present a list of the major symbols introduced in this paper and their corresponding connotations.

Symbol	Connotation
Ψ	Communication-memory constraint set.
Ψ_{G}	Global communication-memory constraint set.
Ψ_{L}	Local communication-memory constraint set.
$\Psi_{L(\mathrm{speed})}$	Bus speed constraint in set Ψ_{L}.
$\Gamma_{\mathrm{TLM}/B}$	Data throughput for bus B in matrix obtained from TLM simulation.
$\Gamma_{\mathrm{TCP}/B}$	Throughput constraint for bus B in matrix.
Ω_B	Clock speed for bus B.
σ	Total number of cycles for application execution from TLM simulation.
M_{SC}	Set of masters connected to slave cluster SC.
$\Pi_{\mathrm{SC1/SC2}}$	Superset of sets of buses, which are merged when slave clusters SC1 and SC2 are merged.
β	Set of buses that are merged during a clustering operation.
ϕ	Null set.
τ	Overlap threshold.

REFERENCES

[1] D. Sylvester and K. Keutzer, "Getting to the bottom of deep submicron," in *Proc. ICCAD*, 1998, pp. 203–211.

[2] S. Pasricha, N. Dutt, E. Bozorgzadeh, and M. Ben-Romdhane, "Floorplan-aware automated synthesis of bus-based communication architectures," in *Proc. DAC*, 2005, pp. 565–570.

[3] S. Meftali, F. Gharsalli, F. Rousseau, and A. A. Jerraya, "An optimal memory allocation for application-specific multiprocessor system-on-chip," in *Proc. ISSS*, 2001, pp. 19–24.

[4] A. Allan *et al.*, "2001 Technology roadmap for semiconductors," *Computer*, vol. 35, pp. 42–53, Jan. 2002.

[5] J. A. Rowson and A. Sangiovanni-Vincentelli, "Interface based design," in *Proc. DAC*, 1997, pp. 178–183.

[6] K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: Orthogonalization of concerns and platform-based design," in *Proc. IEEE TCAD*, Dec. 2000, pp. 1523–1543.

[7] J-M. Daveau, T. B. Ismail, and A. A. Jerraya, "Synthesis of system-level communication by an allocation-based approach," in *Proc. ISSS*, 1995, pp. 150–155.

[8] S. Narayan and D. Gajski, "Protocol generation for communication channels," in *Proc. DAC*, 1994, pp. 547–551.

[9] I. Madsen and B. Hald, "An approach to interface synthesis," in *Proc. ISSS*, 1995, pp. 16–21.

[10] S. Wuytack, F. Catthoor, G. De Jong, and H. J. De Man, "Minimizing the required memory bandwidth in VLSI system realizations," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 7, no. 4, pp. 433–441, Dec. 1999.

[11] L. Cai, H. Yu, and D. Gajski, "A novel memory size model for variable-mapping in system level design," in *Proc. ASP-DAC*, 2004, pp. 813–818.

[12] K. Lahiri, A. Raghunathan, and S. Dey, "System-level performance analysis for designing system-on-chip communication architecture," in *Proc. IEEE TCAD*, Jun. 2001, pp. 768–783.

[13] P. Knudsen and J. Madsen, "Integrating communication protocol selection with partitioning in hardware/software codesign," in *Proc. ISSS*, 1998, pp. 111–116.

[14] *ARM AMBA AXI Specification rev 1.0*. (2004, Mar.). [Online]. Available: www.arm.com/products/solutions/axi_spec.html

[15] *ARM AMBA Specification rev 2.0*. (1999, May). [Online]. Available: www.arm.com/products/solutions/AMBA_Spec.html

[16] *IBM On-chip CoreConnect Bus Architecture*. [Online]. Available: www.chips.ibm.com

[17] "STBus communication system: Concepts and definitions," in *Reference Guide*, STMicroelectronics, Geneva, Switzerland, May 2003.

[18] M. Nakajima *et al.*, "A 400 MHz 32b embedded microprocessor core AM34-1 with 4.0 Gb/s cross-bar bus switch for SoC," in *Proc. ISSCC*, 2002, pp. 274–504.

[19] *SystemC Language Reference Manual, ver 2.1*. (2005, May). [Online]. Available: www.systemc.org/web/sitedocs/lrm_2_1.html

[20] L. Benini and G. D. Micheli, "Networks on chips: A new SoC paradigm," in *Proc. IEEE Comput.*, Jan. 2002, pp. 70–78.

[21] J. Henkel, W. Wolf, and S. Chakradhar, "On-chip networks: A scalable, communication-centric embedded system design paradigm," in *Proc. VLSI Des.*, 2004, pp. 845–851.

[22] V. Lahtinen, E. Salminen, K. Kuusilinna, and T. Hamalainen, "Comparison of synthesized bus and crossbar interconnection architectures," in *Proc. ISCAS*, 2003, pp. 433–436.

[23] K. K Ryu, E. Shin, and V. J. Mooney, "A comparison of five different multiprocessor SoC bus architectures," in *Proc. DSS*, 2001, pp. 202–209.

[24] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon, "Analyzing on-chip communication in a MPSoC environment," in *Proc. DATE*, 2004, pp. 752–757.

[25] M. Gasteier and M. Glesner, "Bus-based communication synthesis on system level," in *Proc. ACM TODAES*, Jan. 1999, pp. 65–70.

[26] S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Fast exploration of bus-based on-chip communication architectures," in *Proc. CODES+ISSS*, 2004, pp. 242–247.

[27] K. Srinivasan, K. S. Chatha, and G. Konjevod, "Linear programming based techniques for synthesis of network-on-chip architectures," in *Proc. ICCD*, 2004, pp. 422–429.

[28] D. Bertozzi *et al.*, "NoC synthesis flow for customized domain specific multiprocessor systems-on-chip," in *Proc. IEEE TPDS*, Feb. 2005, pp. 113–129.

[29] O. Ogawa *et al.*, "A practical approach for bus architecture optimization at transaction level," in *Proc. DATE*, 2003, pp. 176–181.

[30] S. Murali and G. De Micheli, "An application-specific design methodology for STbus crossbar generation," in *Proc. DATE*, 2005, pp. 1176–1181.

[31] M. Shalan, E. Shin, and V. Mooney, "DX-Gt: Memory management and crossbar switch generator for multiprocessor system-on-a-chip," in *Proc. SASIMI*, 2003, pp. 357–364.

[32] P. Grun, N. Dutt, and A. Nicolau, "Memory system connectivity exploration," in *Proc. DATE*, 2002, pp. 894–901.

[33] S. Kim, C. Im, and S. Ha, "Efficient exploration of on-chip bus architectures and memory allocation," in *Proc. CODES+ISSS*, 2004, pp. 248–253.

[34] P. V. Knudsen and J. Madsen, "Communication estimation for hardware/software codesign," in *Proc. CODES*, 1998, pp. 55–59.

[35] A. Nandi and R. Marculescu, "System-level power/performance analysis for embedded systems design," in *Proc. DAC*, 2001, pp. 594–604.

[36] A. Pinto, L. Carloni, and A. Sangiovanni-Vincentelli, "Constraint-driven communication synthesis," in *Proc. DAC*, 2002, pp. 783–788.

[37] K. K. Ryu and V. J. Mooney, III, "Automated bus generation for multiprocessor SoC design," in *Proc. DATE*, 2003, pp. 1531–1549.

[38] M. Gasteier and M. Glesner, "Bus-based communication synthesis on system level," in *Proc. ACM TODAES*, Jan. 1999, pp. 65–70.

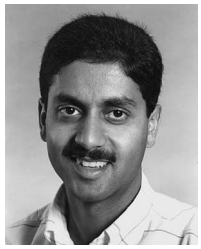
[39] D. Lyonnard, S. Yoo, A. Baghdadi, and A. A. Jerraya, "Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip," in *Proc. DAC*, 2001, pp. 518–523.

[40] S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Automated throughput-driven synthesis of bus-based communication architectures," in *Proc. ASPDAC*, 2005, pp. 495–498.
[41] U. Ogras and R. Marculescu, "Energy- and performance-driven NoC communication architecture synthesis using a decomposition approach," in *Proc. DATE*, 2005, pp. 352–357.
[42] A. Pinto, L. P. Carloni, and A. L. Sangiovanni-Vincentelli, "Efficient synthesis of networks on chip," in *Proc. ICCD*, 2003, pp. 146–150.
[43] A. Jalabert, S. Murali, L. Benini, and G. De Micheli, "XpipesCompiler: A tool for instantiating application specific networks on chip," in *Proc. DATE*, 2004, pp. 884–889.
[44] S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Extending the transaction level modeling approach for fast communication architecture exploration," in *Proc. DAC*, 2004, pp. 113–118.
[45] S. Pasricha, "Transaction level modeling of SoC with systemC 2.0," in *Proc. SNUG*, 2002, pp. 55–59.
[46] S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Constraint-driven bus matrix synthesis for MPSoC," in *Proc. ASPDAC*, 2006, pp. 30–35.
[47] S. Srinivasan, F. Angiolini, M. Ruggiero, L. Benini, and N. Vijaykrishnan, "Simultaneous memory and bus partitioning for SoC architectures," in *Proc. SOCC*, 2005, pp. 125–128.
[48] I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt, "Data reuse analysis technique for software-controlled memory hierarchies," in *Proc. DATE*, 2004, pp. 202–207.

Sudeep Pasricha (S'02) received the B.E. degree in electronics and communication engineering in 2000 from Delhi Institute of Technology, Delhi, India, and the M.S. degree in computer science in 2004 from the University of California, Irvine, where he is currently working toward the Ph.D. degree in computer science.

His research interests include design space exploration and synthesis of SoC communication architectures, design automation, and CAD for embedded systems, system modeling languages and methodologies, middleware for distributed systems, and computer architectures. He has filed for a U.S. patent, presented a tutorial on the topic of on-chip communication architectures at the Asia and South Pacific Design Automation Conference (ASPDAC) 2006, and coauthored over 20 technical papers.

Mr. Pasricha received a Best Paper Award Nomination at DAC 2005 and the Best Paper Award at ASPDAC 2006. He is a member of the Association for Computing Machinery (ACM).



Nikil D. Dutt (S'84–M'89–SM'96) received the Ph.D. degree in computer science from the University of Illinois at Urbana–Champaign in 1989.

He is currently a Chancellor's Professor of computer science (CS) and electrical engineering and computer science (EECS) with the University of California, Irvine (UCI), and is affiliated with the following centers at UCI: Center for Embedded Computer Systems, CPCC, and CAL-IT2. His research interests are in embedded systems design automation, computer architectures, optimizing compilers, system specification techniques, and distributed embedded systems.

Dr. Dutt received Best Paper Awards at CHDL89, CHDL91, VLSI-Design2003, CODES+ISSS 2003, and the Asia and South Pacific Design Automation Conference (ASPDAC)-2006. He currently serves as Editor-in-Chief of the *ACM Transactions on Design Automation of Electronic Systems* and as an Associate Editor of the *ACM Transactions on Embedded Computer Systems*. He serves or has served on the advisory boards of ACM SIGBED and ACM SIGDA, and is Vice-Chair of the International Federation for Information Processing Working Group (IFIP WG) 10.5. He has served on the steering, organizing, and program committees of several premier CAD and embedded system design conferences and workshops, including ASPDAC, Compilers, Architecture and Synthesis for Embedded Systems, CODES+ISSS, Design Automation and Test in Europe, International Conference on Computer Aided Design (ICCAD), International Symposium on Low Power Electronics and Design (ISLPED), and Languages, Compilers, and Tools for Embedded Systems (LCTES). He was an ACM Special Interest Group on Design Automation (SIGDA) Distinguished Lecturer during 2001–2002 and an IEEE Computer Society Distinguished Visitor in 2003–2005.